Extending C++ with Co-Array Semantics

Antoine Tran Tan *
Center for Computation and Technology
Louisiana State University
Louisiana, U.S.A.
atrantan@cct.lsu.edu

Hartmut Kaiser *
Center for Computation and Technology
Louisiana State University
Louisiana, U.S.A.
hkaiser@cct.lsu.edu

Abstract

The current trend of large scientific computing problems is to align as much as possible to a Single Programming Multiple Data (or SPMD) scheme when the application algorithms are conducive to parallelization and vectorization. This reduces the complexity of code because the processors or (computational nodes) perform the same instructions which allows for better performance as algorithms work on local data sets instead of continuously transferring data from one locality to another. However, certain applications, such as stencil problems, demonstrate the need to move data to or from remote localities. This involves an additional degree of complexity, as one must know with which localities to exchange data. In order to solve this issue, Fortran has extended its scalar element indexing approach to distributed structures of elements. In this extension, a structure of scalar elements is attributed a "co-index" and lives in a specific locality. A co-index provides the application with enough information to retrieve the corresponding data reference. In C++, containers present themselves as a "smarter" alternative of Fortran arrays but there are still no corresponding standardized features similar to the Fortran co-indexing approach. In this paper, we present an implementation of such features in HPX, a general purpose C++ runtime system for applications of any scale. We describe how the combination of the HPX features and the actual C++ Standard makes it easy to define a high performance API similar to Co-array Fortran.

Categories and Subject Descriptors E.1 [*Data Structures*]: Distributed data structures

General Terms Languages, Performance, Standardization

Keywords C++, API, Co-array, distributed containers, PGAS

1. Introduction

The recent advances in processor architecture has allowed modern supercomputers to perform more than one quadrillion floating point operations per second. In fact, the transition to the era of exaflops is expected to occur as soon as 2020. The key factors making such performance possible are parallel architectures. However, the gap between conventional sequential programming and parallel programming not only still remains wide, but is widening even further with the advent of ever more complex architectures, deepening memory hierarchies, an increasing amount of available processing units, and heterogeneous instruction sets.

For that reason, many research groups are now actively working on higher level approaches to reduce the complexity of writing parallel programs while maintaining a high level of performance. Among them we cite the *PGAS* model. The PGAS model is presented as a model that merges the very best from the two mainstream concepts which are shared memory programming and distributed memory programming. The feature of the model are the following :

- It preserves the concept of *Bulk Synchronous Programming* wherein one or more participant processors is executing the same code (Single Program Multiple Data) but with their proper data
- It preserves the conventional techniques, *i.e.* sequential programming or multi-threaded programming, when it is about processors working on *local* data.
- It bridges the gap between local and distributed worlds by allowing certain *global* data to exist in the memory of only one of the participating processors while being accessible by any other processor

The PGAS model has some limitations however. The decision which of data items will be located on what compute node has to be made statically, which make their runtime redistribution very difficult.

There are nowadays numerous implementations of the PGAS model. Those who strictly apply this model are UPC (UPC Consortium 2005), Titanium (Yelick et al. 1998) and Co-array Fortran (Numrich and Reid 1998). Those who choose to extend the model are X10 (Charles et al. 2005), Chapel (Chamberlain et al. 2007), and Charm++ (Kale and Krishnan 1993). In general, these last cited implementations choose to unlock the constraint of Bulk Synchronous Programming by assigning each processor a task queue and giving them the ability of creating new asynchronous tasks schedulable locally or remotely. Such model has the advantage of covering all the types of problems ranging from structured (or data-parallel) problems to unstructured (or task-parallel) problems.

Although asynchronous codes can overcome certain performance limits, they tend to progressively loose some expressiveness; this happens almost systematically when code writing goes from SPMD-style to MPMD-style. To solve this issue, we propose

* The STEllAR Group (http://stellar-group.org)

ARRAY'16, June 13-17, 2016, Santa Barbara, CA, USA

DOI: http://dx.doi.org/10.1145/2935323.2935332

in this paper to design a Co-array-like API but implemented on top of an asynchronous environment. More precisely, we focus on:

- Imitating as much as possible the functionalities of Co-array Fortran, because today it is part of the Fortran standard (Numrich and Reid 2005) and it intelligently adapts its array element indexing approach for references of remote objects.
- Relying on the HPX library (Kaiser et al. 2009, 2014) as it offers a best choice asynchronous environment, and a wide range of tools such as future objects, parallel algorithms, and distributed containers, while strictly conforming to the C++ standard.
- Utilizing new features of the C++ standard such as the variadic template parameters or the compile-time assertion checking.

