IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 21, 2021, accepted February 7, 2021, date of publication March 2, 2021, date of current version March 15, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3063628

Operators for Data Redistribution: Applications to the STL Library and RayTracing Algorithm

ANA MORETON-FERNANDEZ[1], YURI TORRES DE LA SIERRA[2], ARTURO GONZALEZ-ESCRIBANO[2], AND DIEGO R. LLANOS[2], (Senior Member, IEEE)

[1]Satlantis, University Science Park, 48940 Bilbao, Spain
[2]Departamento de Informática, Universidad de Valladolid, E.T.S. Ingeniería Informática, Campus Miguel Delibes s/n, 47011 Valladolid, Spain

Corresponding author: Arturo Gonzalez-Escribano (arturo@infor.uva.es)

This work was supported in part by the Spanish Ministerio de Economía, Industria y Competitividad and by the European Regional Development Fund (ERDF) program of the European Union, PCAS Project, under Grant TIN2017-88614-R, in part by the Junta de Castilla y Leon - FEDER Grants, PROPHET and PROPHET-2 Projects under Grant VA082P17 and Grant VA226P20, in part by the Salvador de Madariaga/Fulbright Scholar Grant PRX17/00674, and in part by the HPC-Europa3 supported by the European Commission H2020 Research & Innovation under Grant GA #730897.

ABSTRACT In distributed-memory systems, data redistributions are operations that change the ownership and location of a selected subset of a data structure at runtime. They allow the improvement of the performance of parallel algorithms which operate on changing or partial domains, aiming to create a balanced workload among the active processes. To manually redistribute data is a cumbersome and error-prone task. In this paper, we present a method based on four combinable operators to redistribute partial domains selected by the programmer at runtime in an efficient and simple way. They abstract to the programmer the data-redistribution implementation details, such as the new mapping, relocation, and communication of the selected data. We also present the application of the proposed operators to a RayTracing application and to a significant part of STL (C++ Standard Template Library). Our experimental results show that our approach automatically generates a good load balance, which leads to performance improvements for generic data-distribution policies. It does not introduce significant performance overheads compared with tailored data redistributions directly programmed using MPI (Message Passing Interface), while it greatly reduces the code development effort.

INDEX TERMS Parallel programming, distributed memory, data redistributions, RayTracing, STL library.

I. INTRODUCTION

The message-passing paradigm is a programming method for distributed-memory systems that leads to highly efficient programs in terms of performance. However, the programmer still has to deal with many decisions which are not related to the parallel algorithms, but to implementation issues, such as partition and locality vs. synchronization/communication costs, scheduling details, etc.

Many distributed-memory programs use static-scheduling approaches, where bulk pieces of data structures are mapped to a fixed number of processes, aiming to maintain locality during several computation and data communication stages. The program uses data-partition policies to map bulk parts of a given data structure to a set of distributed processes. These policies determine the ownership of each data element, and each process has a different part of the data structure allocated in its memory space.

Many parallel algorithms and programs need to dynamically change the subset of data elements on which they work across different computation stages or iterations. Examples include adaptive grid computations (see e.g. [1], [2]), image tracking algorithms [3], or simulations of N-body interactions with hierarchical or recursive partitions [4]. In these situations, *data redistribution* is needed. Given a data structure already distributed across processes with different memory spaces, a *data-redistribution operator* applies the same or another data-distribution policy to a selected subset of the data structure domain, relocating only the selected data elements. Data redistributions also allow the improvement of performance by creating a balanced workload among the active processes.

Data redistributions have been widely studied, mainly in the context of data-parallel languages and parallel compilers, such as in High Performance Fortran [5], [6]. The algorithms

The associate editor coordinating the review of this manuscript and approving it for publication was Asad Waqar Malik.

developed in this context are dependent on specific data partition policies, chosen at compile time, and they are in general designed to derive homogeneous sizes for dense data. However, there are still challenges to solve, such as how to generalize their application and combination at runtime in dynamically evolving applications.

The manual programming of data-redistribution techniques, or the adaptation of known ones to a specific problem, is a complicated and error-prone task. The programmer has to take into account many details, such as the partition policy used to initially distribute the data among the processes, the partition policy to be used after redistribution, the mechanism used to select the data to be relocated, the number of active processes, the number of array dimensions, the memory layouts, etc. The final code needs to be tailored to the specific problem, to the data-distribution policies and to the chosen data-redistribution algorithm. Different options for homogeneous or heterogeneous partitions should also be taken into account. Some decisions may even depend on run-time information, such as the execution platform features.

In this paper, we present a common framework to apply and combine four operators to efficiently redistribute selected data at runtime, making the data partition, relocation, and data movement transparent to the programmer. These operators are applied on an array already divided across distributed processes, and the result is another array containing all or part of the original array elements relocated across the available processes. The elements to be redistributed may be selected by bounding boxes of the array domains, or by masks to select sparse data. The data can be selectively mapped to one or more arrays, and the redistributions can be reversed to remap data to their original locations after a computation. These operators can be freely combined, even in a recursive algorithm. The implementation core uses only mapping information obtained at run-time, and does not depend on the particular distribution policies chosen. It supports both homogeneous and heterogeneous partitions with different arbitrary domain sizes per process.

We implement the four operators as an extension of an MPI-based library, named Hitmap [7]. Hitmap provides several features which simplify the data selection on an index domain, and the construction of reusable communication patterns. To show the applicability of our proposal, in this paper, we also present and test the RayTracing algorithm [8], [9], and a significant part of the C++ STL library [10], programmed using the proposed library implementation.

In summary, our main contributions are the following:

- A model for data redistributions at runtime on distributed-memory systems, including recursive or irregular-array-based algorithms, based on an abstract, generic method that applies and combines four operators that do not depend on the chosen partition policies.
- A model for data redistributions at runtime on distributed-memory systems, including recursive or irregular algorithms, based on an abstract and generic method that applies and combines four operators that do not depend on the chosen partition policies.
- The implementation of this framework and operators in an MPI-based library, also including a generic run-time core functionality and new supporting communication structures.
- The application of the operators in a real-world application (RayTracing), and in fifty two algorithms of the C++ STL library, also known as STL, which is the foundation for many other complex algorithms.

The results of an experimental study, using the STL routines and the RayTracing algorithm, show that our proposal achieves a good performance and scalability, while the programming effort is greatly reduced compared to programming directly with MPI.

The rest of the paper is organized as follows: Section 2 discusses some related work. Section 3 shows two motivating examples. Section 4 presents Hitmap, the library that we used to implement our proposal. Section 5 shows the implementation of the proposed operations using Hitmap. Section 6 shows an experimental evaluation. Finally, Section 7 presents the conclusions and future work.

II. RELATED WORK

In distributed-memory programming, there are two main design approaches: The first one is to statically divide the work in as many parts as cooperating processors; and the second one is to create a set of tasks that are dynamically assigned to the available workers. Our work focuses on the first approach, that is, static-scheduled distributed SPMD programs. Data-redistribution techniques for load balance have been studied in the context of data-parallel languages and parallel compilers, such as High Performance Fortran [5], [6]. However, the algorithms developed in this context are dependent on specific data partition policies, fixed at compile time, and in general designed to derive homogeneous sizes for dense arrays. Instead of relying on compile-time solutions, our work offers a data redistribution approach based on the use of four combinable operators to generate load balance when needed at runtime. In our work, the partition policies are functional parameters received by the operators and used at runtime to calculate the redistribution. Our system supports homogeneous and heterogeneous policies, and a mix of combinable techniques that support both dense and sparse selections.

Regarding task-based parallelism, many task-based parallel programming languages and models have arisen [11]–[14] for both shared- and distributed-memory systems, with the goal of abstracting to the programmer decisions about load-balancing, granularity, etc. All these programming models based on task parallelism generate performance penalties, especially in distributed-memory systems. For example, these penalties may be generated by task creation and destruction, the management of distributed queues, or the synchronization and load balancing mechanisms. In task-based data-flow

approaches, such as the distributed-memory extension for FastFlow [15], task construction implies a data partition and a dynamic task control that leads to a balanced load. However, this dynamic scheduling prevents the exploitation of affinities and data locality across tasks. Some of these problems can be alleviated by using runtime data redistribution instead, by transferring data ownership on demand without launching additional tasks.

Regarding existent tools that include some form of data redistribution, there are many works that provide external libraries which provide distributed data structures with a global view. One example is DASH [16], [17], a realization of the PGAS (partitioned global address space) model in the form of a C++ template library. In PGAS languages, access to remote elements is possible, but assuming a higher latency cost. PGAS languages in general benefit from explicit data-redistribution techniques to automatically group communications and reduce remote access latencies. Some PGAS languages provide a limited set of data-redistribution operations which are again based on specific homogeneous data-partition techniques. Our work generalizes this kind of data-redistribution operations, supporting both homogeneous and heterogeneous domains and targeting a wider range of applications.

