

D4.5.3 – Non-Blocking Collectives Runtime Library

WP4: Algorithms and Libraries

| | |
|--------------------------|---|
| Project Acronym | CRESTA |
| Project Title | Collaborative Research Into Exascale Systemware, Tools and Applications |
| Project Number | 287703 |
| Instrument | Collaborative project |
| Thematic Priority | ICT-2011.9.13 Exascale computing, software and simulation |

| | |
|--------------------------------------|---|
| Due date: | M24 |
| Submission date: | 30/09/2013 |
| Project start date: | 01/10/2011 |
| Project duration: | 36 months |
| Deliverable lead organization | CRAY |
| Version: | 1.0 |
| Status | Final |
| Author(s): | Pekka Manninen (CRAY) |
| Reviewer(s) | Jens Doleschal (TUD), Erwin Laure (KTH) |

| | |
|----------------------------|-------------|
| Dissemination level | |
| PU | PU - Public |

Version History

| Version | Date | Comments, Changes, Status | Authors, contributors, reviewers |
|---------|------------|------------------------------|--|
| 0.1 | 16/08/2013 | First draft for comments | Harvey Richardson (CRAY) |
| 0.2 | 31/08/2013 | Version for internal review | Dmitry Khabi (USTUTT) |
| 1.0 | 30/09/2013 | Final version for submission | Erwin Laure (KTH), David Henty (UEDIN), Jens Doleschal (TUD) |

Table of Contents

| | | |
|---|---|-----------|
| 1 | EXECUTIVE SUMMARY | 1 |
| 2 | INTRODUCTION | 2 |
| 3 | CRESTA COLLECTIVE COMMUNICATION LIBRARY | 3 |
| 3.1 | GENERAL DESCRIPTION | 3 |
| 3.2 | COLLECTIVE OPERATIONS | 4 |
| 3.3 | PERFORMANCE | 6 |
| 3.3.1 | <i>Performance comparison of different implementations.....</i> | <i>6</i> |
| 3.3.2 | <i>Overlap availability.....</i> | <i>9</i> |
| 4 | HOW TO USE THE LIBRARY..... | 10 |
| 4.1 | BUILDING THE LIBRARY | 10 |
| 4.2 | USING THE LIBRARY IN AN APPLICATION | 11 |
| 4.3 | THE SPLIT-PHASE API..... | 12 |
| 4.4 | NOTES ON USER SUPPORT | 13 |
| 5 | SUMMARY..... | 14 |
| 6 | REFERENCES | 15 |
| ANNEX A. FORTRAN AND C BINDINGS OF THE CRESTA COLLECTIVE COMMUNICATION LIBRARY FUNCTIONS | | 16 |

Index of Figures

| | |
|---|----|
| Figure 1: Available implementations of different collective operations (library v.0.1). These are subject to change in the future versions of the library. | 4 |
| Figure 2: The communication patterns of selected collective operations. | 5 |
| Figure 3: Performance of different implementations of the all-to-all data exchange operation..... | 7 |
| Figure 4: Performance of different implementations of the Allreduce collective communication operation. | 8 |
| Figure 5: Overlap availability for Alltoall and Allreduce operations..... | 9 |
| Figure 6: An example how to use the split-phase API (Fortran)..... | 12 |

1 Executive Summary

Most algorithms of scientific computing involve communication patterns that are performed collectively across a large number of processing elements. Hence, the scalability of many applications is often bound by collective operations in which even minor load imbalances or other inefficiencies during these phases can cause a stall across a significant number of processes. This holds also for the most of the CRESTA co-design applications.

In order to scale applications to hundreds of thousands of cores, new approaches for collective communication will be needed. These could be, for example, the use of asynchronous algorithms in combination with remote-memory access (also called one-sided) operations, especially when supported by hardware; utilization of non-blocking collectives that allow for overlapping the communication overhead with computation; optimization of communication patterns to improve concurrency but avoid interconnect contention, and so forth.

This document describes a platform for studying scalability bottlenecks caused by collective operations: the CRESTA Collective Communication Library. It basically allows for an application developer to experiment with various alternative implementations for a particular set of collectives with minimal changes into the application source. These implementations include in addition to the traditional collective operations of the message-passing interface (MPI) library the non-blocking collectives as introduced in the most recent version of the MPI standard, collectives implemented with partitioned global address space (PGAS) languages (yet currently only with Fortran coarrays) as well as with remote-memory access operations (also referred to as one-sided communication) available in the MPI library. Furthermore, it defines an application-programming interface (API) where the initiation and finalization of a collective operation are performed in separate stages, to allow for performing other work while the collective communication occurs in the background, here referred to as the split-phase API. The library itself is free software.

The capabilities of the library are described, together with some performance measurements, and user documentation is being provided. We found that on the Cray XC30 platform, applications could obtain substantial performance benefit from replacing the typical bottleneck collectives – the Alltoall data exchange operation and collective computation with the Allreduce operation – with the CRESTA Collectives; and selecting the Alltoall being implemented with Fortran coarrays and making Allreduce calls to utilize the split-phase API.

2 Introduction

In many supercomputing applications, a significant portion of execution time is spent on *collective communication operations*, i.e. communication patterns that involve all or a part of the parallel processing tasks in a synchronized fashion. In the de facto standard parallel programming approach, the message-passing interface (MPI), these communication patterns are available as convenient and optimized standalone library calls, such as MPI_Bcast for data replication, MPI_Gather for data collection, MPI_Reduce for collective computation and MPI_Alltoall for data exchange, to mention a few.