In Sect. 2, we introduce the main ideas of co-arrays and describe how we adapt them for C++. Sect.3 presents the HPX library while focusing on key elements that help to implement the C++ co-arrays. Sect. 4 evaluates performance on two benchmarks and we discuss our results and future perspectives in Sect. 5.

2. Co-array C++

Our main goal in creating a co-array C++ API is to provide the key elements to implement a distributed code that is understandable for most developers including Co-array Fortran users. Additionally, we wanted to design an API that would be non-disruptive for C++ programmers and would take advantage of cutting edge features of the C++ Standard. Note that there is similar research which proposes to adapt the Co-array idiom for C++ (Eleftheriou et al. 2002; Johnson 2013). Our contributions that extent beyond these solutions are building an API on top of a asynchronous environment instead of message passing, proposing new semantics which can increase the clarity of co-subscripts (cf. Sect. 2.1), and supporting range-based operations. (cf. Sect. 2.2)

2.1 Co-subscript - based API

We assume first that a SPMD region, *i.e.* code fragment that is executed simultaneously by all (or some) of the available processors, has been created; each replica of this code is called an *image*.

```
//  Fortran Code
real :: z[10,*]

// C++ Code
spmd_block block;  ❶

coarray<double,2> z( block, "z"    ❷
                   , {10,_}    ❸
                   , partition<double>(1)    ❹
                   );
```

Listing 1. Instantiation of a co-array object : Fortran vs C++

The first noted difference compared with Fortran is the introduction of an object type called `spmd_block` ❶. A `spmd_block` encapsulates the information about the current *image*. This helper object allows to ease the runtime *co-creation* of the co-array `z`. For coordination aspects issued from the co-array object creation, a tag must be provided[1]; in listing 1 the C++ string `"z"` corresponds to this tag ❷. Then, the co-dimension ❸ is declared via a C++ `initializer_list`.[2] In summary, the listing 1 creates a 2-D co-array of elements, each element being of size 1. ❹. The size of the last dimension (described in the sample code by the symbol _) is equal by default to the number of current *images*. In listing 2, an image is performing a read from the standard stream and broadcasts the value to the other images.

```
real :: z[*]  ❷
...
if (this_image()==1) then
    read(*,*) z
    do image = 2, num_images()
        z[image] = z    ❶
    end do
end if

call sync_all()
```

Listing 2. Co-array Fortran sample code

One of the key operations in Co-array Fortran is the use of the operator (=) with a co-indexed l-value ❶. In this case, a *put* operation is performed ; the operation is synchronous for the image holding the r-value. Note in this example that the co-subscript of the r-value is not mentioned although the co-array is defined with 1-D co-dimension ❷. This means that the r-value is a local data (it is equivalent to say that the no mentioned co-subscript is implicitly equal to the identifier of the current image). See in listing 3 the C++ version of the previous code.

```
spmd_block block;
coarray<double,1> z( block, "z"
                   , {_}
                   , partition<double>(1)
                   );
...
if ( block.this_image() == 0 )  ❸
{
    std::cin  >> z.data(_);    ❶

    int num_images = block.get_num_images();  ❹
    for( int image = 1; image < num_images; image++ )
    {
        z(image) = z.data(_);  ❷ ❶
    }
}
block.barrier_sync("b");    ❺ ❻
```

Listing 3. Co-array C++ sample code

To avoid the ambiguity between a reference of local data and a reference of remote data while maintaining the dimensionality of the co-subscripts, we define two means to obtain a reference of a co-indexed element. If one would like to get the reference of a remote data, use the operator `()` ❷ , if one would like to get a reference of a local data, use the method `data()` ❶. Note, that in the latter case , the symbol _ must be specified as the *last dimension index*[3]. Obviously, if the co-array object has its *last dimension size*[4] not equal to _, the call to `data()` is considered to be *undefined behavior*.

