ASPEN: An Efficient Algorithm for Data Redistribution Between Producer and Consumer Grids

Clément Foyer[1,2(✉)], Adrian Tate[1], and Simon McIntosh-Smith[2]

[1] Cray EMEA Research Lab, Bristol, UK
cfoyer@cray.com
[2] High Performance Computing Research group, Department of Computer Science, University of Bristol, Bristol, UK

Abstract. HPC applications and libraries have frequently moved parallel data from one distribution scheme to another, for reasons of performance. In modern times, a resurgence of interest in this *data redistribution* problem has emerged due to the need to relocate data distributed across one Producer grid onto a different distribution scheme across a Consumer grid. In this paper, we study the efficient algorithms to perform redistribution, and show how the best methods from the literature are still dependent on the number of processors in both grids. We describe a new algorithm ASPEN that exploits more cyclic patterns and relations in the distribution, is not dependent on the total number of processors and is thus well suited for use in a workflow management systems. We describe a preliminary implementation of the algorithm within such a workflow system and show performance results that indicate a significant performance benefit in data redistribution generation.

Keywords: Data distribution · Redistribution · Data placement
Data locality · Memory layout · Communication pattern
Parallel programming · Distributed memory

1 Introduction

Explicit data movement libraries and tools are used in HPC applications, coupled models, ensembles and workflows, to communicate data between distinct applications through various means. In many HPC workflows, a simulation running on M nodes (the *Producer Grid* or *Producer*) writes a large amount of data to another job running on a (possibly) distinct set of resources (the *Consumer Grid* or *Consumer*). Although this data movement pattern is far from new, it has become a common concern in modern times due to the prevalence of data-intensive workflows, coupled climate/environment applications and combined workflows of HPC with Data Analytics or AI. Many approaches exist to provide data movement between programs including in-situ frameworks, job couplers, in-memory databases and file-system approaches. In this paper we describe a

G. Mencagli et al. (Eds.): Euro-Par 2018 Workshops, LNCS 11339, pp. 171–182, 2019.
https://doi.org/10.1007/978-3-030-10549-5_14

library for communication of data between jobs over the interconnect fabric. Moving data over the interconnect, direct from DRAM has the benefit that many fewer data copies are incurred, but has the significant hurdle of needing to explicitly manage the parallel data movement in order to move the data. This problem of explicit data redistribution management is the focus of this paper.

A good deal of work (reviewed in Sect. 3) has explored the cost and benefits of explicit data redistribution, typically to a different distribution scheme within *the same processor grid*. While sharing many qualities with the classical data redistribution problem, so called *Producer-Consumer redistribution* or *M:N node redistribution* exhibits significant additional complications arising from the fact that the two grids reside in different jobs and lack awareness of the other's characteristics including distribution scheme. Cray has developed a library called the *Universal Data Junction* (UDJ)[1] that provides the missing information and allows distinct jobs in distinct grids to package, send and receive parallel (distributed) data over the high-performance interconnect as well as other resources that may be preferred.

In this work, we focus on the algorithmic machinery that is required in order to allow a Producer and a Consumer Grid to communicate the correct data, at minimal expense in a scalable fashion. The reason to place so much emphasis on the cost of redistribution is that the operations cannot easily be offloaded or performed asynchronously and thus incur direct overhead on the simulation code, which is often intolerable. In Sect. 3 we show that classical algorithms and those in the literature display running times that are proportional to the number of *remote* processors from the perspective of either the Producer (i.e. *remote* means Consumer) or the Consumer (i.e. *remote* means Producer) grid. In the Exascale era, it is expected that simulation jobs may run on millions of compute cores. Hence, this dependence on remote grid size is intolerable. In Sect. 4 we describe a new approach that exploits three types of periodicity in cyclic data distributions, resulting in a lower complexity redistribution algorithm and one that does not depend on remote grid sizes. In Sect. 5 we show the results of our new approach versus the classical algorithm and some of the most used and well-regarded algorithms published in the literature.