This deliverable documents the CRESTA Collective Communication Library, CCCL in short. It is a library that allows replacement of the most typical MPI collective operations with alternative CRESTA collectives, requiring only very minimal intrusion to the source code, e.g. the API arguments are exactly the same as with the original MPI collective. These CRESTA collectives perform exactly the same communication pattern and yield the same outcome as the original corresponding MPI collective. The library has various implementations available for the collective operations, including original, “blocking” collective operations as defined in the MPI standard; non-blocking collective operations introduced in the MPI standard version 3.0 [1]; implementations employing the coarrays feature of the Fortran 2008 standard [2]; and implementations using remote-memory access (also called one-sided communication) as defined in the MPI standard. (Note that the non-blocking MPI collectives were only standardized and an implementation made available during the course of this work.) The main purpose of the software is to act as a platform for experimentation of various collectives implementations, but it may also be directly useful for existing supercomputing applications.

The user of the software selects the implementation at compile time, makes the aforementioned minimal sentinel changes to his software and links with the CCCL. The library is available in separate language versions for applications written in Fortran or C/C++ programming languages. With some additional effort the library could be usable in applications written in other languages (e.g. Python) as well.

The motivation for using the alternative implementations is of course improved application performance. The performance of the alternative implementations is dependent on the target platform: the MPI library used, compiler used as well as the underlying hardware.

Additionally, the library defines an application programming interface (API) for initiating and finalizing a collective operation with two separate calls. The motivation for doing so is to try to hide the overhead from the operation by performing independent work while the collective is being progressed. Real-world scenarios benefitting from this feature include a classical molecular dynamics simulation employing the sc. particle-mesh Ewald (PME) method to account for long-range interactions – the bottleneck all-to-all communication pattern encountered in PME can be overlapped with e.g. evaluation of real-space quantities.

This document is structured as follows: Section 3 describes the features of the library and discusses its performance on one supercomputer platform (Cray XC30). Section 4 is the user documentation of the library, i.e. how to build the library itself and how to use it on an application. Some concluding remarks are being drawn in Section 5.

3 CRESTA Collective Communication Library

3.1 General description

The purpose of the CRESTA Collective Communication Library is to provide an easy interface for the application programmer to experiment with alternative approaches to possibly alleviate the bottleneck collectives, with minimal changes to the source code and fully without having to change algorithms or data structures.

The design is such that the programmer can convert some or all of the collectives to the CRESTA collectives, and is always able to return to the original MPI collectives with a simple library recompilation. The implementation is selected separately for different collectives, but such that all (e.g.) CRESTA_Bcast in the program are using the same implementation. The user needs to change the “MPI_” sentinel in a collective operation to “CRESTA_”, keeping the same call arguments.

The library is available as C and Fortran versions for easier interoperability with the program, but they differ in available implementations. In its initial scope, the library may implement a collective with:

- Original, “blocking” collective operations as defined in the MPI standard (all operations, both languages).
- Non-blocking collective operations introduced in the MPI standard version 3.0 (all operations, both languages). The completion of the operation is taken care of by the library – i.e. still no changes to the call arguments.
- Implementations employing the PGAS languages – i.e. coarrays feature of the Fortran 2008 standard (all non-vector collectives, Fortran only) and the Unified Parallel C extension of the C language [3] (not available yet).
- Implementations using one-sided operations as defined in the MPI standard (only few routines available, both languages).

| | Fortran | | | | C | | | |
|----------------|---------|------------------|------|----------------|-----|------------------|------|----------------|
| | MPI | Non-blocking MPI | PGAS | One-sided comm | MPI | Non-blocking MPI | PGAS | One-sided comm |
| Bcast | Y | Y | Y | Y | Y | Y | N | Y |
| Gather | Y | Y | Y | Y | Y | Y | N | Y |
| Gatherv | Y | Y | N | N | Y | Y | N | N |
| Scatter | Y | Y | Y | Y | Y | Y | N | |
| Scatterv | Y | Y | N | N | Y | Y | N | N |
| Reduce | Y | Y | Y | Y | Y | Y | N | |
| Allgather | Y | Y | Y | N | Y | Y | N | N |
| Allgatherv | Y | Y | N | N | Y | Y | N | N |
| Allreduce | Y | Y | Y | Y | Y | Y | N | |
| Reduce_scatter | Y | Y | Y | N | Y | Y | N | N |
| Alltoall | Y | Y | Y | Y | Y | Y | N | N |
| Alltoallv | Y | Y | N | N | Y | Y | N | N |

The feature list of the current scope of the library (v.0.1) is presented in

Figure 1.

The implementation is selected when building the library – see section 4.1, and the library needs to be linked into the application. Changing an implementation requires

| | Fortran | | | | C | | | |
|----------------|---------|------------------|------|----------------|-----|------------------|------|----------------|
| | MPI | Non-blocking MPI | PGAS | One-sided comm | MPI | Non-blocking MPI | PGAS | One-sided comm |
| Bcast | Y | Y | Y | Y | Y | Y | N | Y |
| Gather | Y | Y | Y | Y | Y | Y | N | Y |
| Gatherv | Y | Y | N | N | Y | Y | N | N |
| Scatter | Y | Y | Y | Y | Y | Y | N | |
| Scatterv | Y | Y | N | N | Y | Y | N | N |
| Reduce | Y | Y | Y | Y | Y | Y | N | |
| Allgather | Y | Y | Y | N | Y | Y | N | N |
| Allgatherv | Y | Y | N | N | Y | Y | N | N |
| Allreduce | Y | Y | Y | Y | Y | Y | N | |
| Reduce_scatter | Y | Y | Y | N | Y | Y | N | N |
| Alltoall | Y | Y | Y | Y | Y | Y | N | N |
| Alltoallv | Y | Y | N | N | Y | Y | N | N |

recompilation of the library and re-linking of the application.