Note that the synchronization operations ❺ , the number of current images ❹ and the identifier of the current image ❸ are implemented in forms of methods owned by the object `spmd_block`. Similarly to the creation of a co-array object, placing a barrier implies to dynamically create a global object ; hence the tag `"b"` ❻.

2.2 Iterator - based API

Like array scalar elements, one can extract *ranges* of Fortran co-indexed elements directly using Fortran co-subscripts. In C++, *ranges* are still not considered standard. The C++ standard library

[1] Each time a global object is created, a *Global Address Id* (cf. Sect.3) is attributed. This tag stands for a key that will be associated to this newly created GID.

[2] Note the Fortran symbol (*) which is replaced by the symbol (_) in C++.

[3] For a 3-D C++ coarray, the *last dimension index* is the third index

[4] For a 3-D C++ coarray of size {7,8,9}, the *last dimension size* is 9

is based on *iterators* instead. For this reason, we implemented the methods `begin()` and `end()` returning an iterator referring to the begin and end of a range respectively to help traversing all the co-indexed elements of a co-array object. Moreover, it can sometimes be useful to expose only the local elements during such traversal. We illustrate in listing 4 the two means to iterate over the co-indexed elements.

```
spmd_block block;
coarray<double,3> a ( block, "a"
                    , {4,4,_}
                    , partition<double>(5)
                    );
int idx = 0;
if ( block.this_image() == 0 )
{
    for (auto i = a.begin(); i != a.end(); i++ )
    {
        *i = std::vector<double>(5,idx++);
    }
    /* Equivalent to
        for (auto && proxy : a)
        {
            proxy = std::vector<double>(5,idx++);
        }
    */
}
block.barrier_sync("b");

auto alocal = local_view( a );

for (auto ii = alocal.begin();
     ii != alocal.end();
     ii++)
{
    std::vector<double> & ref = *ii;
    ...
}
/* Equivalent to
    for (std::vector<double> & ref : alocal)
    {
        ...;
    }
*/
```

❶ ❸ ❷ ❹

Listing 4. Traversal of co-indexed elements with iterators

In the first part ❶, image 0 traverses the full range of the co-array elements; here the dereference operator returns a proxy to the real reference. In the second part ❷, each image iterates over the co-indexed elements it possesses: the operator dereference is now returning a true local reference. Note that these operations can be performed using *range based for loops* ❸ ❹ that are now available since the C++ Standard 2011.

3. Distributed Containers and SPMD Regions in HPX

3.1 Distributed containers

To realize the data distribution that results from the co-array object creation, we use the class `partitioned_vector` available in HPX. This structure aims to simplify the tedious process of distributing partitions of a segmented structure to the different localities, and to facilitate their access from any locality. It further provides a logical shared view and a API very similar to that of C++ Standard `vector`.

When an object `partitioned_vector` is created, partitions made of standard vectors are created and placed in localities that can be specified by the user. During the creation of a partition, a *Global Address Id* [5] is attributed. These different GIDs are then listed and stored as a `partitioned_vector`'s attribute. See in listing 5 the code showing how to create a distributed container.

```
int N, n;
std::vector<hpx::id_type> locs
    = hpx::find_all_localities();

auto layout = hpx::container_layout(n, locs);  ❶

// Creation of the distributed vector
hpx::partitioned_vector<double> v( N, 0.0, layout); ❷
```

Listing 5. Creation of a distributed vector in HPX

In this code, `N` is the total number of scalar elements owned by the distributed vector, `n` is the number of partitions and `locs` is a vector of locality identifiers (of size `n`) which specifies the placement of each of the `n` partitions. From these two parameters, a `layout` object is created ❶ and passed as the last parameter ❷ of the distributed vector constructor.

To make the link with our API, we choose to implement the `coarray` class by using the concept of *view* that will be built on top of a distributed vector. Working on a co-indexed element is thus equivalent to work on a distributed vector partition. To translate a co-subscript into a partition vector subscript we choose the following convention: for an object co-array of codimension N and of size $(n_1, n_2, ..., n_N)$, the co-subscript $(i_1, i_2, ..., i_N)$ will correspond to the kth partition of the underlying distributed vector, k being equal to $i_1 + i_2 \times n_1 + ... + i_N \times (n_1 \times n2 \times ... \times n_{N-1})$.