Our work uses the STL library as an example to show the flexibility of our approach. The STL library has been one of the most parallelized and studied libraries in the literature. For example, the work in [18] presents an implementation for multicore architectures of the STL library using Cilk++. Works such as [19], [20] developed parallel versions of this library for shared- and distributed-memory systems using OpenMP and MPI respectively. These approaches only parallelize some selected functions of this library with independent and tailored solutions. Our work efficiently implements a wider range of STL routines with our generic framework, systematically applying our four operators. This also allows the transparent redistribution of data across consecutive calls to different STL routines.

III. MOTIVATING EXAMPLES

Data redistributions help to improve performance when the computation is performed in an unbalanced way according to an initial data distribution. Their use is convenient when the overhead produced by the communications needed to perform the data redistribution is expected to be less than the potential performance gain obtained by a better load balance. There are also algorithms based on recursive, divide & conquer, or similar paradigms, for example *QuickSort*, which always imply dynamic subselections of array structures. Thus, data redistributions are necessary when these algorithms are applied on previously-distributed arrays.

This section presents some motivating examples to show the complications of programming them in a plain message-passing model, and the advantages of having methods that systematically encapsulate low-level decisions, using partition policies as functional parameters. Our first

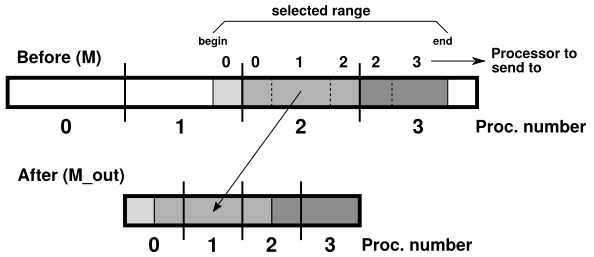


FIGURE 1. Motivating example 1: Data redistribution of a selected range of elements in a 1-dimensional array.

example implements a simple redistribution that works on 1-dimensional arrays. It uses classical block-partition policies that produce homogeneous sizes, and we assume sizes which are perfect multiples of the number of processes. The second example discusses some issues associated to arrays with two or more dimensions. Finally, we discuss the complications associated to different partition policies or different partition sizes on each process.

We focus the discussion on a message-passing or distributed SPMD (single-program-multiple-data) programming model. In this model, the processes have disjoint memory spaces, and they communicate data exclusively by explicit message-passing operations. Data from one process need to be communicated and copied into the local memory space of other processes that want to access them.

A. SIMPLE REDISTRIBUTION ON 1D-ARRAYS

Our first motivating approach is a data redistribution of a selected range of elements in a 1-dimensional array named M (see Figure 1). This situation arises in real applications, such as simulations of computational fluid dynamics. The idea is to recalculate only values in a significant neighborhood surrounding a particular element, for example to refine the results around a point of the array that experiments a sudden and significant change, without recalculating the values of the whole array. In our example, this neighborhood includes half of the elements of the array.

In this figure, we have an original array M partitioned across four processors. It is necessary to remap 50% of the elements evenly to the four processes. Different shades represent elements mapped to different processes. The figure aims to show the different possible situations that can arise. In our example, the data in the local subdomain of process 0 should not be redistributed; processes 1 and 3 should redistribute part of the data in their local subdomains; and process 2 should redistribute all of the data in its local subdomain. As it is shown in the figure, processes 2 and 3 should detect which part of their local subdomains correspond to subdomains mapped to different processes in the output structure. They need to split their data into two or three different Send() operations directed to different processes, or to themselves, in order to copy the data in the correct position of the new data structure. For example, processor 2 should move part of its data to the start of the local part of the new allocated structure, filling up the rest of it with data coming from

```
1  ** Algorithm interface
2  Inputs:
3    int size: Vector size
4    int pos: main position to update
5    int id: Local process identifier
6    int P: Number of processes
7    A<type> M[]: Distributed Vector
8
9  Outputs:
10   A<type> M_out[]: Distributed Vector
11
12 ** 1. Each process initializes its part
13    Range local;
14    local.begin = id * size/P;
15    local.end = local.begin + size/P -1;
16    for (i=0; i<local.end-local.begin+1; i++)
17      M[i] = init( i+local.begin, ... );
18
19 ** 2. Calculate the range of the neighbors
20 ** (in this example, 50% of elements of array M)
21    nElem = size/2;
22    first = (pos-nElem/2+1);
23    last = (pos+nElem/2);
24
25 ** 3. Calculate the redistribution and data comm.
26    Range local2, remote, remote2, s, r;
27    local2.begin = id*nElem/P+first;
28    local2.end = local2.begin-1+nElem/P;
29
30    for(pRem=0; pRem<P; pRem++)
31      remote.begin = pRem * size/P;
32      remote.end = remote.begin + size/P -1;
33      remote2.begin = pRem * nElem/P;
34      remote2.end = remote2.begin + nElem/P -1;
35      s.begin = max( local.begin, remote2.begin );
36      s.end = min( local.end, remote2.end );
37      if (s.end >= s.begin)
38        Send(M, s.begin-local.begin,
39                s.end-s.begin+1, pRem);
40      r.begin = max( local2.begin, remote.begin );
41      r.end = min( local2.end, remote.end );
42      if (r.end >= r.begin)
43        Recv(M_out, r.begin-local.begin,
44                    r.end-r.begin+1, pRem);
45
46 ** 4. Update the elements
47    for (i=0; i<local2.end-local2.begin+1; i++)
48      M_out[i] = Compute( i+local2.begin, ... );
```

FIGURE 2. Simplified distributed message-passing algorithm to perform the data redistribution depicted in Figure 1.

processor 3. All processes receive an even part of the selected data, but different sized subranges of its portion may come from different processes (see processors 0 and 2), implying different Recv() operations.

A simplified parallel algorithm of our first motivating example, using a message-passing approach, is presented in Figure 2. This algorithm starts with an array of an arbitrary base type *A<type>*, distributed in even parts across the available processes. Each processor initializes its local part in parallel. Then, only half of the elements, centered around a given position `pos` are selected to be updated. The output will be the distributed array *M_out* containing the updated elements. For simplicity, in this example, we assume that *size* and *size*/2 are divisible by *P*, and that the data were originally mapped assigning *size*/*P* elements with contiguous indexes to each process.

For a 1-dimensional array with contiguous blocks of indexes, all these Send() and Recv() communications can be programmed with a single MPI collective operation, `MPI_Alltoallv()`. Displacements and element counts,

```
1  ** ExtraInputs:
2    Map L1, L2: Mapping functions
3
4  ** 1. Each process initializes its part
5    Range local = L1(id,size,P);
6    for (i=0; i<local.end-local.begin+1; i++)
7      M[i]=init( i+local.begin, ... );
8
9  ** 2. Calculate the range of neighbors
10 ** (in this example, 50% of elements of array M)
11   nElem=size/2;
12   Range select;
13   select.begin = pos-nElem/2+1;
14   select.end = pos+nElem/2;
15
16 ** 3. Calculate the redistribution
17   M_out=ArrayRemapRange( M, select, L2 )
18
19 ** 4. Update the distributed "nElem" elements
20   Range local2 = L2(id,nElem,P);
21   for (i=0; i<local2.end-local2.begin+1; i++)
22     M_out[i] = Compute( i+local2.begin, ... )
```

FIGURE 3. The same algorithm of Figure 2, programmed using the proposed `ArrayRemapRange()` operator.

for both Send() and Recv() operations, may be different for each processor pair and should be calculated for each process by local and remote range intersections. For code clarity, to keep generality, and to avoid using extra data structures to store counts and displacements, the algorithm is presented using classical point-to-point Send/Recv operations.

The stages of the algorithm are marked in the figure with comments starting with two star symbols. In stage 1, each process initializes its local part of the input vector, which is defined by the index of the first element assigned to the local process, named *local.begin*, and the index of the last one, named *local.end*. The type *Range* represents a contiguous subdomain of indexes expressed as a pair of natural numbers $\langle begin, end \rangle$. In stage 2, the program calculates the size of the neighborhood and the range of elements that should be updated (*first*, *last*). Stage 3 redistributes the selected range of elements to be updated in a balanced output array, evenly distributed, among the available processes. Each process calculates which part of its original local range of indexes should be moved to another process, and which parts it should receive. Point-to-point communications are issued to do the data movements. Finally, in stage 4, each process updates the local part of its new data structure.

