ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/380547114

Four abstract array distribution operators

Conference Paper · January 2016

DOI: 10.5281/zenodo.11213594

CITATIONS
0

READS
4

3 authors:

Ana Moreton-Fernandez
Universidad de Valladolid
16 PUBLICATIONS **51** CITATIONS

SEE PROFILE

Arturo Gonzalez-Escribano
Universidad de Valladolid
144 PUBLICATIONS **801** CITATIONS

SEE PROFILE

Diego R. Llanos
Universidad de Valladolid
155 PUBLICATIONS **915** CITATIONS

SEE PROFILE

All content following this page was uploaded by Diego R. Llanos on 14 May 2024.

The user has requested enhancement of the downloaded file.

Noname manuscript No.
(will be inserted by the editor)

Four abstract array distribution operators

Ana Moreton-Fernandez · Arturo Gonzalez-Escribano · Diego R. Llanos

Received: date / Accepted: date

Abstract Programming for distributed-memory systems imposes specific challenges. In these systems, minimizing synchronization and communication overheads is key for performance improvement. A typical approach is to use a message-passing paradigm to exploit static partition policies and to generate coarse-grain computations with aggregated communication phases. However, irregular or data-dependent programs that need dynamic data redistributions become more complicated to develop and debug.

In this paper we present four abstract array distribution operators that allow to efficiently implement programs on distributed-memory systems, making the data partition, relocation, and data movement transparent to the programmer. Our experimental results show that our approach does not imply significant overheads, while achieving good scalability for combinations of data-dependent, irregular, or recursive parallel structures on distributed-memory systems.

Keywords distributed-memory systems · data redistributions · external libraries

1 Introduction

The message-passing paradigm (implemented for example by MPI libraries) has been shown to be a programming method for distributed-memory systems

Ana Moreton-Fernandez
Departamento de Informatica, Universidad de Valladolid, Campus M. Delibes, 47011 Valladolid, Spain
E-mail: ana@infor.uva.es

Arturo Gonzalez-Escribano
E-mail: arturo@infor.uva.es

Diego R. Llanos
E-mail: diego@infor.uva.es

that leads to highly efficient programs in terms of performance. However, the programmer still has to deal with many decisions not related with the parallel algorithms, but with implementation issues, such as decisions about partition and locality vs. synchronization/communication costs, scheduling details, etc.

Parallel programming models based on tasks, data-flow approaches, or the creation of data dependence graphs are commonly used in dynamic, recursive, and divide & conquer applications. Many of this kind of programming models have been proposed during the last years. These works provide an easy-to-use methodology to parallelize dynamic applications, that reduces the effort for the programmer. However, the task creation and destruction, the management of distributed queues, the synchronization and load balancing mechanisms, or the data communications due to dynamic task scheduling and/or migration, produce performance penalties, especially in distributed-memory systems. Programs with static-scheduled processes that perform coarse-grain computation and communication phases can minimize synchronization overheads. However, applications that use irregular domains have a very limited support in compile-time frameworks that automatically generate static-scheduled programs from sequential codes [4]. Thus, the ease of managing parallelism using distributed data has become an important feature for software developers.

In this paper we present four array distribution operators to efficiently implement distributed-memory algorithms, making the data partition, relocation, and data movement transparent to the programmer. These unary operators are applied on an array divided across distributed processes, and the result is another array containing all or part of the original array elements relocated on the available processes. These operators can be freely combined, even in a recursive algorithm.

To show the applicability of our proposal we have developed the four operators in the Trasgo system [10], a parallel programming model and compilation framework to generate parallel programs from a high-level parallel specification. As an example, we show how these four operators allow to implement for distributed-memory systems many of the array routines included in the C++ Standard Library, also known as STL [19], which is a foundation for many complex algorithms. In summary, our main contributions are the following:

- The design of four new operators that allow the efficient development of recursive or irregular array-based algorithms on distributed-memory systems.
- The implementation of these operators in a high-level parallel programming model; including new supporting communication structures in its runtime system.
- We have developed a parallel implementation of many algorithms of the C++ STL library using the proposed operators to show their applicability and benefits.