Note that when the last dimension size of the co-array is defined equal to `_`, the layout of the underlying partitioned vector is such that all the co-subscripts sharing the same last dimension index are referencing partitions that are placed in the same locality.

3.2 SPMD Region

To define a SPMD region, we need to specify the different localities that participate in the bulk computation and an entry point from which to start. This entry point is defined by using a HPX action[6]. To allow all participant localities to know with which locality they have to work, this entry point must accept at least one parameter of type `spmd_block`. See in listing 6 a sample code in which a SPMD region is created.

```
void example_image(spmd_block block)
{   ...
}
HPX_DEFINE_PLAIN_ACTION(example_image, my_action);

int main()
{
    std::vector<hpx::id_type> locs
        = hpx::find_all_localities();

    // Invocation of the spmd region
    define_spmd_block( locs, my_action );

    return 0;
}
```

Listing 6. Creation of a SPMD region

Here the function `define_spmd_block` creates multiple images of the code `example_image` and launches them in each of the localities listed in the variable `locs`. A temporary object `spmd_block` is created and diffused to each image. When this function returns, an implicit barrier is placed to ensure that each image has been completed.

[5] A *Global Address Id* is the way to ensure the uniqueness of an object in the global address space.

[6] An HPX action is a wrapper allowing normal functions to be called remotely.

4. Performance Evaluation

For our performance evaluation, we performed all of our benchmarks on one node. This decision was made to allow us to compare our results with published benchmarks written for shared memory multicore architectures. We hope to observe that message exchanges from the *parcelport* layer[7] and management of the locality in a PGAS-style runtime system will have a sufficiently low impact on overheads and have a comparable performance to pure multithreaded codes. See in table 1 the characteristics of our machine.

Processor name	Intel Xeon Haswell
Number of cores	2 × 8 cores
Last level cache size (L3)	20 Mo
Number of NUMA nodes	2
Peak performance in simple precision	480 GFlop/s
Memory bandwidth	90 GB/s

Table 1. Test platform characteristics

4.1 Matrix Transpose

For our performance evaluation we first implement the matrix transpose algorithm. See in listing 7 the co-array C++ version of matrix transpose.

```
void transpose_coarray(  spmd_block & block
                       , coarray<double,2> & out
                       , coarray<double,2> & in
                       , int height
                       , int width
                       , int local_height
                       , int local_width
                       , int local_leading_dimension
                       )
{
    // Outer Transpose operation
    for (int j = 0; j<width; j++)
    for (int i = 0; i<height; i++)
    {
        // Put operation
        out(j,i) = in(i,j);
    }

    block.barrier_sync("outer_transpose");

    auto out_local = local_view(out);

    // Inner Transpose operation
    for (std::vector<double> & elt : out_local)
    {
        for (int jj = 0; jj<local_width-1;  jj++)
        for (int ii = jj+1; ii<local_height; ii++)
        {
            std::swap(
              elt[jj + ii*local_leading_dimension]
            , elt[ii + jj*local_leading_dimension]
            );
        }
    }
    block.barrier_sync("inner_transpose");
}
```

Listing 7. Transpose with co-array C++

In this version of the algorithm, we first perform a transpose by block (a block is accessed via co-array subscripts). We assume that coarray objects `out` and `in` have been predefined with an explicit last dimension size (different to _). In that case, the respective blocks are distributed in a *round-robin* maneer over the participating localities.

A barrier is placed thereafter to ensure that each block has arrived to its destination. Then, *in-place* transpose is performed for each block at scalar value level. Note that these two steps can potentially be parallelized using local tasks. We present, in figures 1 and 2, the scalability results for two different matrix orders (10000 and 22000). We compare our matrix transpose implementation with the OpenMP version, and with the optimized hand-written HPX version. Deep explanations about this latter version can be found in (Kaiser et al. 2015).