The data redistribution (stage 3) could be skipped, but it is desirable to balance the computational load. For example, if we try to redistribute one tenth of a vector of 16 000 elements with $P = 4$, but without balancing the data redistribution, the computation of the 1 600 elements will be performed by only one or at most two processes, not exploiting all the available nodes. Balancing the load, 400 of these elements will be assigned to each one of the 4 processors.

Figure 3 shows an alternative implementation that uses one of the operators proposed in this work, `ArrayRemapRange()`, which performs a transparent data redistribution operation, while the rest of the code uses two mapping functions of type $L(id, size, P)$ that returns the range of indexes to be mapped to process *id*. Both the operator and the mapping functions will be described in depth in the

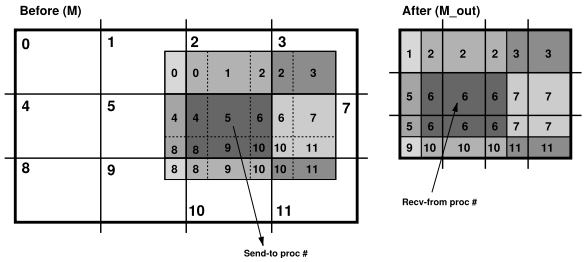


FIGURE 4. Motivating example 2: Data redistribution of a selected range of elements in a 2-dimensional array, mapped to 3 × 4 processors. Numbers inside the shaded boxes indicate to/from which processor the information is transferred.

following section. As it is shown in Figure 3, the programming effort is highly reduced with respect to the message-passing implementation shown in Figure 2. The programmer does not have to deal with all the necessary communications details to balance the computational load. Moreover, this operator works for generic array sizes, while the complexity of the original message-passing code will increase even more if we simply decide to consider sizes of arrays and selections which are not divisible by the number of processes.

B. REDISTRIBUTION ON 2D-ARRAYS

For this second example, we consider a problem in two dimensions. The processes are organized in a mesh, and the original array is mapped to them using non-overlapping rectangular shapes of indexes, as shown in Figure 4. The selected range of the input matrix elements to be redistributed is determined by an arbitrary 2-dimensional point indicating the left-top corner, and a 2-dimensional span (height and length). This scheme can be used, for example, in image processing algorithms that do object tracking. For simplicity, we again consider only sizes that always lead to exact divisions when the partitions are applied.

In this case, some processes should send 2-dimensional subranges of data to different processes. In most cases, the 2-dimensional subranges select a vertical stripe of rows, instead of full rows (all the subranges in the example present this situation). Thus, the selected bands of different rows are not contiguous in memory. This redistribution cannot be directly solved with a single collective communication primitive. The algorithm depicted in Figure 5 shows a simple approach in which one Send/Recv operation is calculated and issued for each piece of a row. See lines 47–50, and 57–60 of the code. However, this scheme leads to multiple communications between the same pair of processes. This may derive in poor communication efficiency, especially when there is only one or there are very few elements selected on each row. A slightly more complicated but better alternative is to traverse the selected domain or rows calculating the local-memory locations of each row band. The sender can pack the pieces of memory in a buffer before doing a single Send() operation. The receiver receives a single message and calculates the local-memory locations of each row band to unpack the portions. Libraries such as MPI provide methods

```
1  ** Common interface of both algorithms
2  Inputs:
3    int size[2]: Matrix dimensions
4    int corner[2]: left-top corner
5    int span[2]: Height and Length
6    int id[2]: Local process identifier
7    int P[2]: Processes mesh dimensions
8    A<type> M[][]: Distributed Matrix
9
10 Outputs:
11   A<type> M_out[][]: Distributed Matrix
12
13
14 ** 1. Each process initializes its part
15   Range local[2];
16   local[0].begin= id[0] * size[0]/P[0];
17   local[0].end= local[0].begin + size[0]/P[0]-1;
18   local[1].begin= id[1] * size[1]/P[1];
19   local[1].end= local[1].begin + size[1]/P[1]-1;
20   for (i=0; i<local[0].end-local[0].begin+1; i++)
21    for (j=0; j<local[1].end-local[1].begin+1; j++)
22      M[i][j] = init( ... );
23
24 ** 2. Calculate the redistribution
25   Range local2[2], remote[2], remote2[2], s[2], r[2];
26   local2[0].begin= id[0]*span[0]/P[0]+corner[0];
27   local2[0].end= local2[0].begin-1+span[0]/P[0];
28   local2[1].begin= id[1]*span[1]/P[1]+corner[1];
29   local2[1].end= local2[1].begin-1+span[1]/P[1];
30   for( i=0; i < P[0] ; i++ )
31    for( j=0; j < P[1] ; j++ )
32     remote[0].begin= i * size[0]/P[0];
33     remote[0].end= remote[0].begin + size/P[0]-1;
34     remote[1].begin= j * size[1]/P[1];
35     remote[1].end= remote[1].begin + size/P[1]-1;
36     remote2[0].begin= i * span[0]/P[0];
37     remote2[0].end= remote2[0].begin + span[0]/P[0]-1;
38     remote2[1].begin= j * span[1]/P[1];
39     remote2[1].end= remote2[1].begin + span[1]/P[1]-1;
40
41     s[0].begin=max(local[0].begin, remote2[0].begin);
42     s[0].end=min(local[0].end,remote2[0].end);
43     s[1].begin=max(local[1].begin, remote2[1].begin);
44     s[1].end=min(local[1].end,remote2[1].end);
45     if ( s[0].end >= s[0].begin AND
46          s[1].end >= s[1].begin )
47       for( k=s[0].begin-local[0].begin;
48            k<=s[0].end-local[0].begin; k++ )
49          Send( M[k][s[1].begin-local[1].begin],
50                s[1].end-s[1].begin+1, i*P[1]+j );
51     r[0].begin=max(local2[0].begin, remote[0].begin);
52     r[0].end=min(local2[0].end, remote[0].end );
53     r[1].begin=max(local2[1].begin, remote[1].begin);
54     r[1].end=min(local2[1].end, remote[1].end );
55     if ( r[0].end >= r[0].begin AND
56          r[1].end >= r[1].begin )
57       for( k=r[0].begin-local[0].begin;
58            k<=r[0].end-local[0].begin; k++ )
59          Recv( M_out[k][r[1].begin-local[1].begin],
60                r[1].end-r[1].begin+1, i*P[1]+j );
61
62 ** 3. Update the local elements
63   for (i=0; i<local2[0].end-local2[0].begin+1; i++)
64    for (j=0; j<local2[1].end-local2[1].begin+1; j++)
65      M_out[i][j] = Compute( ... );
```

FIGURE 5. Simplified distributed message-passing algorithm to perform the 2D data redistribution depicted in Figure 4.

to pack/unpack contiguous pieces of data elements in a buffer that can be directly communicated. Another approach is to calculate custom derived data-types for the communication across each pair of processes.

Nevertheless, there are situations where the rows can be contiguous in memory in the sender or the receiver. In Figure 4, consider for example the square pieces of data that are moved from process 2 to 1, or from process 6 to 5. In the sender, the row stripes are non-contiguous in memory, but in the receiver, the data completely fills the local part

```
1  ** Distributed algorithm using operators
2  ExtraInputs:
3    Map L1, L2: Mapping functions
4
5  ** 1. Each process initializes its part
6     Range local[2];
7     local2 = L1(p,size,P);
8     for( i=0; i<=local[0].end
9                   -local[0].begin+1; i++)
10      for( j=0; j<=local[0].end
11                    -local[0].begin+1; j++)
12         M[i][j] = init( ... );
13
14 ** 2. Select the range to be redistributed
15    Range select[2];
16    select[0] = <corner[0],span[0]>
17    select[1] = <corner[1],span[1]>
18
19 ** 3. Calculate the redistribution
20    M_out=ArrayRemapRange( M, select, L2 )
21
22 ** 4. Update the elements
23    Range local2[2];
24    local2 = L2(p,span,P);
25    for( i=0; i<=local2[0].end
26                  -local2[0].begin+1; i++)
27      for( j=0; j<=local2[0].end
28                    -local2[0].begin+1; j++)
29        M_out[i][j] = Compute( ... );
```

FIGURE 6. The same algorithm of Figure 5, programmed using the proposed ArrayRemapRange() operator.

of the target data structure. Thus, these row pieces will be contiguous in the receiver local memory. Traversing the rows doing a partial unpacking for each row is less efficient than a single contiguous unmarshalled operation. The code should take into account these situations by adding conditionals with different behaviors. The number of situations in which data are or are not contiguous across different dimensions are countless.

More complicated situations that lead to more complex codes also appear when the partition policies are different, or when one of them is non-homogeneous. Different policies are used when dealing with different computational power on heterogeneous systems, or in cases where part of the data needs more computational operations than the rest. All these changes need to be reflected in the code, and in the marshaling/unmarshalling procedures. The amount of possibilities is huge, and the codes which are tailored and highly-efficient for one situation are not the most appropriate for others.