We present an experimental study, using applications based on STL routines, to show that our proposal achieves a good performance and scalability for data-dependent, irregular, or recursive applications.

```
** Distributed algorithm in message-passing
Inputs:
  int Size: Vector size
  int sel: Elements to update
  int id: Local process identifier
  int P: Number of processes
  A<type> M[]: Distributed Vector,
               with initial values
Outputs:
  A<type> M2[]: Distributed Vector

1. myRange.begin= id * Size/P;
   myRange.end= myRange.begin + Size/P -1;
   for ( i=myRange.begin;
         i<=myRange.end; i++)
     sum=sum+M[i]
2. AllReduce (sum)

** Calculate a redistribution
3. myRange2.begin= id * sel/P;
   myRange2.end= myRange2.begin + sel/P -1;

   for(id_p=0; id_p< P ;id_p++){
    range.begin= id_p * Size/P;
    range.end= range.begin + Size/P -1;
    range2.begin= id_p * sel/P;
    range2.end= range2.begin + sel/P -1;

    Range send_p =
    intersect(<myRange.begin,myRange.end>,
              <range2.begin,range2.end>);
    Send(M, send_p, id_p);
    Range recv_p =
    intersect(<myRange2.begin,myRange2.end>,
              <range.begin, range.end>);
    Recv(M2, recv_p, id_p);
   }

 4. for ( i=myRange2.begin;
          i<=myRange2.end; i++)
      M2[i]=M2[i]* sum
```

```
** Distributed algorithm
   using proposed operators

 Inputs:
   int Size: Vector size
   int sel: Elements to update
   int id: Local process identifier
   int P: Number of processes
   Map L: Mapping function
   A<type> M[]: Distributed Vector,
               with initial values

 Outputs:
   A<type> M2[]: Distributed Vector

 1. for ( i=L(id,Size,P).begin;
          i<=L(id,Size,P).end; i++)
      sum=sum+M[i]
 2. AllReduce (sum)

 3. M2= ArrayRemapRange(M,<0,sel-1>,L)

 4. for ( i=L(id,sel,P).begin;
          i<=L(id,sel,P).end; i++ )
      M2[i]=M2[i]*sum
```

Fig. 1 Motivating example algorithms using two different approaches: Reference algorithm in message-passing style (left); and programmed using the proposed operators in a single-program-multiple-data model (right).

The rest of the paper is organized as follows: Section 2 shows a motivating example. Section 3 describes the proposal. Section 4 presents the Trasgo model. Section 5 describes the proposal implementation. Section 6 shows an experimental evaluation. Section 7 discusses some related work. Section 8 presents the conclusions and future work.

2 Motivating example

This section presents a motivating example to show the advantages of our proposal. We have chosen an example where redistributing data is needed to create load-balance and improve efficiency. However, programming this data redistribution in a plain message-passing model is complicated and error-prone.

The parallel algorithm of the motivating example using a message-passing approach is presented on the left of Fig. 1. It sums in parallel all the elements of the whole vector (each process sums its local part and reduces the result), and then, it updates also in parallel the elements of a sub-selected range of the whole array using the previously calculated sum (each process updates its elements assigned after balancing the load). The input parameters are: (1) An integer $Size$ that determine the size of the input array. (2) An integer named sel that determines the amount of elements to update. (3) An integer id that represents the identifier of the local process. (4) An integer P that determine the number of processes, and (5) an array M of $Size$ elements distributed among the P processes. The output will be a distributed array $M2$ containing the first sel elements of M updated.

Let $Range$ be the type to represent a contiguous subdomain of indexes expressed as a pair of natural numbers $\langle begin, end\rangle$. Let $L : \mathbb{N}^3 \rightarrow Range$ be a *Mapping function*, where $L(id, Size, P)$ returns the range of indexes to be mapped to the process id. For simplicity, in this example, we assume that $Size$ is divisible by P, and that the mapping function L assigns to each process $Size/P$ elements with contiguous indexes. Finally, let $A_L\langle type\rangle$ denote an array A distributed using the mapping function L.

In the first stage of the algorithm, each process calculates the sum of the elements of its local part (being *myRange.begin* the first element assigned to the local process, and *myRange.end* the last one). Then, a reduction is needed to sum the calculated values of the different process (Stage 2 of Fig. 1 (left)). Our example updates only the first sel elements of the input array. For this reason, the program first redistributes the sel selected elements in a distributed-balanced output array among the available processes (Step 3 in the left of Fig. 1). Finally in the stage 4, each process updates its new local part using the sum calculated in Stage 2.