2 Background

We define the regular redistribution problem in the same way as [1] using updated producer-consumer terminology: given a d-dimensional array A on a set of Producer resources (processors and memory) $\mathscr{R}_{producer}$ that uses some distribution scheme $\mathscr{D}_{producer}$ we wish to move all the data to another set of resources $\mathscr{R}_{consumer}$ using some other distribution scheme $\mathscr{D}_{consumer}$. $\mathscr{D}_{producer}$ and $\mathscr{D}_{consumer}$ represent arbitrary array element mappings across each dimension of the array.

The global array indices of A are given by $G_1, \ldots, G_d$. The set of distribution schemes of primary interest are BLOCK, CYCLIC, 1-d BLOCK CYCLIC and

[1] https://gitlab.com/cerl/universal-data-junction.

k-d BLOCK-CYCLIC. Since BLOCK, CYCLIC and 1-d CYCLIC are special cases of k-d BLOCK-CYCLIC, we study only the latter in this paper. Like [1,7] we use a *Local Data Descriptor* approach, but we choose to ignore this representation since it is an implementation feature not relevant to the algorithmic descriptions. Local data sizes of A on rank p are given by $L_1^p, L_2^p, \ldots, L_d^p$. Processors compose a d-dimensional processor grid $p_1 \times \cdots \times p_d$ where $p_i (1 \leq i \leq d)$ gives the number of processors in the grid dimension i. We will discuss two such processor grids $\mathscr{G}^{producer}$ and $\mathscr{G}^{consumer}$ where the resources are assumed to be distinct though this is not necessary. We define the mapping $G2L(p, d)$ as the function that maps global indices to the local indices for processor p in dimension d, and the inverse relation $L2G(p, d)$ mapping local indices to global indices.

The non-triviality of redistribution of cyclic data can be illustrated by the graphic example of Fig. 1. A 2-d array is divided into 2-d partitions using some block sizes b_1^1, b_2^1. The blocks of this partitioning are distributed using a 2d block-cyclic distribution scheme across a producer grid of size 4×4. We denote the block ownership by labelling the blocks by processor owner, round-robin style along each dimension (Fig. 1a). We wish to redistribute the same 2-d data across a different consumer grid of size 3×3 using different block sizes b_1^2, b_2^2 labelled similarly (Fig. 1b). For any process pair (p, c) where p is in the producer grid and c is the consumer grid, we can overlay the global data owned by each processor to begin to ascertain shared indices, e.g., producer process $(0, 0)$ (Fig. 1b) and consumer process $(0, 0)$ (Fig. 1c) superimposed in Fig. 1d. The intersection of the superimposed data in Fig. 1e represents the global indices that these two processors must directly exchange over the network (i.e., producer $(0, 0)$ must send these indices and consumer $(0, 0)$ must receive these indices). The d-dimensional situation is a direct extension of the illustrated 2-dimensional case.

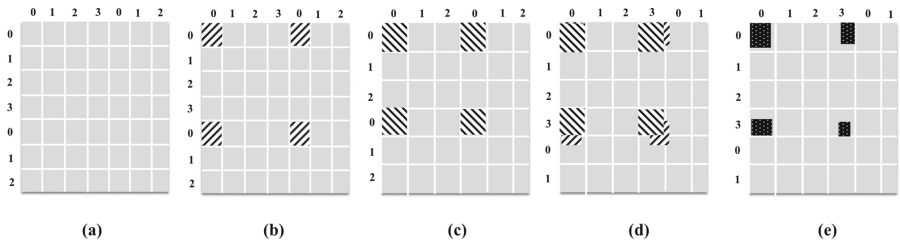

Fig. 1. Example of non-triviality of data index calculations for trivial distribution across 4×4 producer and 3×3 consumer grids

3 Related Work

The question of parallel data redistribution has been addressed many times, both statically and dynamically as this question was central when dealing with the imposed data distributions of early distributed memory programming models such as High Performance Fortran (HPF) [4].

Extensive analysis has been performed on both the nature of block-cyclic distribution, and its relevance to distributed memory relations as it stands as a generalisation of both block distribution and cyclic distribution. Multiple improvements have been proposed taking advantage of certain characteristics of this kind of data distribution [1,2,6,7]. These solutions also focused on the message scheduling part of array redistribution, which is out of scope for this paper. Petitet and Dongarra [5] described techniques for redistribution taking into account the severe alignment restrictions induced by the architecture, as well as further treatment of the scheduling.