Figure 6 shows an alternative implementation that again uses our `ArrayRemapRange()` operator, which performs a transparent data redistribution operation, and the two mapping functions of type $L(id, size, P)$, which returns the range of indexes to be mapped to process id. It is important to highlight (a) that this implementation is very similar to the 1-dimensional case depicted in Figure 3; (b) that the programming effort has been even more reduced with respect to the message-passing version; and (c) that our operators work again with generic array sizes and ranges.

IV. HITMAP IN A NUTSHELL

The new operators have been implemented using Hitmap [7], [21], an MPI-based library with many features

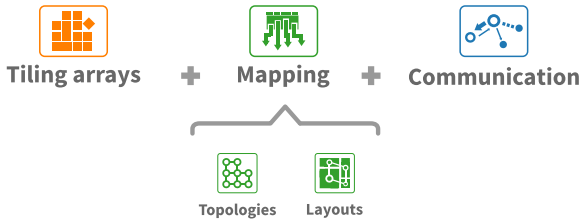


FIGURE 7. Hitmap library functionalities [22].

that are particularly appropriate for our proposal. In this section, we briefly describe the Hitmap features that allow us to implement the operators described in the previous section.

Hitmap is a library designed for hierarchical tiling and mapping of dense and sparse arrays. Hitmap is based on a distributed SPMD programming model, using abstractions to declare data structures with a global view. It automates the partition, mapping, and communication of tile and tile hierarchies, while still delivering good performance.

In Hitmap, data-layout and load-balancing techniques are independent modules that belong to a plug-in system. The techniques are invoked from the code and applied at runtime when needed, using internal information of the target-system topology to distribute the data [22]. The programmer does not need to reason in terms of the number of physical processors. Instead, it uses highly abstract communication patterns for the distributed tiles at any grain level. Thus, coding and debugging operations with entire data structures are easy.

The Hitmap library supports functionalities to: (1) Generate a virtual topology structure; (2) map the data grids to the different processors with chosen load-balancoing techniques; (3) automatically determine inactive processors at any stage of the computation; (4) identify the neighbor processors to use in communications; and (5) build communication patterns to be reused across algorithm iterations. These functionalities are organized in three different categories: Tiling, Mapping, and Communications. These categories are represented in Figure 7.

Hitmap defines several concepts to write parallel programs. A *shape* represents the domain of the data used in the program while a *tile* is the entity that keeps the actual elements. The tiles are created using the shapes. In Hitmap, a *topology* describes the structure of the available processes. A *layout* is the entity that distributes the shapes onto a topology, dividing the domain for each element of the topology. The concept of *communication* represents data transmission between two or more processes. Finally, a *pattern* groups together several communications.

We describe now in more detail the Hitmap objects and their usage. The proposed operators are built on top of these objects. An example of a simple Hitmap program is shown in Figure 8. A *HitShape* object represents a subspace of domain indexes defined as an n-dimensional rectangular parallelotope. The limits on each dimension are represented with a *HitSig* object, containing the range limits (begin, end),

```
// DECLARE USAGE OF TILES WITH int ELEMENTS
hit_tileNewType(int);
...

int main( int argc, char *argv[] ) {
  hit_comInit( &argc, &argv );

  // CREATE A BLOCK DISTRIBUTION OF 1D INDEX SPACE
  HitShape globalSh = hit_shape( 1, hit_sigStd(1,SIZE) );
  HitTopology topoPlain = hit_topology(plug_topPlain);
  HitLayout layBlck =
     hit_layout(plug_layDimBlocks, topoPlain, globalSh);

  // CREATE AND ALLOCATE LOCAL PART OF A DISTRIBUTED ARRAY
  HitTile_int my_arr;
  hit_tileDomainShape(&my_arr, int, hit_layShape(layBlck));
  hit_tileAlloc(&my_arr);

  // INITIALIZE ELEMENTS OF MY PART
  hit_tileForDimDomain( my_arr, 0, i )
     hit_tileElem(my_arr,0,i) = generateValue();

  // COMMUNICATION: CIRCULAR SHIFT OF BLOCKS
  HitComm shift = hit_comShiftDim(layBlck,0,+1,my_arr,int);
  hit_commDo( shift );

  // USE THE RECEIVED DATA
  hit_tileForDimDomain( my_arr, 0, i )
     doSomethingUseful( hit_tileElem(my_arr,0,i) );

  hit_comFinalize();
}
```

FIGURE 8. Example of a simple Hitmap program that creates an array distributed by contiguous blocks, and shifts data blocks once across processes. The local block is sent to the next process and the remote block from the previous process substitutes the local one.

and an optional stride. Multidimensional indexes, ranges, and domains can be expressed using these abstractions.

A *HitTile* maps actual data elements to the index subspace defined by a shape. New internally allocated tiles use a contiguous block of memory to store data. Subsequent hierarchical subselections of a tile reference data of the ancestor tile, using the signature information to locate and access data efficiently. Tile subselections may also be allocated to their own memory space.

HitTopology and *HitLayout* abstract classes are interfaces for two different plug-in systems. These plug-in modules are selected by the name of their constructor method. Topology plug-ins implement simple functionalities to arrange physical processors in virtual topologies, with their own rules to build neighborhood relationships. Layout plug-ins implement methods to distribute a domain shape across the processors of a virtual topology. The resulting Layout object contains information about the local part of the domain, neighborhood relationships, and methods to locate the remote subdomains. HitLayout objects can also be used to represent the T functions used in the *ArrayDivide* and *ArrayMerge* operators.

Finally, *HitComm* and *HitPattern* objects represent information to synchronize or communicate data tiles among processes. The class provides multiple constructor methods to build different communication schemes based on point-to-point or collective communications. The library is built on top of the MPI communication library for portability across different architectures. Hitmap internally exploits MPI techniques that increase performance, such as MPI derived data-types and asynchronous communications.

V. IMPLEMENTATION OF THE OPERATORS USING HITMAP

In this section, we present the extensions developed in Hitmap to support the proposed framework for the four operators.

A. SUPPORTING DATA REDISTRIBUTIONS AT HITMAP RUNTIME LEVEL

We need to provide Hitmap with the necessary features to develop the four operators. We introduce a new function named *localRange(Tile, Shape)*. It receives a distributed tile structure, and a selection range in global coordinates. It returns a *hit_shape* object representing the part of the input range that is allocated in the local process. For example, in Figure 9, the function *localRange(M, [2:10])* returns, for the process 0, the shape that selects its last two local elements, and for process 2, the shape that contains its first three local elements.

We also develop in Hitmap a generic redistribution communication pattern constructor (*hit_patRedistribute()*), which is the core of the redistribution operators. It receives two already distributed arrays (which in Hitmap contain a reference to their respective layout functions L and L', originally used to distribute their domains). The constructor simply traverses the process-identifiers space with two loops. In the first loop, we compute the intersections of the result of applying L at the local process, with the result of applying L' at each remote process, to calculate the data indexes to be sent. In the same way, the second loop computes the inverse intersections, applying L' at the local process, and L at each remote process, to calculate the data to be received. The representation of multi-dimensional domains is always done in terms of *HitShapes*. The loops traverse the process identifiers cyclically, starting at the local identifier plus 1; $(myRank + 1)\ mod\ P$. This generates a skewed communication scheme, that helps to reduce communication saturation bottlenecks in specific processes. The library internally analyzes the dimensions and memory layout registered in the meta-information of the tiles, and builds hierarchical derived data-types to allow the MPI implementation to perform marshalling/unmarshalling operations with the minimum number of contiguous data blocks.

We implemented the four operators as C macro-functions in the Hitmap library. For the rest of the section, we represent ranges by *HitShape* objects, independently of the number of dimensions, and data structures by *HitTile* objects.

B. ArrayRemapRange()

The function prototype of this operator is:

```
ArrayRemapRange(tileIn, tileOut,
  rangeIn, rangeOut, baseType);
```

where *tileIn* is the data structure to be redistributed; *tileOut* is the output data structure with the selected data; *rangeIn* is the range of data to be selected in *tileIn*; *rangeOut* is the range that will contain the data in *tileOut* after the redistribution; and *baseType* is the name of the native or structured type.

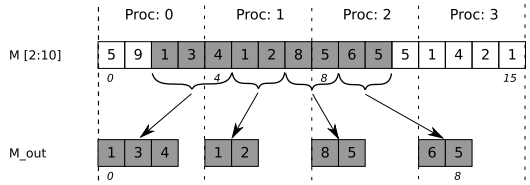


FIGURE 9. Data redistribution performed by the *ArrayRemapRange* operator. In this case the call to the operator is $M\_out = ArrayRemapRange(M, \langle 2, 10\rangle, L)$.