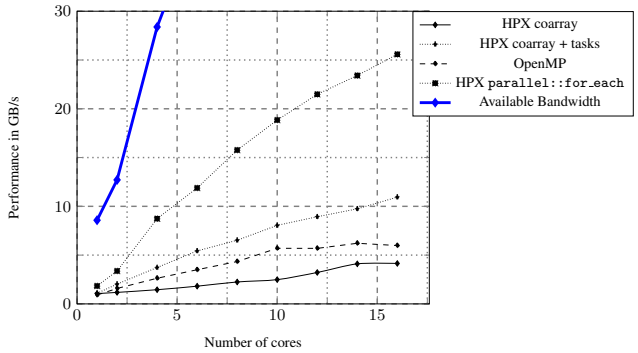

Figure 1. Performance comparison of Matrix Transpose codes - matrix order = 10000, tile order = 500

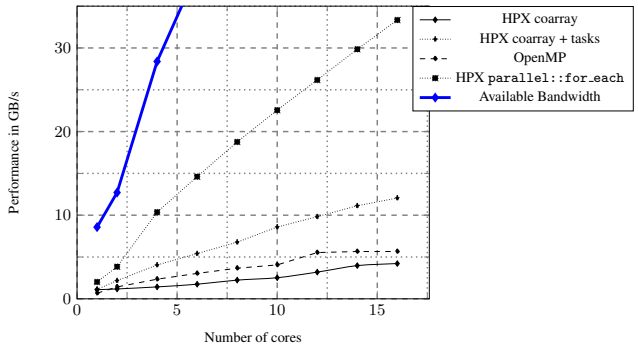

Figure 2. Performance comparison of Matrix Transpose codes - matrix order = 22000, tile order = 1100

We can see that the best version of Matrix Transpose code is the one where HPX calculations are parallelized manually via the use of `parallel::for_each`. Note that this version has a gain of up to 7 compared to the OpenMP version. The main reason is that the data arrangement in the HPX version is different : each matrix tile is stored contiguously in memory. For OpenMP, data keeps the conventional shared memory *Lapack* layout. Our C++ Coarray version uses just the number of images to achieve its proper scalability (one image per used core). According to this version, obtained performance remains inferior to the OpenMP version ; the main reason is that the algorithm we developed includes extra copies from the *in-place* transpose stage. We can nevertheless observe some scalability.

To improve this performance, we decide to fix the number of images to 1 and to locally parallelize the two stages of the algorithm implemented in the Coarray version. This change allows to surpass the performance of the OpenMP version. The data arrangement by

[7] *Parcels* in HPX are active messages for inter-locality communication. They are used to move the work to the data and to gather data back to the caller.

co-indexed blocks of the Co-array version is indeed similar to that used in the HPX version. But we are less better than the hand optimized HPX version once again because our implementation includes two copy steps in place of one.

4.2 Sparse Matrix Vector Multiplication

For our second performance evaluation we implement the sparse matrix vector multiplication (SpMV) algorithm. This operation has a computation complexity of $\mathcal{O}(nnz)$, with nnz the number of non-zero elements of the matrix.

For this benchmark, we choose to compare with the multithreaded version provided by Intel MKL. This implementation of the code performs the computation by using the CSR sparse matrix format and by using a combination of vectorized and OpenMP codes. In our case, we decide to extract only the vectorized part of the Intel MKL code (via a dependency to `mkl_sequential`) and to explore the possibilities of parallelism between CPU cores. See in listing 8 the co-array C++ version of SpMV.

```
struct spmatrix
{
    // Constructor definition ...

    int m_, n_, nnz_;
    std::vector<int> rows_, indices_;
    std::vector<double> values_;
    std::vector<int> begins_, sizes_;  (1)
};


void spmv_coarray( spmd_block & block
                 , spmatrix const & a
                 , std::vector<double> & x
                 , coarray<double,1> & y  (2)
                 , int unroll_factor
                 )
{
    int N = block.get_num_images();
    int image_id = block.this_image();

    int begin = a.begins_[image_id];
    int chunksize = a.sizes_[image_id];

    double * out = y.data(_).data();

    const int * row = a.rows_.data() + begin;
    const int * idx = a.indices_.data() + *row - 1;
    const double * val = a.values_.data() + *row - 1;

    for(int iter = 0; iter!=unroll_factor; ++iter) (4)
    {
        for(int i = 0;
            i < chunksize;
            i++, row++, out++)
        {
            double tmp = 0.;
            int end = *(row + 1);

            for( int o = *row;
                 o < end;
                 o++, val++, idx++
               )
            {
                tmp += *val * x[*idx - 1];
            }

            *out = tmp;
        }
    }
    block.barrier_sync("spmv");
}
```