Thakur et al. compared different solutions and presented solutions both specific and general varying the size of the blocks but restricted to fixed size process grids [8,9]. The presented techniques rely either on computing the source and destination for each element of the array outside of where it was possible to use improvements due to any common factors between the two block-cyclic sizes. Hsu et al. [3] also described some optimisations for specific cases where the two block-cyclic distributions have a common factor, but with irregular number of processors. In their more generic approach [2] the authors provide a thorough proof of the algorithm. Our version, although very close in the principle, is based on LDD usage and does not enforce the sizes to be relatively prime numbers, allowing simpler generation of the final scalar product.

4 Data Redistribution Algorithms

From the literature, the redistribution problem has been expressed as: given two possibly different regular distributions of data over two grids P and C with distributions D_P and D_C, for each pair of processes $(p \in P, c \in C)$ find the intersecting elements in $D_p \cap D_c$. The general idea when addressing the redistribution problem is to compute the intersecting `blocks` of data between the ranks. Each block is characterized by its starting index, its length, its dimension and its destination. As stated in [7], a multidimensional distribution can be expressed as a cross-product of multiple one-dimensional distributions. Using this approach, the general solution for a multidimensional approach is presented in Sect. 4.1, while Sects. 4.2, 4.3 and 4.4 focus more specifically on the required block comparisons.

4.1 General Problem

Algorithm 1 presents the outer loop over the dimensions needed to generate the block sets describing how to scatter the local data. In this algorithm, `ComputeIntersection` refers to any of the algorithms presented in Sects. 4.2, 4.3 and 4.4. The version described here considers the computation of the complete redistribution. It is however possible to pass to Algorithms 2 or 3 the remote process coordinates in order to compute the unique intersection with the local process. As the full description of the intersecting blocks is created with a crossproduct, any empty returned $\texttt{block}^{\texttt{d}}$ would allow the algorithm to finish early in this case.

Algorithm 1. Base algorithm for redistribution

```
/* Each rank performs the following for each dimension d                    */
input  : Producer Grid P, Consumer Grid C, Producer Distribution D_P,
         Consumer Distribution D_C
output: Set of blocks^d
for d ← 1 to ndims(Data) do
    blocks^d ← ComputeIntersection(P^d, C^d, D_P^d, D_C^d);
end
```

Algorithm 2. Classical Redistribution Algorithm

```
/* Each rank performs the following for each dimension d                    */
input  : Producer Grid P^d, Consumer Grid C^d, Producer Distribution D_P^d,
         Consumer Distribution D_C^d
output: Int_rank consists of tuples (remoteRank, start, end)
N_local ← Number of local blocks owned by this rank;
for remote ← 1 to |C^d| do
    N_remote ← Number of local blocks on remote rank;
    for localBlockId ← 1 to N_local do
        localBlock ← getBlock(D_P^d, localBlockId);
        for remoteBlockId ← 1 to N_remote do
            remoteBlock ← getBlock(D_C^d, remoteBlockId);
            Left ← max(localBlock.start, remoteBlock.start) ;
            Right ← min(localBlock.end, remoteBlock.end) ;
            if Left < Right then
                Int_rank ← Int_rank ∪ (remote, Left, Right) ;
            end
        end
    end
end
```

4.2 Classical Algorithm

This algorithm presents the naïve way of computing the intersection, by taking each block of the given local distribution and looking for an overlap by comparing its boundaries with those of each block of the remote distribution. This comparison is performed for each process of the remote grid.

The total number of operations is given by

$$\mathtt{Ops}_{Classical} = D \cdot L \cdot R \cdot N^{local} \cdot N^{remote} \tag{1}$$

where N^{local} represents the number of local blocks and L represents the number of processes in the local grid dimension, respectively remote blocks and remote grid dimensions are N^{remote} and R.