This operator is used as the baseline in the development of the rest of the operators, so we also describe its internal code, shown in Figure 10. It first declares the necessary variables. In the cases where the input array has been modified by the programmer, including in the parts of the local domain overlapped parts with other remote domains (such as halos in Stencil computations), the operator reproduces the same halos in the output array, by comparing the allocated domain with the domain assigned by the mapping function (step 2). The data moving to the halos is not included in the operator, as it depends on the program stage in which the operator is invoked.

After that, the program calculates and selects, from the input array, the part of the selection range which is in the local process (step 3). The last step creates and executes the pattern containing the necessary communications. The mapping functions used in the redistribution are those used to create the data structures. They are represented by *HitLayout* objects, and are kept as meta-data in its own *HitTile* data structure.

C. ArrayRemapMask()

The function prototype of this operator is:

```
ArrayRemapMask(tileIn, tileOut, maskIn,
                baseType);
```

where *tileIn* is the data structure to be redistributed; *tileOut* is the output data structure with the selected data; *maskIn* is the mask with the indexes of the data structure to be selected in *tileIn*; and *baseType* is the base type of the data structure.

Internally, this function code selects, for each process, the data elements whose mask value is 1 in the local process. In this case, to select the data, we generate a loop that traverses the local domain, analyzing the mask to identify the selected elements. It copies contiguously the selected data elements in an auxiliary array with contiguous memory. After that, a communication reduction is performed to share the information about the number of elements to be copied for each process. Finally, the redistribution is performed using ranges, as in the first operator.

D. ArrayDivide()

The function prototype of this operator is:

```
ArrayDivide(grouping, tileIn, tileOut,
            maskIn, baseType);
```

```
/* AUXILIAR MACROS */
#define NORMALIZE(TOK) NORMALIZE_ ## TOK
#define NORMALIZE_int HIT_INT
#define NORMALIZE_float HIT_FLOAT
#define NORMALIZE_double HIT_DOUBLE

/* ArrayRemapRange OPERATOR */
#define ArrayRemapRange(tileIn, tileOut,
 rangeIn, rangeOut, baseType){ \
 /* STEP 1: DECLARE AUXILIAR VARIABLES */ \
 HitTile_##baseType _TT_In, _TT_Out; \
 HitShape _TT_shape_Out,  _TT_shape_In; \
 /* STEP 2: EXTENT RANGES WITH NECESSARY HALOS */ \
 int i; \
 for(i=0;i< hit_shapeSigDims(rangeIn);i++) { \
  int diff = hit_tileDimCard(tileOut,i) - \
   hit_shapeSigCard(hit_layShape(tileOut.Layout),i);\
  hit_shapeDimExpand (range1, i, HIT_SHAPE_END, diff);\
  hit_shapeDimExpand (range2, i, HIT_SHAPE_END, diff);\
 } \
 /* STEP 3: SELECT THE DATA CORRESP. TO EACH PROCESS */\
 _TT_shape_In = localRange(tileIn, rangeIn); \
 _TT_shape_Out = localRange(tileOut, rangeOut); \
 hit_tileSelectArrayCoords (&_TT_In, &tileIn, \
  _TT_shape_In); \
 hit_tileSelectArrayCoords (&_TT_Out, &tileOut, \
  _TT_shape_Out); \
 /* STEP 4: PERFORM THE DATA REDISTRIBUTION */ \
 HitPattern redis= hit_pattern (HIT_PAT_UNORDERED); \
 redis = hit_patRedistribute( (tileIn).Layout, \
  (tileOut).Layout, &_TT_In, &_TT_Out, rangeIn, \
  rangeOut, NORMALIZE(baseType) ); }
```

FIGURE 10. Internal code of the *ArrayRemapRange* operator along with some auxiliary macro functions.

where *grouping* is a HitLayout object representing the T function (recall the operator definition), which contains the information of the number of natural values in the mask; *tileIn* is the data structure to be redistributed; *tileOut* is the output data structure with the selected data; *maskIn* is the mask with the indexes of the data structure to be selected in *tileIn*; and *baseType* is the base type of the data structure.

The internal code creates a collection of arrays, where each array stores the elements that belong to the same array, using the same methodology as the *ArrayRemapMask* operator. However, this operator also stores the global index domain of the original array in the meta-data of the data structures. This last feature enables the use of the *ArrayMerge* operator.

E. ArrayMerge()

The function prototype of this operator is:

```
ArrayMerge(grouping, tilesIn, tileOut,
            baseType);
```

where *grouping* is the HitLayout object representing the T function, that contains the information of how to concatenate the collection of input arrays; *tilesIn* is the collection of arrays to be concatenated; *tileOut* is the output array; and *baseType* is the base type of the data structure. The internal code of this operator calls the Hitmap redistribution function for each input array, relocating the data in their corresponding ranges of the single output array. The ranges are calculated using the meta-data with the information about the index space on the original array, which was set by the *ArrayDivide* operator.

TABLE 1. Summary of the implemented STL routines for one dimensional numeric arrays, for distributed-memory systems, using the new four operators.

Algorithm Class	Routines
Embarrassingly Parallel	all_of, any_of, none_of, copy, copy_if, copy_n, count, count_if, fill, fill_n, for_each, generate, generate_n, replace_if, transform, swap_ranges
Find	find, find_if, find_end, find_first_of, adjacent_find, mismatch, equal
Search	search, search_n, lower_bound, upper_bound
Numerical Algorithms	min_element, max_element, minmax_element, accumulate, adjacent_difference, inner_product, rotate, rotate_copy
Partition	is_partitioned, partition, partition_copy, partition_point
Merge	merge, inplace_merge
Sort	quickSort, is_sorted, is_sorted_until, partial_sort_copy,
Complex Set Operations	remove_if, remove_copy_if, set_difference, set_symmetric_difference, set_intersection, set_union, unique, unique_copy

VI. EXPERIMENTAL EVALUATION

We conducted several experimental studies to verify the efficiency of the new codes that use the proposed operators, in terms of runtime execution and development effort.

A. EXPERIMENTAL PLATFORM AND SETUP

For the performance studies, we executed the experiments in CETA. This is a hybrid cluster that belongs to CIEMAT and the Spanish government. The cluster nodes are connected by Infiniband technology, and each one has two Intel Xeon 5520 CPUs at 2.27 GHz, with 4 cores each. Using 16 nodes of the cluster, we exploit up to 128 computational units. We have compiled the codes with the GCC v4.8.3 compiler, using the optimization flag *-O3*. We used *mpich3* v3.1.3 as MPI implementation. We executed all the experiments ten times, registering the average total execution times. We used a static mapping policy, associating one MPI process to each processing element. For all the routines and examples tested, we always used a mapping policy of contiguous balanced blocks, because of the homogeneous execution platform.

For the development effort comparison, we used four classical development effort metrics: COCOMO lines of code, the number of tokens, McCabe's cyclomatic complexity [23], and the Halstead development effort [24]. The number of tokens detected by the programming-language parser measures the code volume of C/C++ programs better than the number of code lines. McCabe's cyclomatic complexity is a quantitative measure of the number of linearly independent paths through a program's source code. Finally, Halstead's development-effort metric is also a quantitative measure based on the number of operators and operands in the source code. They are related to the mental activity needed by a programmer to develop the code, and to the amount of test cases needed to check the program correctness. Low cyclomatic complexity and Halstead's development effort indicate codes which are simpler to develop and debug. These metrics are typically used in the assessment of software design complexity.

B. APPLYING THE OPERATORS: CASE STUDIES

In order to validate our approach, we implement the following case studies using the operators:

- **STL Benchmarks**: The STL Library is a well-known supporting tool for developers [25] that includes many useful algorithms. During the last few years, many works have presented parallel versions of this library [19], [20], [26], as well as new parallel programming models that support the development or use of this library in parallel [18].
  We implement the routines summarized in Tab. 1 using the operators. However, in this paper, we only show the results of four specific routines. They cover the different kinds of data redistribution that, together, can support all the other implemented STL algorithms. The representative results can be extrapolated to the other STL routines tested. The routines we choose are:
  - **for_each**: This example updates a range of elements of an array. The program performs N operations for each element on the selected range of the array, N being the array size. We use the *ArrayRemapRange* operator. This kind of data redistribution appears in most of the algorithms in the STL.
  - **find**: This routine searches the first position in a range of an array that fits with a specified condition. It applies a function to each element before a global reduction. This example also uses a data redistribution that remaps a selected range of a distributed array. It is another example of the use of the *ArrayRemapRange* operator, but continued with a conditional and a reduction communication.
  - **unique_copy**: This example copies the elements of a range of an array to a second array, skipping the consecutive duplicates. The size of the second data structure contains the elements copied from the first data structure. We use the *ArrayRemapMask* operator to only select data elements that have to be copied. Each process fills its assigned part of the mask prior to the operator invocation, comparing each element with its neighbor. In the experimental study, we initialize 20% of the contiguous data elements with the same value. Thus, all these elements except the first one are eliminated in the output array.