Figure 1: Available implementations of different collective operations (library v.0.1). These are subject to change in the future versions of the library.

In addition to this non-intrusive alteration of implementation of collectives, the library features an application programming interface (API) for initiating and completing a collective operation in separate calls to allow for performing computation or other communication during a collective operation. This is referred to as the split-phase API. With it, each collective is started with a call to CRESTA_<collective name>_begin. Then the control returns to the program, allowing for an attempt to perform other work while the collective communication occurs. The collective has to be finalized with a call to CRESTA_Coll_end. Currently it is equal to the similar feature as available in recent MPI library implementations conforming to the 3.0 version of the MPI standard, but the reason for defining a new API is that also the split-phase API can be based on alternative implementations in the subsequent versions of the library.

The Fortran library routines are available only for INTEGER, REAL and DOUBLE PRECISION type data to be communicated. The C version accepts all kinds of elementary (native to C) data types. User-defined data types as available in MPI are not supported in the PGAS or one-sided communication implementations.

For the implementations with Fortran coarrays, only collectives over all the processes, i.e. operating on a communicator equaling to MPI_COMM_WORLD, are available. The other implementations do support user-defined communicators, however.

3.2 Collective operations

The collective operations included in CCCL are:

- CRESTA_Bcast
- CRESTA_Gather
- CRESTA_Gatherv
- CRESTA_Scatter
- CRESTA_Scatterv

- CRESTA_Reduce
- CRESTA_Allgather
- CRESTA_Allgatherv
- CRESTA_Allreduce
- CRESTA_Reduce_scatter
- CRESTA_Alltoall
- CRESTA_Alltoallv

Their meanings (outcomes) are exactly the same as their MPI_ counterparts. The topology-aware collectives introduced in the 3.0 version of the MPI standard are not yet available. Some of them are illustrated in Figure 2. The vector versions of certain collectives (e.g. Gather/Gatherv) have the same purpose but allow for varying-sized blocks of data being communicated. For further information on a particular collective, please consult the corresponding MPI collective from the MPI standard.

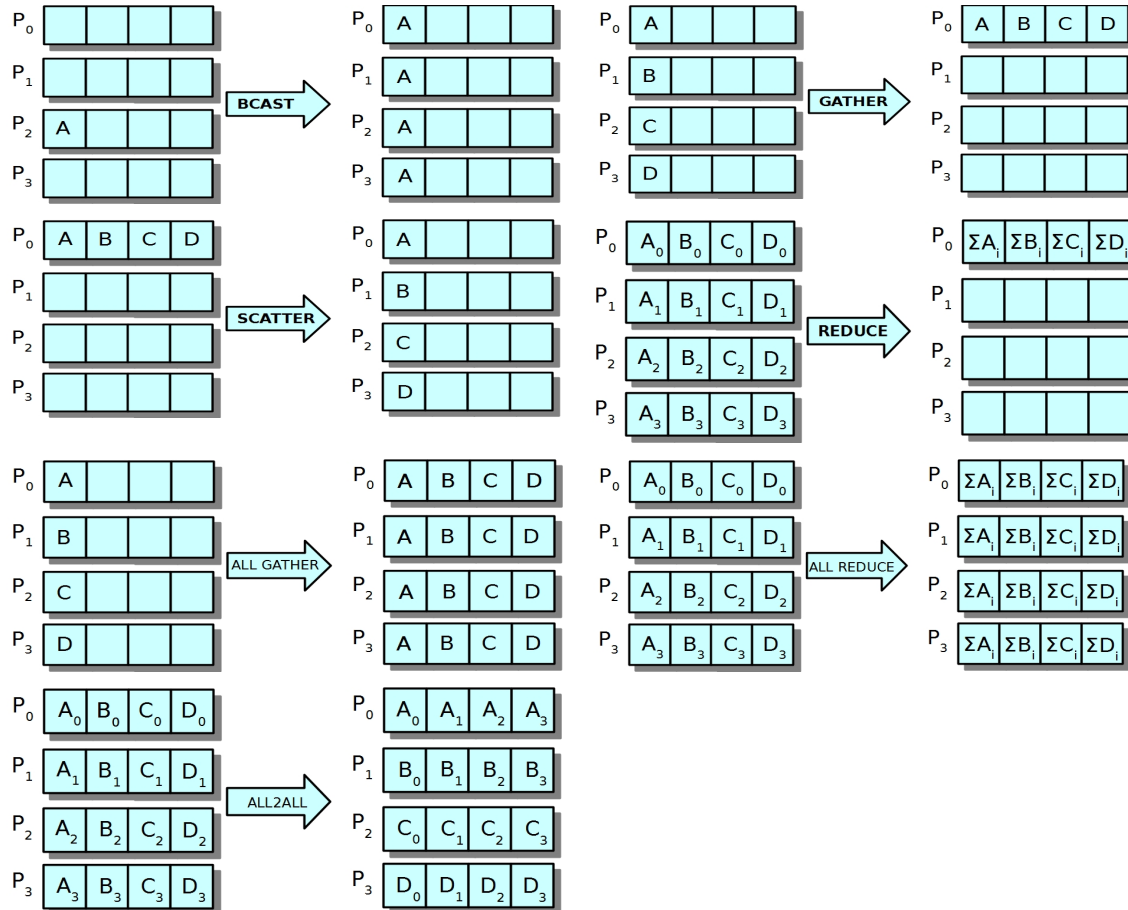


Figure 2: The communication patterns of selected collective operations.