Algorithm 3. FALLS Redistribution Algorithm

```
   /* Each rank performs the following for each dimension d                    */
   input : Producer Grid P^d, Consumer Grid C^d, Producer Distribution D_P^d,
           Consumer Distribution D_C^d
   output: Int_rank consists of tuples (remoteRank,start,end)
 1 S_local ← local stride between blocks;
 2 S_remote ← remote stride between blocks;
 3 S ← lcm(S_local, S_remote);
 4 N_local ← Number of local blocks owned by this rank;
 5 for remote ← 1 to |C^d| do
 6     N_remote ← Number of local blocks on remote rank;
 7     for localBlockId ← 1 to max(N_local, S / S_local) do
 8         localBlock ← getBlock(D_P^d, localBlockId);
 9         firstIndex ← max(0, ⌈(localBlock.start − remoteOffset − remoteBlocksize) / S_remote⌉);
10         lastIndex ←
           min(1 + (localBlock.start + localBlocksize − remoteOffset) / S_remote, N_remote, S / S_remote);
11         for remoteBlockId ← firstIndex to lastIndex do
12             remoteBlock ← getBlock(D_C^d, remoteBlockId);
13             Left ← max(localBlock.start, remoteBlock.start);
14             Right ← min(localBlock.end, remoteBlock.end);
15             if Left < Right then
16                 for disp ← 0 to |Data^d| / S do
17                     start ← Left + disp × S;
18                     end ← Right + disp × S;
19                     Int_rank ← Int_rank ∪ (remote, start, end);
20                 end
21             end
22         end
23     end
24 end
```

4.3 FALLS Algorithm

This algorithm is the best version found in the literature for *M:N node redistribution*. The same idea is expressed in [1,7], and summarized in Algorithm 3. Although comparing boundaries block-by-block, these articles present a huge improvement over the classical algorithm in terms the number of block comparisons required.

The bounds are reduced by using the fact that the intersection of two block cyclic distributions can be expressed as the union of some set of block cyclic distributions, each origin being the beginning of the intersection, each block length being the length of the intersecting block and the distance between blocks[2] is equal to the lower common multiple of the two original strides. The result is that it is only necessary to compare blocks within one stride S. In other words,

[2] Later referred to simply as *stride*.

for each element x in the found intersection then $x + n(S)$ is also inside the intersection, where n is all integers for which $x + n(S)$ remains smaller than the extent of the full array. Additionally, it is only necessary to compare the blocks in the onwards direction and those already checked can be ignored.

Compared to the classical algorithm, the reduction of the bounds reduces drastically the number of blocks to be considered when evaluating the intersection for big grids of data. The total number of operations is given by

$$\texttt{Ops}_{\text{FALLS}} = D \cdot L \cdot R \cdot \hat{N}^{local} \cdot \hat{N}^{remote} \tag{2}$$

where the $\hat{N}^{local}$ and $\hat{N}^{remote}$ represent the reduced number of local blocks due to searching only with one S. In theory then, Eq. 2 resembles Eq. 1 but in practice, $\hat{N}$ and N typically differ greatly with $\hat{N} \ll N$.

4.4 ASPEN Algorithm Description

To improve redistribution performance, we develop a scheme that can exploit further qualities of the periodic nature of the distributed data, and the known relationships between adjacent blocks. We call the approach *Adjacent Shifting of PEriodic Node data* or ASPEN. To illustrate the approach we first describe the two remaining weaknesses of the existing algorithms.

Periodicity of Remote Block Data. In the Algorithms 2 and 3 each local block's position in the global scheme is compared against multiple remote blocks (all remote blocks in the case of Algorithm 2 and many fewer than all remote blocks in the case of 3). In fact, the need to perform more than one comparison ignores periodic qualities of the data distribution since the constant stride should enable a direct periodic comparison. Consider the code in Algorithm 3 lines 11–14. This code searches over the loop of remote blocks (Algorithm 3 line 11) to generate all remote $RemoteBlockID$s, then inside that loop $remoteBlock$ is extracted using $getBlock(D^d_C, remoteBlockId)$ (Algorithm 3 line 12). $Left$ and $Right$ are then both generated using various extents of $localBlock$ and $remoteBlock$ (Algorithm 3 lines 13 and line 14). We can avoid this logic if we generate a *periodic offset* as follows

$$offset \leftarrow localBlock.start \mod remoteBlocksize$$

Offset can be seen visually in Fig. 2 and appears in Algorithm 4 line 8.