(3)

Listing 8. SpMV with co-array C++

In our implementation, the sparse matrix and the operand vector are duplicated in each *image*. We define a `spmatrix` class that encapsulates the three tables of the CSR matrix and some helper arrays (1) to inform *images* of {matrix sub-block, *image*} correspondence. Only the output vector (2) is partitioned via a use of a co-array object. Note in this code that we let the hand-written code of the sub-block computation (3) but our results were obtained by using the `mkl_dcsrgemv` sequential kernel.

An important element to note is that we unrolled the test loop by a factor equal to `unroll_factor` (4) (a barrier synchronization is thus placed only once every `unroll_factor` iterations). The main reason for this choice is to put ourselves in the case in which there is no need for an *image* to regularly synchronize with the other images when several SpMV operations shall be effected in a sequence. Recent studies in the context of iterative methods in linear algebra (methods known for their high use of the SpMV kernel) have also shown by various approaches (Hoemmen 2010; Ghysels et al. 2013) that it is possible to overcome collective operations, at least partially, sometimes at a cost of additional calculations.

We test our codes by implementing two problems derived from the *Matrix Market Collection* web resource. We present in table 2 the key features of these systems.

Problem name	**s3dkt3m2**	**memplus**
Matrix order	90449	17758
Number of non-zero values	1921955	126150
Average distance from diagonal	339	4600

Table 2. Statistics of tested matrices

We can see in Figure 3 and 4 that the best version of the SpMv code is the HPX one when parallelization is done manually. In the case of problem *s3dkt3m2* (Figure 3), we note that the achieved performance exceeds the available memory bandwidth of the machine. The reason is that unrolling the test loop, promotes the data reuse, meaning that the data does not need necessarily to be fetch from the main memory at each iteration. In addition, the proximity of the non-zero elements from the diagonal elements will promote the contiguous loads of operand vector elements and therefore allows for better use of prefetch mechanisms in cache memory. In the case of *memplus* problem (Figure 4), performances we obtained are roughly at the same level of the matrix transpose performance. This result which is quite different in comparison to the first problem is caused by a greater dispersion of non-zero elements from the diagonal elements, reducing the contiguity of elements successively loaded from the operand vector.

By focusing on our implementation, we can see that the C++ Co-array version is overall better than the multithreaded Intel MKL version except with the *memplus* problem when we use more than 10 *images*. This gain is due in part because the test loop has been unrolled in the case of C++ Co-array and secondly because the sparse matrix and the operand vector are duplicated in each *image*. Although this solution consumes more memory, it allows to have better control of the locality especially in a NUMA environment. Note that the sparse matrix can be potentially partitioned instead of being replicated. Since our two problems are relatively small and our platform has enough memory, we have opted for the replication solution.

The decline in performance on the problem *memplus* is explained by the relatively small size of the sparse matrix (10 times

less non-zero elements than the problem *s3dkt3m2*), which accentuates the impact of the barrier in C++ Co-array since it is naturally more expensive than the one used in MKL. Using a hybrid approach[8], *i.e.* by adding local parallelism to the co-array code, we achieved performance that is closer to the performance of manually optimized HPX code, notably when the size of the problem in terms of non-zero elements is large enough. The management of local parallelism is done mostly by parallelizing the computation loop with the help of inner block counters and inner blocksizes.

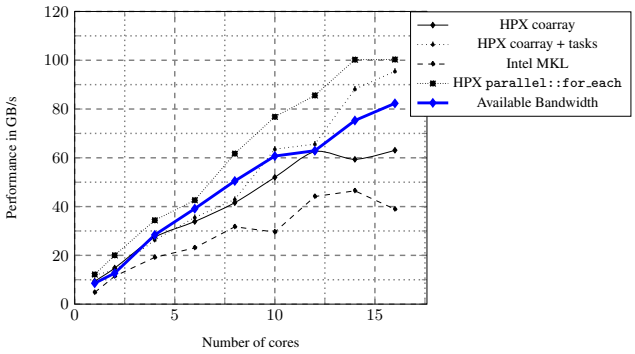

Figure 3. Performance comparison of SpMV codes with problem *s3dkt3m2* - unroll factor = 20