3.3 Performance

As underlined earlier, the performance of the available realizations is heavily dependent on the various aspects of the underlying supercomputing platform: CPUs, interconnect, MPI library, compiler suite and so on, varying even in the same system over time as the software components evolve. In general, it is expected that the user is mostly interested in replacing the bottleneck collectives in his application with the CCCL alternatives; giving him an automated possibility to explore the most efficient implementation. The PGAS versions of the “cheap” collectives such as the one-to-all operations Bcast, Scatter and Gather are not optimized but very straightforward ones, and available mostly for the sake of completeness and as a starting point for further studies is optimizing these is of importance.

3.3.1 Performance comparison of different implementations

The typical bottleneck collectives in supercomputing applications in general are MPI_Alltoall(v) and MPI_Allreduce. Indeed, those have been identified to be the two bottleneck collectives in the CRESTA application suite [4]. We will discuss the performance of those. The test platform was a Cray XC30 machine (Intel Sandy Bridge CPUs, Cray Aries interconnect, Cray MPI library, Cray compiler suite). For the other collectives, especially fast one-to-all collectives, MPI implementations are much faster than the alternative implementations made during the course of work (coarrays or one-sided). The “compute and scatter” operation (MPI_Reduce_scatter) is an exception; coarrays implementation being comparable or slightly better to MPI.

The average time spent on performing the all-to-all data exchange as a function of the single message size (the total amount of communication being then $\#tasks^2 \times \text{message size}$) with 256, 1024 and 4096 PE's (i.e. a small and a medium-size job on current standards) is given in **Error! Reference source not found.** It can be observed that the MPI-based approaches (blocking or non-blocking without an overlap) yield similar performance. The approach based on Fortran coarrays outperforms MPI starting from message sizes of few tens of bytes. The one-sided approach is an order of magnitude slower than the others for small messages, but a cross-over with MPI is observed when the total sendbuffer size (message size * number of PE's) is around half a megabyte.

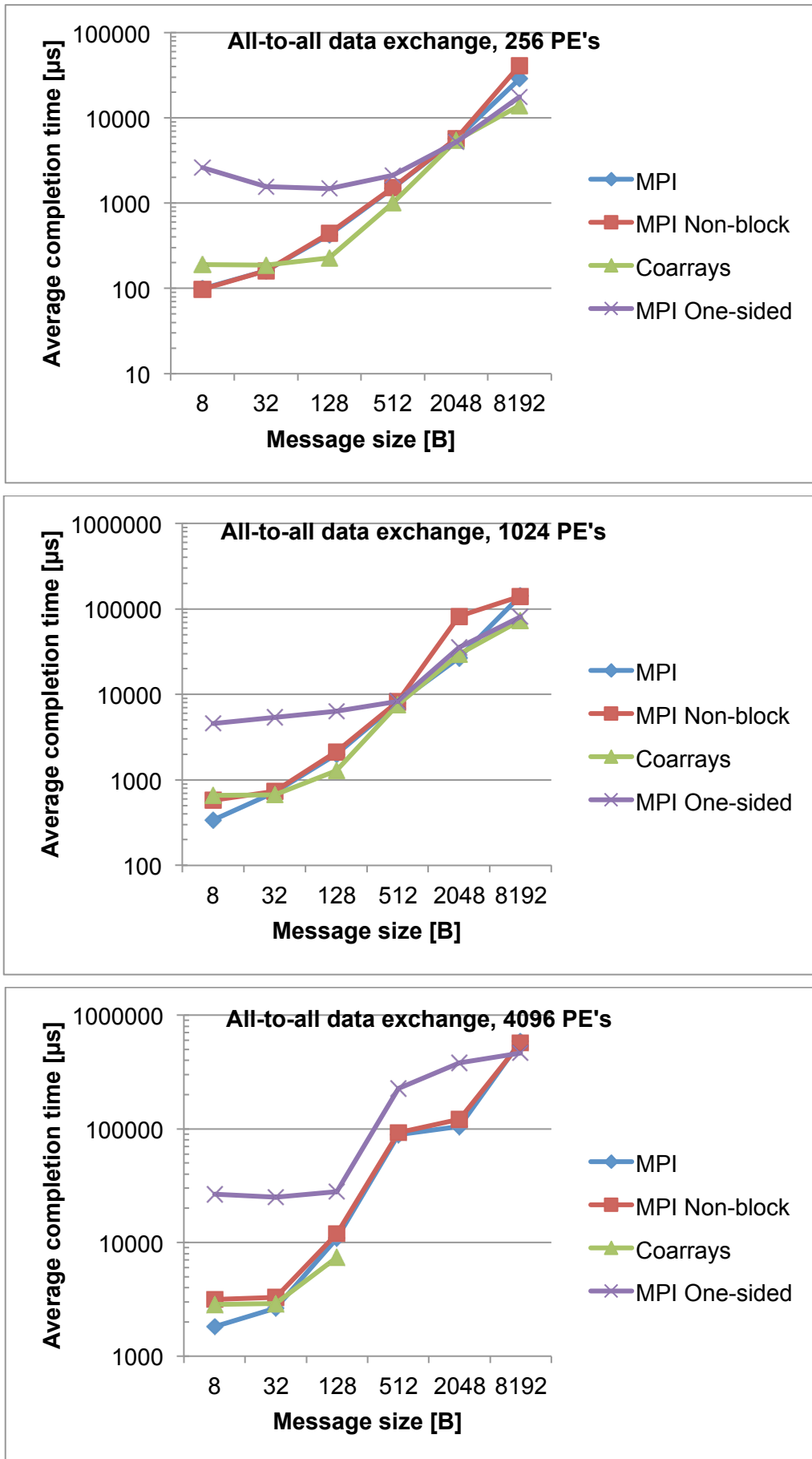
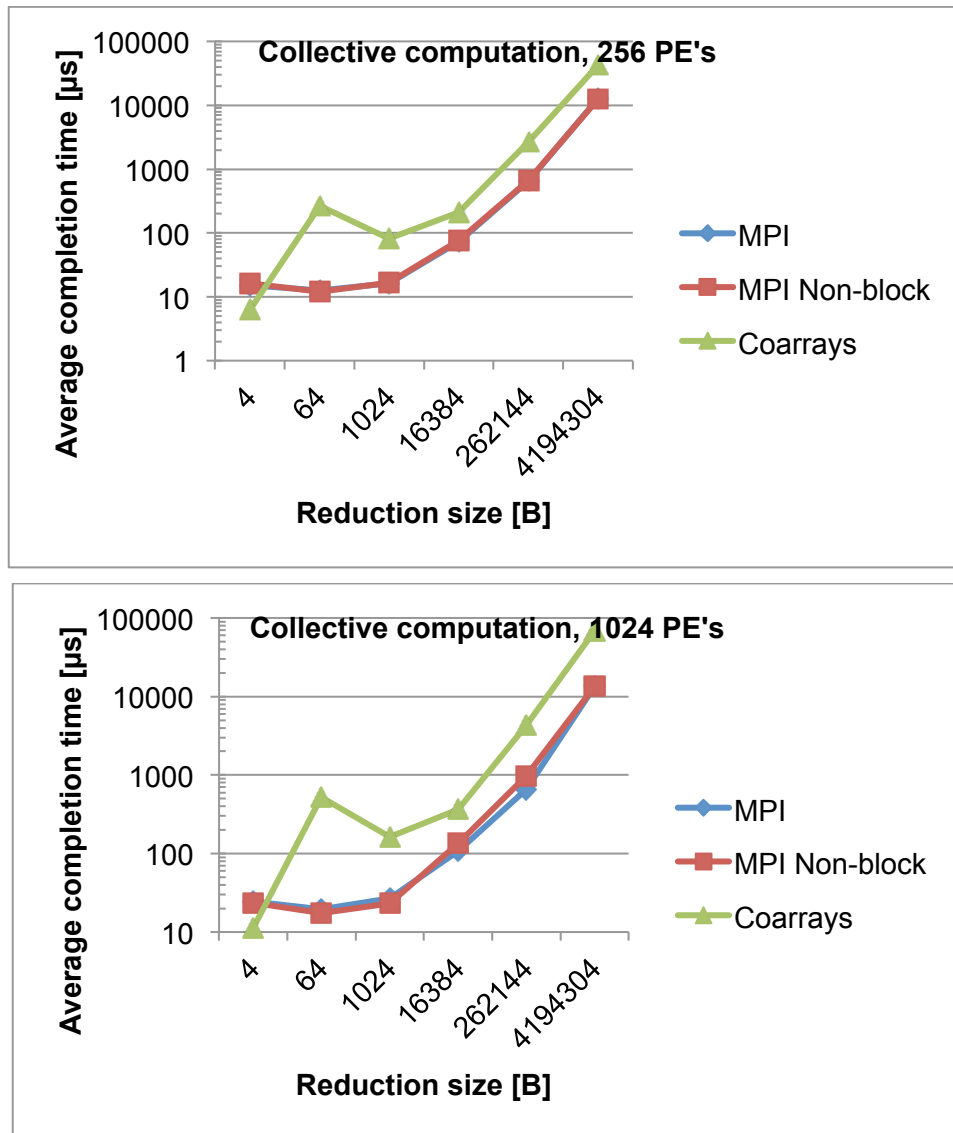