The $offset$ can be used to indirectly obtain the same information, without doing explicit comparisons to individual remote blocks, by checking the inequality

$$localBlock.start - offset + remoteBlocksize \leq localBlock.end \tag{3}$$

If condition Eq. 3 is true, then this particular local and remote block comparison overlaps on the left-hand side of the local block. This can be understood by seeing that the blue box of Fig. 2 would be non-empty when Eq. 3 holds.

Algorithm 4. ASPEN Redistribution Algorithm

```
   /* Each rank performs the following for each dimension d                    */
   input : Producer Grid P^d, Consumer Grid C^d, Producer Distribution D_P^d,
           Consumer Distribution D_C^d
   output: Int_rank consists of tuples (remoteRank,start,end)
 1 S_local ← local stride between blocks;
 2 S_remote ← remote stride between blocks;
 3 S ← lcm(S_local, S_remote);
 4 N_local ← Number of local blocks owned by this rank;
 5 for localBlockId ← 1 to max(N_local, S / S_local) do
       /* GTL is a global to local index conversion function.                  */
 6     localBlock ← getBlock(D_P^d, localBlockId);
 7     remote ← (localBlock.start / remoteBlocksize) mod |C^d|;
 8     offset ← localBlock.start mod remoteBlocksize;
 9     Left ← localBlock.start;
10     if localBlock.start − offset + remoteBlocksize ≤ localBlock.end then
11         Right ← min(localBlock.start − offset + remoteBlocksize, |Data^d|);
12         diff ← Right − Left;
13         for ps ← Left to |Data^d| by S do
14             start ← G2L(ps);
15             end ← G2L(min(ps + diff, |Data^d|));
16             Int_rank ← Int_rank ∪ (remote, start, end);
17         end
18         Left ← Right;
19         remote ← (remote + 1) mod |C^d|;
20     end
21     for Left to min(localBlock.end, |Data^d|) by remoteBlocksize do
22         Right ← min(Left + remoteBlocksize, localBlock.end, |Data^d|);
23         diff ← Right − Left;
24         for ps ← Left to |Data^d| by S do
25             start ← G2L(ps);
26             end ← G2L(min(ps + diff, |Data^d|));
27             Int_rank ← Int_rank ∪ (remote, start, end);
28         end
29         remote ← (remote + 1) mod |C^d|;
30     end
31     Right ← min(localBlock.end, |Data^d|);
32     if Left ≤ Right then
33         diff ← Right − Left;
34         for ps ← Left to |Data^d| by S do
35             start ← G2L(ps);
36             end ← G2L(min(ps + diff, |Data^d|));
37             Int_rank ← Int_rank ∪ (remote, start, end);
38         end
39     end
40 end
```

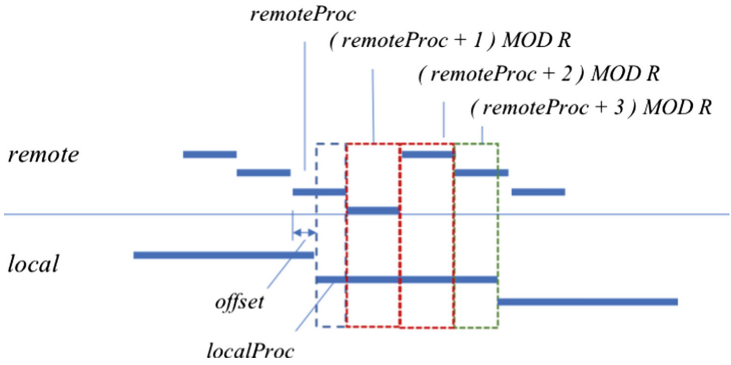

Fig. 2. Illustration of Adjacent shifting and periodic relations. *offset* is a periodic difference that will mean a local-remote comparison is valid for this local block when *offset* is greater than a threshold. The leftmost part of local data maps to processor *RemoteProc*. Adjacent data on the local processor can be known to then map to *RemoteProc+1* (and repeated for any further adjacent blocks); R represents the Remote grid dimension. (Color figure online)

When it holds, $remoteBlocksize - offset$ elements will be shared with processor *remote*. This is how ASPEN exploits the periodic nature of remote data to avoid looking at all remote blocks.