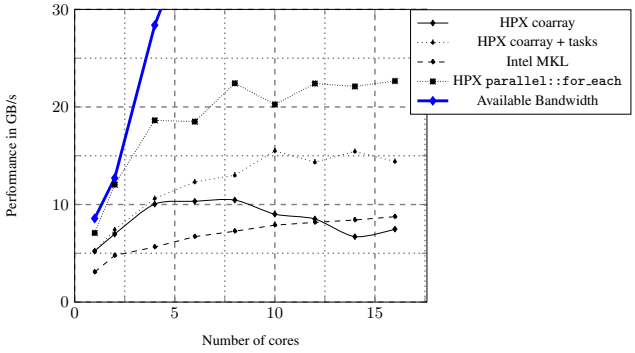

Figure 4. Performance comparison of SpMV codes with problem *memplus* - unroll factor = 20

5. Conclusion

In this paper, we propose a C++ API based on Co-array semantics which are known to be powerful for managing distributed parallelism. Compared to similar works, our API increases the clarity of element access, enables range operations, and enhances the manageability of the different participating processors. The key element of our API is that it is built on top of an asynchronous environment that is able to deliver significant performance. Our results show that reasonable scalability can be achieved by only relying on co-array constructs, but also highlights the importance of local parallelism to reach better performance. While utilizing finer grained approaches to parallelism can produce more performant and scalable code, the programmability offered by our API offers an efficient alternative to developers such as domain scientists. Ongoing work focuses on evaluating our API in larger scale applications and on gathering feedback from Co-array Fortran users to better tailor our API to their needs.

[8] Up to 8 cores, we launch only one image and the used CPU cores share the same NUMA node. After 8 cores, we launch 2 images (1 on each NUMA node) and balance the number of used CPU cores among the NUMA nodes.

Acknowledgments

The work described in this paper is supported by the National Science Foundation through award 1447831. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, Aug. 2007. ISSN 1094-3420. doi: 10.1177/1094342007078442. URL `http://dx.doi.org/10.1177/1094342007078442`.

P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.

M. Eleftheriou, S. Chatterjee, and J. E. Moreira. A c++ implementation of the co-array programming model for blue gene/l. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 1–, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1573-8. URL `http://dl.acm.org/citation.cfm?id=645610.661533`.

P. Ghysels, T. J. Ashby, K. Meerbergen, and W. Vanroose. Hiding global communication latency in the gmres algorithm on massively parallel machines. *SIAM Journal on Scientific Computing*, 35(1):C48–C71, 2013. doi: 10.1137/12086563X. URL `http://dx.doi.org/10.1137/12086563X`.

M. Hoemmen. *Communication-avoiding Krylov Subspace Methods*. PhD thesis, Berkeley, CA, USA, 2010. AAI3413388.

T. A. Johnson. Coarray c++. In *International Conference on PGAS Programming Models, PGAS*, volume 13, 2013.

H. Kaiser, M. Brodowicz, and T. Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*, pages 394–401. IEEE, 2009.

H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3247-7. doi: 10.1145/2676870.2676883. URL `http://doi.acm.org/10.1145/2676870.2676883`.

H. Kaiser, T. Heller, D. Bourgeois, and D. Fey. Higher-level parallelization for local and distributed asynchronous task-based programming. In *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*, ESPM '15, pages 29–37, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3996-4. doi: 10.1145/2832241.2832244. URL `http://doi.acm.org/10.1145/2832241.2832244`.

L. V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *IN PROCEEDINGS OF THE CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS*, pages 91–108, 1993.

R. W. Numrich and J. Reid. Co-array fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

R. W. Numrich and J. Reid. Co-arrays in the next fortran standard. *SIGPLAN Fortran Forum*, 24(2):4–17, Aug. 2005. ISSN 1061-7264. doi: 10.1145/1080399.1080400. URL `http://doi.acm.org/10.1145/1080399.1080400`.

UPC Consortium. Upc language specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005. URL `http://www.gwu.edu/ upc/publications/LBNL-59208.pdf`.

K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. In *In ACM*, pages 10–11, 1998.