Figure 3: Performance of different implementations of the all-to-all data exchange operation.

Error! Reference source not found. presents the average time spent on performing the Allreduce operation (i.e. global reduction operation followed by the result replication to all tasks) as a function of the amount of data to be reduced with 256, 1024 and 4096 PE's. Here the approach based on Fortran coarrays is clearly slower than the MPI implementations. As yet we do not have an explanation for the kink observed at 64 bytes in the coarrays curve.



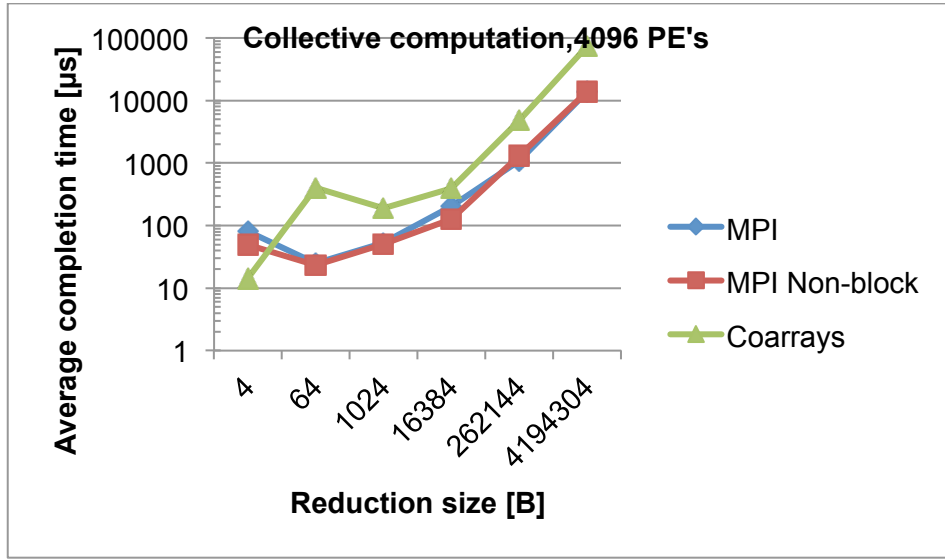


Figure 4: Performance of different implementations of the Allreduce collective communication operation.