Properties of adjacent Sub-blocks. In the case that condition Eq. 3 holds, some number of elements are shared with processor *remote*. Instead of resetting knowledge with respect to the rest of the local block, ASPEN exploits the fact that if a set of global indices $g_l, \ldots, g_r$ of length less than $localBlockSize$ map to processor *remote*, and if some set of global indices $\{g_{r+1}, \ldots, g_{r+p}\}$ with $p + (r - l) \leq localxBlockSize$ then `localProc` will also share indices with processor $(remoteProc+1) \mod |C^d|$. Similarly, if several blocks of size `remoteBlockSize` fit into the `localBlock`, then each full block will map to the next processor in the remote grid. This approach is how ASPEN assigns contiguous local sub-blocks to adjacent processors in the remote grid (adjacency shifting). Hence the loop over remote processors in Algorithm 2 line 6 and Algorithm 3 line 11, does not appear in 4. The number of operations in Algorithm 4 is given by

$$\texttt{Ops}_{\text{ASPEN}} = D \cdot L \cdot \hat{N}^{local} \tag{4}$$

Comparing this to Eq. 2, we see a factor of $R \cdot \hat{N}^{remote}$ reduction in operations. The missing R term in particular will affect scalability since each grid will not require distinct calculations for each process element in the size of the remote grid. Theoretically then, we expect ASPEN to scale significantly better with larger Producer or Consumer grids involved in redistribution.

5 Results

The followings tests were run on Cray XC30 systems, each node featuring two Intel Xeon Haswell E5–2698 with 16 cores each (2.30 GHz). The benchmark was made of MPI applications computing independently the complete redistribution from one 2D grid of processes to another 2D grid, varying dimensions, shape and size of each grid. The data was a fixed size 2D square grid of ten thousand by ten thousand elements. Because this benchmark aimed at evaluating the redistribution performance, the computation were only executed on the indices and no actual communication of data occurred.

Each case of block-cyclic to block-cyclic distribution was run many times for all 4 methods: the naïve, the implementation of the algorithm presented in [1], the FALLS algorithm, the ScaLAPACK redistribution computation algorithm, and the ASPEN version[3]. The correctness of computed intersection was checked by comparing with the naïve approach results, and on later work in the *Universal Data Junction* library unit tests.

The main loop as show in Algorithm 1 was timed. In order to limit the impact of system related issues, all memory needed for the creation of intersection description sectors were pre-allocated before any measure of timing was taken. Nevertheless, outliers may appear because of cache misses.

The process grids were made of 2 to 32 processes per grid, and the block-cyclic sizes were one of 1024 by 1024, 256 by 256, 30 by 50 or 654 by 321. The objective was to highlight performance behaviour in regular-to-irregular redistributions, and the impact of partial blocks on the performance.

ASPEN showed to be very robust over disturbance induced by irregularity in structures. The main factor of influence over the execution time are the number of remote processes per rank. As shown in Fig. 3, while the number of blocks is scaled by a factor ≈ 8.5 and ≈ 5 in each dimension, timings scaled linearly for

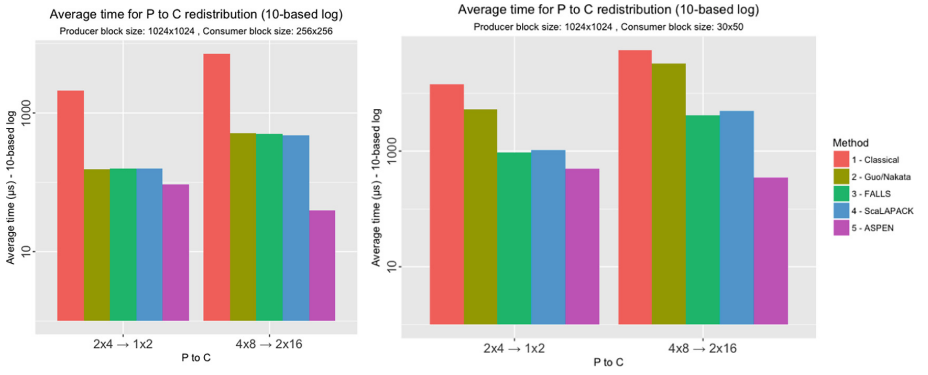

Fig. 3. Data redistributions for different blocksize and different grid sizes

[3] For all methods except classical, changing total data size does not affect the performance.