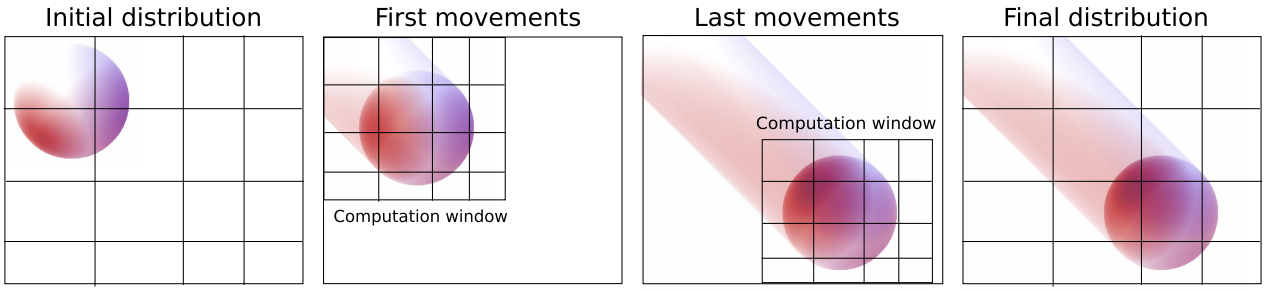


FIGURE 11. Consecutive applications of the RayTracing algorithm on a moving sphere.

  - **quickSort**: This example sorts the elements of an array. To implement this algorithm in a distributed-memory system, it is necessary to perform a sequence of recursive data redistributions. We use the *ArrayDivide* and *ArrayMerge* operators to divide and merge the pivoted arrays at each recursion level.

- **RayTracing Algorithm**: RayTracing is a technique to develop an image from a 3D scene by trajectories of light rays through pixels in a view plane [8], [9]. This technique produces a very high degree of photo-realism. However, its high computational cost makes its use prohibitive in real-time applications. RayTracing is an embarrassingly-parallel application, where a scene-dependent workload partition could achieve huge improvements on performance.
  We use a sequence of applications of the RayTracing algorithm on an object in movement. Figure 11 shows the behavior of the study case. In the figure, the scene is a sphere, where a shadow of this sphere is printed in each step. In the first stage (*Initial distribution*), the image is distributed among the active processes (16 in the figure) using a 2D irregular partition policy by blocks based on the computational power of the execution machines. For this first distribution, there is no knowledge of the scene, so the whole image is distributed. As we observe in the example image, there are only four processes initially involved in computing the RayTracing on the sphere. The computation window is detected at each moment, and a data redistribution is performed in order to balance the workload, as we can see in the First and Last movements of Figure 11. In order to avoid a data redistribution at each movement, the computation window is bigger than the sphere. Thus, the data redistribution is only performed when the moving object arrives close to the boundaries.
  For this application, we develop a code based on the sequential code of [27]. Our case study scene has a sphere with a size of one eighth of the image size, and we execute a number of movements equal to a tenth of the image size in a diagonal direction, as in the image. As for data redistributions, in our implementation, we use the *ArrayRemapRange* operator, selecting a 2D domain. Using the current scene and parameters, the program needs to perform 6 redistributions. In this experimental study, we use a mapping policy of contiguous balanced blocks, because of the homogeneous execution platform.

C. IMPACT OF REDISTRIBUTING WORKLOAD ON PERFORMANCE

In this section, we present a performance study of a distributed-memory system to show the positive effect of the data redistributions on several applications, in a similar way to [9], [28].

First, we test the *for_each* routine on an array of 10^6 elements. It is an example that easily allows the exploration of the effects on performance related to the variation of the amount of load redistributed, and its location on the original domain. We design the experiments to study the impact of two parameters in the data-redistribution operations of this application:

1) The amount of data selected from the original array where the routine is applied. We perform the experimentation selecting 20, 50 and 80% of the data in the whole vector.
2) The place in the original array where the range of data is selected. Data redistributions can have a different performance in function of the number of processes actually involved in the communications. Thus, we perform the experimentation selecting the data in three ways: (1) Selecting a range of data chosen from the beginning of the array (*Left*), (2) selecting the data chosen at the end of the array (*Right*), and (3) selecting a range of data chosen, with the center of the selected range at the middle point of the whole array (*Center*).

Figure 12 shows the performance obtained for the different versions and parameters of the *for_each* routine in CETA. We have tested four kinds of code, using different range selections. The first type is a Hitmap implementation that does not include data redistributions (named *Hitmap No Redist* in the plots). In this version, each process works with the originally mapped data in the selected range (if any). The second is a Hitmap implementation that redistributes the data in the selected range across all the processes to balance the computational load using the operators (named *Hitmap Redist*

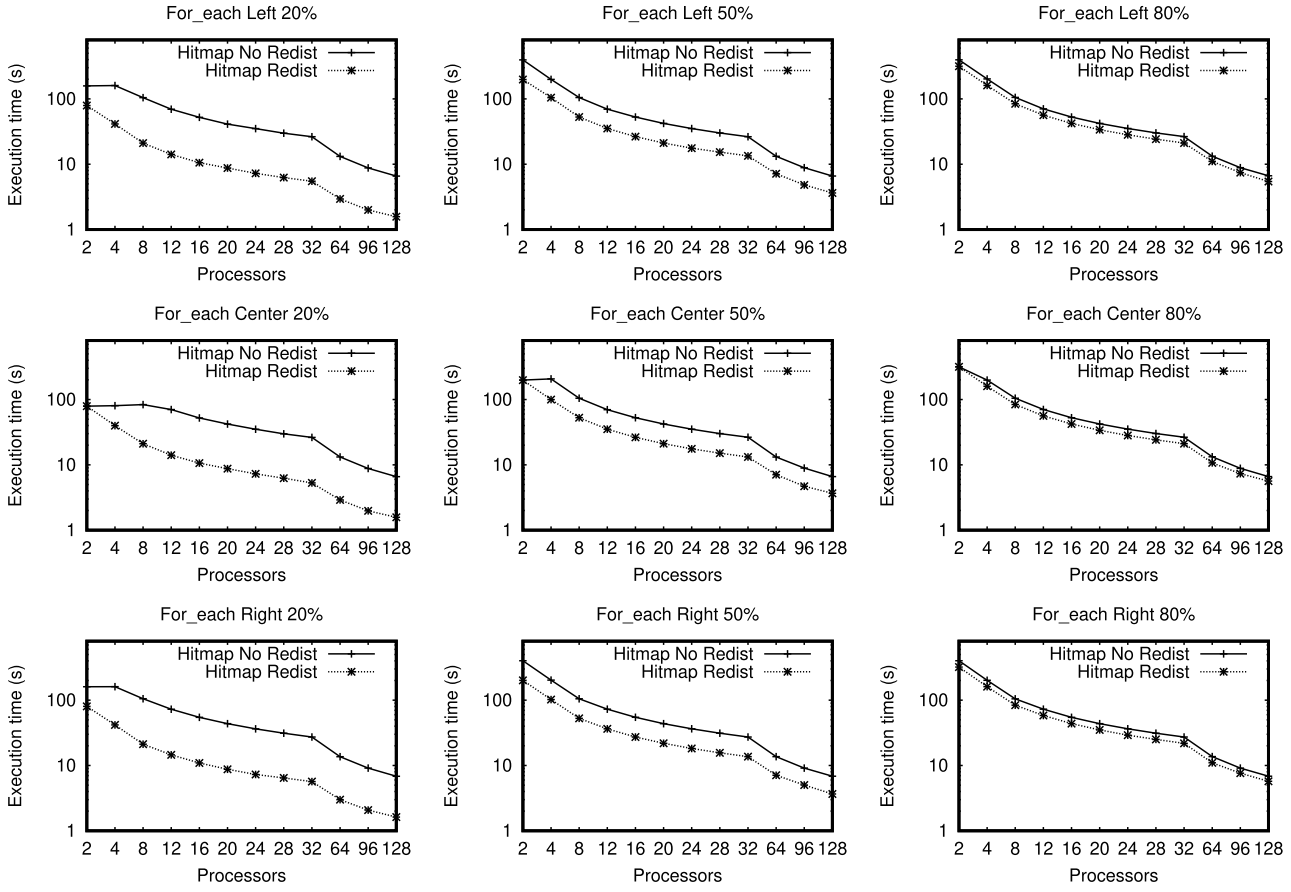


FIGURE 12. Performance scalability results (in seconds) for the *for_each* algorithm in CETA, the distributed-memory system (logarithmic scale). *size* = 1 000 000.

in the plots). The comparison of these two versions shows the performance improvement of using data redistributions. Two pure-MPI versions, one without redistributions and the other with manually-implemented redistributions, have also been tested. The latter contains a tailored redistribution code specifically designed for 1-dimensional arrays and for the block data-partition policy. In both cases, the pure-MPI versions behave in a similar way to their Hitmap counterparts, with an overhead of less than 2% in both cases. All implementations use the same sequential functions and semantic structure, so we only see the performance penalty or gain that comes from using our data-redistribution operators, which is the focus of our study.