3.3.2 Overlap availability

An important consideration from the point of view of exascale applications is how well the communication overhead related to collective operations can be “hidden” by doing something productive while waiting for the collective operation to complete. We define an “overlap availability time” O as

$$O = (C + N) - T$$

where

- T : a time required for completing a collective operation and an overlapped computational task
- C (computation time): time required for performing the computational task (i.e. operations on data not referred by the collective) alone
- N (network i.e. collective time): time required for completing the collective operation alone.

Further, by defining

$$\text{ovl\%} = \frac{(C + N) - T}{C + N - \max(C, N)} * 100 = \frac{O}{\min(C, N)} * 100$$

ovl% would be zero in case of no overlap (the case with blocking MPI collectives) and 100 in case where the communication overhead has completely been hidden (T being equal to either the computation or collective time, whichever is larger). Negative value would mean performance penalty from trying the overlap.

We have benchmarked the CCCL split-phase API for the overlap availability of the Alltoall and Allreduce operations as a function of the size of the communication, as presented in Figure 5 for a job of 1024 PEs on the Cray XC30, placing 16 MPI tasks per node (that contains 16 cores).

We observe that roughly half of the CPU resources during an Allreduce for smaller than four kB data would available for other work. This means an opportunity to cut to half the fraction of application wall-clock time spent on a bottleneck Allreduce. The overlap possibility reduces to zero for larger vectors than that. For Alltoall the availability is around 20%; 40% at the most. We however expect these ratios to improve over time as the MPI libraries improve.

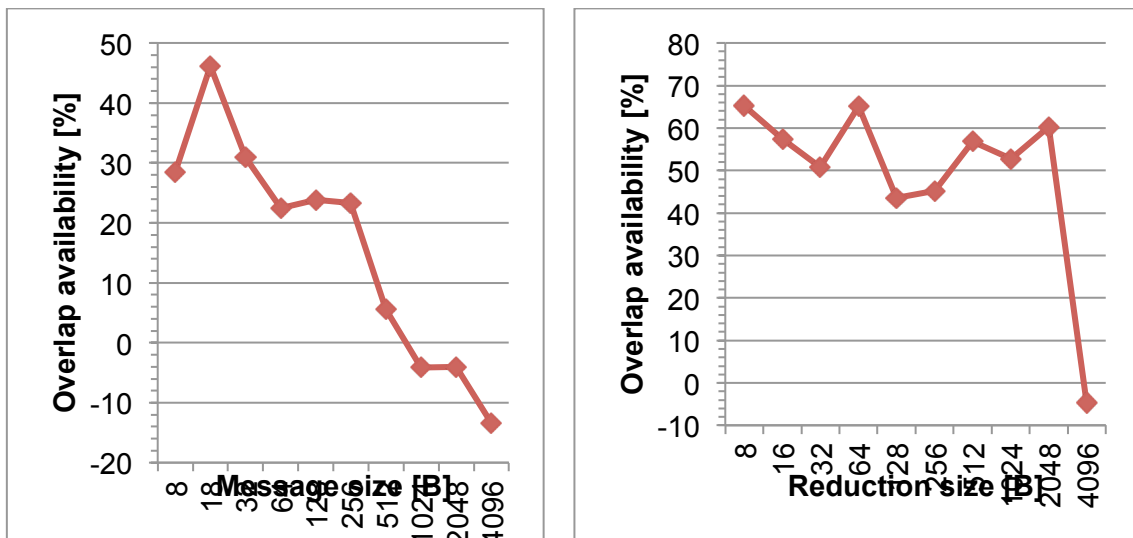


Figure 5: Overlap availability for Alltoall and Allreduce operations.

4 How to use the library

In this section step-by-step instructions on obtaining, building and using the library are given.

4.1 Building the library

1. Obtain the source code from the CRESTA subversion repository:
`svn co https://svn.ecdf.ed.ac.uk/repo/ph/cresta/wp4/cresta_libraries/cccl/trunk`
This requires an account to the subversion service. The package will be publicly available later.
2. Select the language version according to the application with which you intend to employ the library. Even if it is in theory possible to use the Fortran version of the library in a C/C++ software, in practice it is much easier to use the Fortran version of the library with Fortran software and C version within C programs.
3. Edit the corresponding Makefile. Insert the proper MPI compiler (e.g. mpif90) to the fields F90 and CC and related compiler flags.
4. Edit the field DFLAGS_LIB to select the implementation of certain collectives. Be aware that the alternative implementations provided by the library may be slower than the MPI implementations, or even non-functional, on your target platform. The syntax is `-D_<IMPLEMENTATION>_<COLLECTIVE>`, where IMPLEMENTATION is **CAF**, **OS**, or **NONBLOCK** for collectives written with Fortran coarrays, MPI one-sided operations or MPI 3.0 nonblocking collectives, respectively. COLLECTIVE is the name of the collective without the "MPI_" prefix, e.g. ALLREDUCE points to the MPI_Allreduce operation.
 - All desired special implementations have to be declared explicitly. If no request for a special build for a collective is given, the collective will be equal to blocking MPI. Note that not all collectives are available as Fortran coarrays or one-sided implementations.
 - The wildcard parameter `-D_ALL_NONBLOCK` will make all collectives to correspond the MPI 3.0 nonblocking version, unless they are not explicitly specified to be either CAF or OS.
 - Example: `DFLAGS = -D_CAF_ALLTOALL -D_OS_BCAST -D_NONBLOCK_ALLGATHERV` would use the Fortran coarrays implementation for MPI_Alltoall, one-sided operations for MPI_Bcast and MPI 3.0 nonblocking for MPI_Allgatherv. The rest of the routines would correspond to their original MPI versions.
5. Once finished with the Makefile, type "**make static**" (or "make") to build a static version the library. For a dynamic/shared library, type "**make shared**". After completed, you should have files libcrestacoll.a and/or libcrestacoll.so in the same folder. Typing "**make help**" will show all options.
6. There are a couple of small programs for verifying and benchmarking certain collectives. Type "**make test**" to build these, and run them in your target platform. If the output in any of these contains a statement "Error in CRESTA_<COLLECTIVE>", modify the Makefile such that this <COLLECTIVE> is not using any special implementation and rebuild the library. Typing "**make ovl**" will compile two benchmarks for the overlap availability when using the split-phase API.