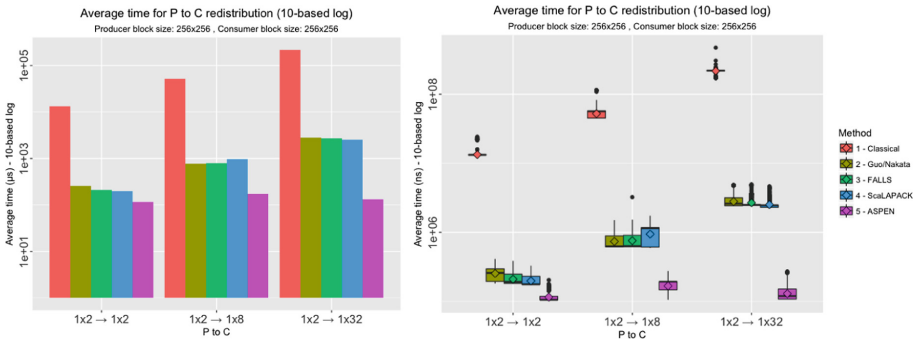

Fig. 4. Data redistributions for different blocksize and different grid sizes

ASPEN, which is not dependent on the remote number of blocks nor on remote grid dimension while for all other algorithms scale with the square (or worse) of grid length.

The results shown in Fig. 4 suggest a strong influence on performance by the number of remote processes. Since the R term can become significant even with small grids, we see the time begin to rise even for modest grid exchanges. With ASPEN, the R term is absent and this effect is limited. With large grid sizes, we expect to see this effect becoming critically significant.

6 Conclusion and Further Work

We have demonstrated that the ASPEN algorithm can generate redistributions more efficiently (both theoretically and in practice) when moving cyclic data across distinct processor grids. As there is a growing requirement to perform such redistributions across larger grids, the ASPEN algorithm is likely to be impactful. The total cost of moving data across jobs will depend on many factors, such as cost of generating the redistribution, cost of buffering data and message latencies. Subsequent work will study all of these factors by describing the ASPEN algorithmic framework integrated into the Universal Data Junction library, which will be used to send complex distributed data across production HPC jobs. We will investigate and implement ASPEN for redistribution of data using Gaussian grids and in complex workflow situations such as many-Producer, many-Consumer.

Acknowledgement. This work was partly funded by the EXPERTISE project (http://www.msca-expertise.eu/), which has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 721865.

References

1. Guo, M., Nakata, I.: A framework for efficient data redistribution on distributed memory multicomputers. J. Supercomput. **20**(3), 243–265 (2001). https://doi.org/10.1023/A:1011602732570
2. Hsu, C.H., Bai, S.W., Chung, Y.C., Yang, C.S.: A generalized basic-cycle calculation method for efficient array redistribution. IEEE Trans. Parallel Distrib. Syst. **11**(12), 1201–1216 (2000)
3. Hsu, C.H., Chung, Y.C., Yang, D.L., Dow, C.R.: A generalized processor mapping technique for array redistribution. IEEE Trans. Parallel Distrib. Syst. **12**(7), 743–757 (2001)
4. Loveman, D.B.: High performance fortran. IEEE Parallel Distrib. Technol. Syst. Appl. **1**(1), 25–42 (1993)
5. Petitet, A.P., Dongarra, J.J.: Algorithmic redistribution methods for block-cyclic decompositions. IEEE Trans. Parallel Distrib. Syst. **10**(12), 1201–1216 (1999)
6. Prylli, L., Tourancheau, B.: Fast runtime block cyclic data redistribution on multiprocessors. J. Parallel Distrib. Comput. **45**(1), 63–72 (1997)
7. Ramaswamy, S., Simons, B., Banerjee, P.: Optimizations for efficient array redistribution on distributed memory multicomputers. J. Parallel Distrib. Comput. **38**(2), 217–228 (1996)
8. Thakur, R., Choudhary, A., Fox, G.: Runtime array redistribution in HPF programs. In: Proceedings of the Scalable High-Performance Computing Conference, pp. 309–316. IEEE (1994)
9. Thakur, R., Choudhary, A., Ramanujam, J.: Efficient algorithms for array redistribution. IEEE Trans. Parallel Distrib. Syst. **7**(6), 587–594 (1996)