We can also observe the impact of the load balance obtained with the data redistributions. When the data selection is 80% of the whole array, redistributing the data does not have a big performance impact. However, when the amount of data selection is low (20% or 50%), the performance obtained by the load-balanced codes is significantly better than in the codes which do not use it.

Our operators redistribute the data that need computation, avoiding idle processors and creating load balance. They alleviate the extra communication costs in most of the cases, without adding significant code development effort, as will be shown in the next section.

D. USING THE STL LIBRARY TO ANALYZE THE FOUR OPERATORS

The previous section shows the advantages of the data redistributions. Now we focus on analyzing the behavior of the different kinds of operators presented in this paper, by comparing our proposal with MPI in terms of both execution time and development effort, for the chosen STL routines, covering the different kinds of data redistributions.

In Table 2 we show several development effort measures, comparing the described STL routines, coded in Hitmap with the new operators (*Hitmap Redist*), or coded directly in MPI (*MPI Redist*). For the QuickSort, we use the implementation presented in [29], [30] as the baseline. We can see that using the operators, the measures for the chosen metrics are highly reduced. This indicates a clear simplification of the programmability for the developer.

Table 3 shows the execution times obtained for the STL study cases (the *for_each* routine has already been studied in the previous section). We show the results when the routines are executed with 128 MPI processes in CETA. We observe

TABLE 2. Measures of development effort for the STL study cases, comparing our proposal with MPI.

	For each		Find	
Measure	**MPI Redist**	**Hitmap Redist**	**MPI Redist**	**Hitmap Redist**
N. Lines	135	72	151	68
N. Tokens	1407	495	2030	495
Mccabe C. C.	19	3	23	3
Halstead D. E.	517,153	110,692	786,800	105,709

	Unique copy		QuickSort	
Measure	**MPI Redist**	**Hitmap Redist**	**MPI Redist**	**Hitmap Redist**
N. Lines	266	117	237	156
N. Tokens	2334	1000	1686	1225
Mccabe C. C.	42	4	47	25
Halstead D. E.	1,763,360	306,887	1,411,782	575,337

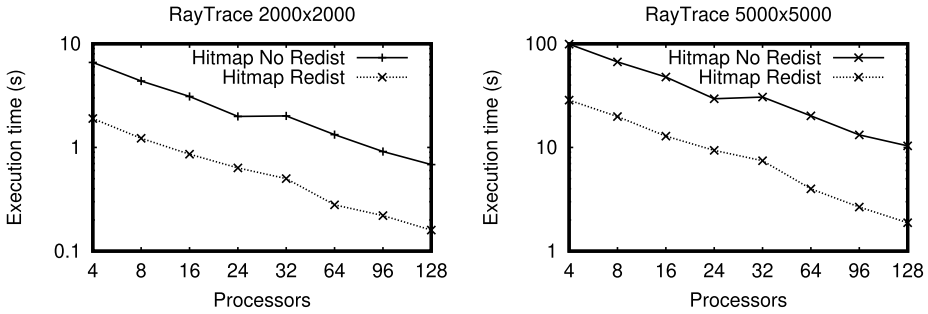


FIGURE 13. Performance scalability results (in seconds) for the *RayTracing* algorithm in CETA with different image sizes (logarithmic scale).

TABLE 3. Measures (in milliseconds) of performance for the STL study cases comparing our proposal with MPI using 128 MPI processes (the *for_each* routine results were discussed in Sect. VI-C).

Routine	Size	MPI Redist	Hitmap Redist
Find	10^5	66.253	66.983
Unique copy	10^5	51.015	51.705
QuickSort	10^5	1.023	1.098

TABLE 4. Measures of development effort for the RayTracing algorithm, comparing our proposal with MPI.

Routine	MPI No Redist	Hitmap No Redist	Hitmap Redist
N. Lines	322	255	302
N. Tokens	2 967	2 245	3 037
Mccabe C. C.	32	14	15
Halstead D. E.	3 150 194	1 912 643	3 224 414

that the performance obtained by our approach is similar to MPI. These examples have a very low computational load. Thus, the main advantage of redistributing the data structures comes from keeping them distributed among several nodes, instead of reducing the computation time.

E. EVALUATING THE USE OF THE PROPOSAL ON A REAL-WORLD APPLICATION: RayTracing ALGORITHM

In this study, we focus on the impact of the data redistribution, in terms of both performance and the development effort using the proposed operators. We used a 2D real-world application, the RayTracing algorithm.

In this case we have tested three codes. The first is an original MPI reference code (that we call *MPI No Redist*). There is no selection of interest area or redistribution. Each process traverses, on each iteration, the whole assigned subdomain. The second is a Hitmap code, also with no selection of interest area or data redistributions (*Hitmap No Redist*). The third is a Hitmap code, where data redistributions are applied using the approach presented in this paper (*Hitmap Redist*). The comparison of these codes shows the little extra development effort needed to introduce data redistributions which implement a sliding window technique using our proposal, and the performance advantage of the whole transformation. We do not include an *MPI Redist* version in this case study. To manually develop with MPI, a two-dimensional data redistribution using a sliding window is a complex task that involves an important development effort, and we believe it would not return new insights.

Table 4 shows the development effort measures of the studied codes. We observe that, using the operators, the development effort needed to transform the code by implementing a data redistribution on a sliding window, is slightly increased when compared to the *Hitmap No Redist* code (with no sliding window and data redistributions), but lower than the *MPI No Redist* reference code (again with no sliding window and data redistributions).

Regarding performance (Figure 13), the *Hitmap Redist* code that uses the new operators greatly improves the *Hitmap No Redist* version, also showing a good scalability. The *Hitmap No Redist* shows a similar performance to the *MPI No Redist* code (not shown in Figure 13), with less than a 2% overhead.

While the complexity introduced by the sliding window mechanism and data redistribution using *Hitmap Redist* is still comparable to developing the *MPI No Redist* code, the performance obtained is much better. Our operators abstract to the programmer all the data-redistribution implementation issues related to the data partition, data ownership, and data communication. Moreover, despite the potential overhead derived from dealing with distributed data in an abstract way, our approach produces a good scalability, and a negligible penalty performance compared with the MPI implementations.

VII. CONCLUSION AND FUTURE WORK

This paper presents a framework with four array data redistribution operators to efficiently implement distributed-memory algorithms, making the data partition, relocation and data movement transparent to the programmer. Our proposed solution based on these operators provides programming abstractions to manage data redistributions. We also present the application of the operators in a real-world application (RayTracing), and in many algorithms of the C++ STL library.

With our proposal, the programmer does not need to deal with data-redistribution implementation issues that are not related with the algorithms, but are key in terms of performance. Experimental results show that our proposal achieves the same performance as optimized MPI codes with tailored data redistribution solutions hard-wired into the code, while the programming effort is greatly reduced. The software used in this study, including the Operators library and the case-study programs, are available at https://trasgo.infor.uva.es/sdm_downloads/operadores/.

This proposal introduces a powerful tool to exploit data redistribution on a higher abstraction level. Future work includes the use of these operators and framework as a lower-level layer for parallel-code generating tools or parallelizing compilers. These tools could derive from sequential or high-level parallel code the most appropriate data partition policies and data selection details to efficiently map computations to complex platforms, as well as providing the foundation to use redistribution operators in general to program the communication structures of whole complex applications.

ACKNOWLEDGMENT

The authors have used the computing facilities of Extremadura Research Centre for Advanced Technologies (CETA-CIEMAT), funded by the European Regional Development Fund (ERDF). CETA-CIEMAT belongs to CIEMAT and the Government of Spain.

REFERENCES

[1] F. Nobile, L. Tamellini, F. Tesei, and R. Tempone, "An adaptive sparse grid algorithm for elliptic PDEs with lognormal diffusion coefficient," in *Sparse Grids and Applications—Stuttgart 2014*. Cham, Switzerland: Springer, 2016, pp. 191–220.