There are further flags to control how to build the coarrays and one-sided versions.

- In the Fortran side of the library (only), both CAF and OS libraries can be built using either "static" or "dynamic" coarrays and communication windows. This means using a pre-allocated Fortran coarray or a common memory area that has a static communication window pointing to it. The benefits of "static" coarrays/windows are in avoiding repeated memory allocations, and these are a priori much faster than the default "dynamic" implementations. Insert flags `-D_STATIC_COARRAYS` and/or `-D_STATIC_OS_WIN` to enable these. When using

the one-sided routines, further source code modifications will be needed (see the next section). Static coarrays do not need any modifications beyond the Makefile and are a recommended practice.

- The size (memory allocation) of these static coarrays or windows are being controlled with the D-flags `-DTMP_COARRAY_SIZE=N` and `-DTMP_OS_WIN_SIZE=N` where N is the maximum number of elements in the coarray or window. Thus the suitable size for this parameter is the anticipated size of the largest buffer involved in a collective using either the coarrays or one-sided implementation. Note that in Fortran a separate coarray and memory window is being reserved throughout the program execution for integers, single-precision real numbers and double-precision real numbers, meaning that the memory consumption is $(4+4+8)*N$ (N as in the D flags above) bytes.

4.2 Using the library in an application

1. Add `#include "crestacoll.h"` into your C source code files and/or `USE crestacoll` into your Fortran source code files.
2. In the source code of your application, search MPI collective operations you would like to expose for the change of implementation, and change their sentinels from `MPI_` to `CRESTA_`, without changing the procedure call arguments. All collectives are recognized by the library. For example, all calls to `MPI_Alltoall` routines are being replaced with calls to `CRESTA_Alltoall`, not touching the list of arguments. It is safe to do this for all collectives, since it will be always possible to use the original MPI implementations.
 - A recommended practice is to employ pre-processor directives to enable co-existence with a version not requiring the CCCL. For example a call to `MPI_ALLTOALL(...)` would be replaced with

```
#if defined(_CRESTA_COLLECTIVES)
CALL CRESTA_ALLTOALL(...)
#else
CALL MPI_ALLTOALL(...)
#endif
```

Then the CCCL would be enabled by inserting `-D_CRESTA_COLLECTIVES` onto the compilation command of the file in question (and linking the library, see below).
 - This is the only modification needed besides when using one-sided implementations together with the "static" communication windows (see the building instructions). In this case one will need to insert a call to a routine `CRESTA_Win_init` before the first collective that employs the one-sided implementations. The `comm` argument is the MPI communicator the collective takes place at, and `rc` is an optional return code. In addition, one should add a call to `CRESTA_Win_finalize` after all operations have been completed. See the library or the annex of this document for the call syntax. A recommended practice is to add the `CRESTA_Win_init` call right after `MPI_Init`, and `CRESTA_Win_finalize` just before `MPI_Finalize`.
3. Build and link your software with adding a linker flag `-lcrestacoll` (possibly together with a pointer to the location of the library with `-L<directory path>`). In C programs, also the location of the CCCL .h files has to be provided to the compiler with `-I <directory path>`.

At this stage the application should be using the collective implementations selected during the library build phase.

Remember to validate the results and track the performance of your application, reverting back to the MPI implementations in case of any problems in either of those.

4.3 The split-phase API

The instructions outlined in Section 4.2 describe the basic, non-source-code-intrusive usage of the library. On top of that, the library features an application-programming interface (API) for initiating and ending a collective in two separate calls. The purpose of this API is to allow for overlapping computation or some other work while the collective communication takes place.

The intended use of the API is as follows:

1. Add **USE cresta_sp_coll** (Fortran) or **#include "cresta_sp_coll.h"** (C) into the beginning of the program unit utilizing the API.
2. Start communication by calling **CRESTA_<COLLECTIVE>_begin**. This will initiate the communication but return immediately in the similar fashion with MPI non-blocking communication, i.e. MPI_Isend/Irecv and the MPI 3.0 non-blocking collectives. The arguments for the function are equal to the ones of the corresponding collective, besides that the routine will return a **request handle** (of INTEGER type in Fortran, an MPI_Request struct in C) that the programmer has to store manually. The request parameter will always be the second last argument, just before the (optional) return value "rc".
3. Do computation or other meaningful work (communication, I/O, etc). The buffer communicated in the first step cannot be read or written in this stage.
4. Call **CRESTA_Coll_end** with the following input arguments: the number of requests and the request parameters (multiple parameters stored in an array). After this call returns, the communication initiated in the first step has been finished. This corresponds to calling MPI_Wait/Waitall. This one routine is being used for all the collectives. No status parameter will be returned. It is important to keep in mind that the communication is not guaranteed to have finished without completing this step. Omitting this step may lead to hard-to-trace deviations in results. Also any addressing of the buffers of the collective is not permitted before the CRESTA_Coll_end returns (unless the -D_PROTECTED_BUFFER flag has been enabled, see below).

An example code snippet utilizing the steps is presented in Figure 6.

```
...
use cresta_sp_coll
...
real(kind=8), dimension(:), allocatable :: sendbuf, recvbuf
integer :: n, request, rc
call CRESTA_Alltoall_begin(sendbuf, n, MPI_DOUBLE_PRECISION, &
    recvbuf, n, MPI_DOUBLE_PRECISION, MPI_COMM_WORLD, request, rc)
! do some computation, I/O or other communication not reading nor
! writing sendbuf or recvbuf here
...
call CRESTA_Coll_end(1, request)
! start utilizing information in recvbuf
...
```

Figure 6: An example how to use the split-phase API (Fortran).

Currently, the split-phase collectives that allow for the overlap are being realized only with the non-blocking MPI collectives – i.e. the user would get exactly the same functionality and performance by calling directly the corresponding non-blocking MPI collective in the step #1 and MPI_Wait (or MPI_Waitall for several on-going communication routines) in the step #3. The API has been defined to allow for selecting

also other implementations (based e.g. on Fortran coarrays) for realizing the overlap in the future. These implementations are to be added to the library in its subsequent versions.

There is one D-flag to control the building of the library: inserting `-D_PROTECTED_BUFFER` into the Makefile will compile the library such that it is safe to read or write over the send buffer involved in the collectives also before making a call to `CRESTA_Coll_end`. Without the flag the typical MPI rules of non-blocking communication apply (the buffers cannot be read or rewritten before the communication has been finished). This involves an additional memory copy, so it should be enabled only if being able to read or reuse the send buffer would bring some other benefit for the algorithm. This feature is not available for routines where the send and receive buffers are the same (Bcast). The receive buffer cannot be ever read or written before `CRESTA_Coll_end` has been called. Currently the feature is available only on the Fortran side of the library.

4.4 Notes on user support

The library is being distributed assuming the user understands its essence as experimental software, and that no warranties whatsoever are being given or implied about its performance or reliability. We also highlight that more than a directly applicable and polished API for supercomputing applications; it is a platform for benchmarking and experimenting with implementations of collective communication patterns and the primary initial use is within the CRESTA project itself

All bug reports, requests for enhancements, user experiences, benchmark data, code contributions etc. are gratefully received by the author, manninen@cray.com.

5 Summary

This document describes the capabilities and the usage of a new CRESTA Collective Communication Library.

- The library has been written to provide an easy interface for supercomputing application developers into various implementations of the typical collective communication operations, together with overlap of computation and collective communication.
- Enabling the CRESTA Collective Communication Library requires only minimal changes to the application source code. Changing an implementation needs recompilation of the library and relinking of the application with it.
- On the supercomputer platform where the performance was evaluated – a Cray XC30 supercomputer - the alternative implementation of the All-to-all data exchange based on Fortran coarrays outperforms those available in the message-passing interface library. It was also observed that another typical bottleneck collective, the Allreduce operation, can be overlapped with computation such that roughly one half of the execution time needed for an Allreduce operation can be utilized otherwise.
- The library is still in its early development state. More alternative implementations will be added into it and implementations optimized in the future.

6 References

- [1] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard version 3.0 (2012).
- [2] R. W. Numrich, J. Reid, SIGPLAN Fortran Forum 17, 1-31 (1998).
- [3] UPC Language Specifications, v1.2, Technical Report LBNL-59208, Lawrence Berkeley National Lab, (2005).
- [4] J. Nowell, D2.6.1 – CRESTA benchmark suite. CRESTA Project deliverable (2012).

Annex A. Fortran and C bindings of the CRESTA Collective Communication Library functions

A.1 Fortran bindings

For the CRESTA collective operations, refer to the corresponding collective operations of the MPI standard [1]. The routine is converted by changing the MPI_ sentinel into CRESTA_.

For the collectives using the split-phase API, an additional _begin suffix will be added to the collective name. E.g. MPI_Bcast becomes CRESTA_Bcast_begin. The argument list is the same in the corresponding collective, besides an extra request handle of type integer (output) as the second last argument. (That is, the list of arguments is the same as in a corresponding non-blocking collective).

For example

```
CRESTA_Bcast_begin (buffer, count, datatype, root, comm, request, rc)
```

```
<type>, dimension(...) :: buffer  
integer, intent(in) :: count, datatype, root, comm.  
integer, intent(out) :: request  
integer, intent(out), optional :: rc
```

The additional routines (that are a part of the API) introduced by CCCL are:

- CRESTA_Win_init(comm, rc)
integer, intent(in) :: comm
integer, intent(out), optional :: rc
Generates static memory areas used for the one-sided operations. Using them avoids generating a communication window every time a collective is posted.
- CRESTA_Win_finalize(rc)
integer, intent(out), optional :: rc
Frees the memory allocated for the static communication windows.
- CRESTA_Coll_end(count, request, rc)
integer, intent(in) :: count
integer, dimension(...), intent(in) :: request
integer, intent(out), optional :: rc
Completes the collective computation controlled by the request (single request or an array of requests).

A.2 C bindings

For the CRESTA collective operations, refer to the corresponding collective operations of the MPI standard [1]. The routine is converted by changing the MPI_ sentinel into CRESTA_.

For the collectives using the split-phase API, an additional _begin suffix will be added to the collective name. E.g. MPI_Bcast becomes CRESTA_Bcast_begin. The argument list is the same in the corresponding collective, besides an extra request handle of type integer (output) as the second last argument. (That is, the list of arguments is the same as in a corresponding non-blocking collective).

For example:

```
int CRESTA_Bcast_begin(void *buffer, int count, MPI_Datatype datatype,  
int root, MPI_Comm comm, MPI_Request request )
```

The additional routine (that is a part of the API) introduced by CCCL is:

- int CRESTA_Coll_end(int count, MPI_Request requests[])
Completes the collective computation controlled by the request (single request or an array of requests).