[2] M. Mirzadeh, A. Guittet, C. Burstedde, and F. Gibou, "Parallel level-set methods on adaptive tree-based grids," *J. Comput. Phys.*, vol. 322, pp. 345–364, Oct. 2016.

[3] Y. Yang, J. Yang, L. Liu, and N. Wu, "High-speed target tracking system based on a hierarchical parallel vision processor and gray-level LBP algorithm," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 47, no. 6, pp. 950–964, Jun. 2017.

[4] F. Kong, R. H. Stogner, D. R. Gaston, J. W. Peterson, C. J. Permann, A. E. Slaughter, and R. C. Martineau, "A general-purpose hierarchical mesh partitioning method with node balancing strategies for large-scale numerical simulations," in *Proc. IEEE/ACM 9th Workshop Latest Adv. Scalable Algorithms Large-Scale Syst. (scalA)*, Nov. 2018, pp. 65–72.

[5] M. Guo and I. Nakata, "A framework for efficient data redistribution on distributed memory multicomputers," *J. Supercomput.*, vol. 20, no. 3, pp. 243–265, Nov. 2001.

[6] J.-W. Huang and C.-P. Chu, "An efficient communication scheduling method for the processor mapping technique applied data redistribution," *J. Supercomput.*, vol. 37, no. 3, pp. 297–318, Sep. 2006.

[7] A. Gonzalez-Escribano, Y. Torres, J. Fresno, and D. R. Llanos, "An extensible system for multilevel automatic data partition and mapping," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 5, pp. 1145–1154, May 2014, doi: 10.1109/TPDS.2013.83.

[8] S. Kadir and T. Khan, "Parallel ray tracing using MPI and OpenMP: Introduction to high performance computing," Rep., Semantic Scholar, 2008.

[9] S. A. Kadir and T. Khan, "Parallel ray tracing using MPI: A dynamic load-balancing approach," Rep., Semantic Scholar, 2009.

[10] A. Stepanov and M. Lee, "The standard template library," HP Lab., Palo Alto, CA, USA, Tech. Rep. 95-11(R.1), 1995.

[11] R. Barik, Y. Yan, Y. Zhao, V. Sarkar, Z. Budimlic, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, and S. Taşrlar, "The habanero multicore software research project," in *Proc. 24th ACM SIGPLAN Conf. Companion Object Oriented Program. Syst. Lang. Appl.*, 2009, pp. 735–736.

[12] 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*, vol. 40, no. 10, pp. 519–538, 2005.

[13] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cavé, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with MPI," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process. (IPDPS)*, May 2013, pp. 712–725.

[14] V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, and V. Sarkar, "HabaneroUPC++: A compiler-free PGAS library," in *Proc. 8th Int. Conf. Partitioned Global Address Space Program. Models*, 2014, p. 5.

[15] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati, "Targeting distributed systems in fastflow," in *Proc. 18th Int. Conf. Euro-Par Parallel Process.*, Rhodes Island, Greece. Berlin, Germany: Springer-Verlag, 2012, pp. 47–56.

[16] K. Fürlinger, C. Glass, J. Gracia, A. Knüpfer, J. Tao, D. Hünich, K. Idrees, M. Maiterth, Y. Mhedheb, and H. Zhou, "DASH: Data structures and algorithms with support for hierarchical locality," in *Proc. 20th Int. Conf. Euro-Par Parallel Process.*, Porto, Portugal. Cham, Switzerland: Springer, 2014, pp. 542–552.

[17] T. Fuchs and K. Fürlinger, "A multi-dimensional distributed array abstraction for PGAS," in *Proc. IEEE 18th Int. Conf. High Perform. Comput. Commun.; IEEE 14th Int. Conf. Smart City; IEEE 2nd Int. Conf. Data Sci. Syst. (HPCC/SmartCity/DSS)*, Dec. 2016, pp. 1061–1068.

[18] Z. Szugyi, M. Török, and N. Pataki, "Towards a multicore C++ standard template library," in *Proc. Workshop Gener. Technol. (WGT)*, 2011, pp. 38–48.

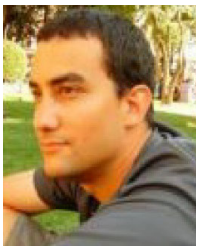
[19] T. J. Sheffler, "A portable MPI-based parallel vector template library," Res. Inst. Adv. Comput. Sci., Mountain View, CA, USA, Tech. Rep. RIACS-TR-95.04, 1995.

[20] J. Singler, P. Sanders, and F. Putze, "MCSTL: The multi-core standard template library," in *Proc. 13th Int. Euro-Par Conf. Euro-Par Parallel Process.*, Rennes, France. Berlin, Germany: Springer-Verlag, 2007, pp. 682–694.

[21] J. Fresno, A. Gonzalez-Escribano, and D. R. Llanos, "Extending a hierarchical tiling arrays library to support sparse data partitioning," *J. Supercomput.*, vol. 64, no. 1, pp. 59–68, Apr. 2013.
[22] J. F. Bausela, "Supporting general data structures and execution models in runtime environments," Ph.D. dissertation, Dept. Informática, Univ. de Valladolid, Valladolid, Spain, 2015.
[23] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-4, no. 4, pp. 308–320, Dec. 1976.
[24] M. H. Halstead, *Elements of Software Science*, vol. 7. New York, NY, USA: Elsevier, 1977.
[25] J. Singler and B. Konsik, "The GNU libstdc++ parallel mode: Software engineering considerations," in *Proc. 1st Int. Workshop Multicore Softw. Eng.*, 2008, pp. 15–22.
[26] L. Frias and J. Singler, "Parallelization of bulk operations for STL dictionaries," in *Euro-Par 2007 Workshops: Parallel Processing*. Berlin, Germany: Springer-Verlag, 2007, pp. 49–58.
[27] (2011). *RayTrace Code*. Accessed: Aug. 22, 2016. [Online]. Available: http://www.purplealienplanet.com/node/20
[28] M. H. Willebeek-LeMair and A. P. Reeves, "Strategies for dynamic load balancing on highly parallel computers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 9, pp. 979–993, Sep. 1993.
[29] B. Wagar, "Hyperquicksort: A fast sorting algorithm for hypercubes," in *Proc. Hypercube Multiprocessors*, vol. 1987, 1987, pp. 292–299.
[30] Rochester Institute of Technology. (2003). *MPI Implementation of Hyperquicksort*. Accessed: Aug. 22, 2016. [Online]. Available: https://www.cs.rit.edu/usr/local/pub/ncs/parallel/mpi/hqs.c

ANA MORETON-FERNANDEZ graduated in electronic engineering from the University of Valladolid, Spain, in 2013. She received the M.S. degree in research for information and communication technologies and the Ph.D. degree in computer science from the University of Valladolid, in 2017.

She has worked with *Renault* as a Motor-Control Engineer, from 2017 to 2018, and *MAHLE ebikemotion* as a Firmware Engineer, from 2018 to 2020. She is currently the Leader of the Software Team, *Satlantis*, working in embedded and ground-station software for operating high-resolution cameras in satellites. Her research interests include parallel and heterogeneous computing, and firmware and electronics for embedded computing.



YURI TORRES DE LA SIERRA received the B.S. degree in computer science and engineering, the M.S. degree in information communications, and the Ph.D. degree in computer science from the University of Valladolid, Spain, in 2009, 2010, and 2014, respectively.

From 2014 to 2017, he was an Associate Professor with Isabel I University, Burgos, Spain. He is currently an Assistant Professor of computer science with the Universidad de Valladolid. His research interests include parallel and distributed computing, parallel programming models, and embedded computing.

ARTURO GONZALEZ-ESCRIBANO received the M.S. and Ph.D. degrees in computer science from the University of Valladolid, Spain, in 1996 and 2003, respectively.

Since 2008, he has been an Associate Professor of computer science with the Universidad de Valladolid. He has participated in more than 100 scientific papers in journals and conferences. He has been a principal researcher of national funded projects, lead several research contracts with enterprises, and participated in the committee of several international conferences. His research interests include parallel and distributed computing, parallel programming models, portability in heterogeneous systems, and embedded computing.

DIEGO R. LLANOS (Senior Member, IEEE) received the M.S. and Ph.D. degrees in computer science from the University of Valladolid, Spain, in 1996 and 2000, respectively. He has co-founded RDNest, a company that transfers to market research results in the field of Internet of Things and high-performance computing. He is currently a Full Professor of computer architecture with the Universidad de Valladolid. His research interests include parallel and distributed computing, automatic parallelization of sequential code, and embedded computing. He is a Senior Member of ACM. He was a recipient of the Spanish Government's National Award for Academic Excellence.