A data redistribution (stage 3) is not needed, but it is desirable to balance the computational load. For example, if we execute the application with $Size = 10\,000$, $sel = 2\,000$ and $P = 4$, without redistributing the data, as long as we apply a function only on the first 2 000 elements, the computation will be performed only by the first process in a sequential way, and we will not take advantage of parallelism at that stage. To achieve a good performance, we should perform a data redistribution. Thus, in the case of $Size = 10\,000$, $sel = 2\,000$ and $P = 4$, each process computes 500 elements.

On the right of Fig 1, we show the motivating algorithm using one of our operators that performs a transparent data redistribution operation. As we observe in the figure, the programming effort is highly reduced avoiding to the programmer the need to deal with all the necessary communications.

Data redistributions are only appropriated when the computation is performed in a unbalanced way according to the initial data distribution. They are convenient when the overhead produced by the communications needed to perform the redistribution is less than the potential performance gain. Moreover, there are algorithms based on recursive, divide & conquer, or similar paradigms, like for example *QuickSort*, which imply dynamic modifications or

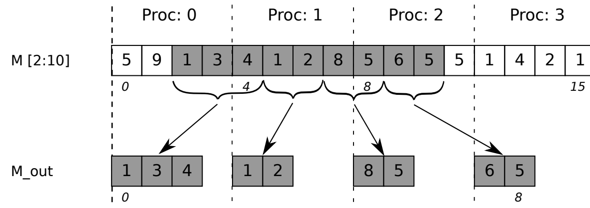

Fig. 2 Data redistribution performed by the *ArrayRemapRange* operator. In this case the call to the operator is $M_out = ArrayRemapRange(M, \langle 2, 10 \rangle, L)$.

subselections of array structures where data redistributions are totally necessary to execute the algorithms on distributed-memory systems.

3 Proposal: Redistribution operators

In this section we describe four new high-level operators to perform array-data redistributions at runtime. They can be freely combined, even recursively, to transparently implement the communication structures of a wide range of array applications. Our operators receive an already-distributed array and different compulsory or optional parameters to select subdomains of the original array in different ways. The selected elements are redistributed across the whole range of available processes using a mapping function that assigns indexes to processes according to its predefined policy. The function is also a parameter for the operators. Thus, the operators are not dependent on the mapping policy chosen by the programmer. We name A_L to the array A that is distributed using the mapping function L. For simplicity, the L mapping function used in all the figures of the paper is a contiguous blocking function for homogeneous load balance.

3.1 *ArrayRemapRange*: Remap of array range

The possibility to redistribute only a given selection or range of the original array is necessary to allow the development of recursive or divide & conquer algorithms, and it is appropriated for other kinds of functions to improve load-balance (such as the case of our motivating example in Sect. 2). This operator selects a range of an already distributed input array, and copy the selected elements in a new distributed output array that is created and allocated by the operator. The interface of this operator is the following:

$$ArrayRemapRange : A_L\langle type\rangle, Range, L' \rightarrow A_{L'}\langle type\rangle$$

Both arrays (input and output) are distributed among the different processes not necessarily using the same mapping function. Thus, several data communications per process can be needed to perform the data movement. Figure 2 shows a visual representation of how this operator works.

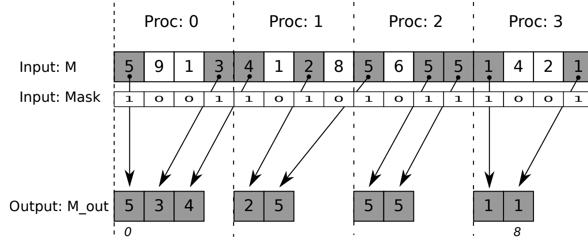

Fig. 3 Data redistribution performed by the *ArrayRemapMask* operator. In this case the call to the operator is $M_out = ArrayRemapMask(M, Mask, L)$.

3.2 *ArrayRemapMask*: Remap of an irregular selection using a mask

There are cases where the data that we want to select are not contiguous in memory. We propose a method based on *masks* to select this kind of sparse subdomains. The goal of this operator is similar to the *ArrayRemapRange* operator but using a mask to select the desired elements to be remapped. The mask is a boolean array with the same size than the input array. The mask has to be also distributed using the same partition policy that the input array. This ensures that the mask value, for any given index, is mapped to the same process as the corresponding data element of the array. The interface of this operator is the following:

$$ArrayRemapMask : A_L\langle type\rangle, A_L\langle bool\rangle, L' \rightarrow A_{L'}\langle type\rangle$$

The operator will select the elements whose associated value on the mask is 1 (true), the other ones are discarded. Figure 3 shows a visual representation of how this operator works.

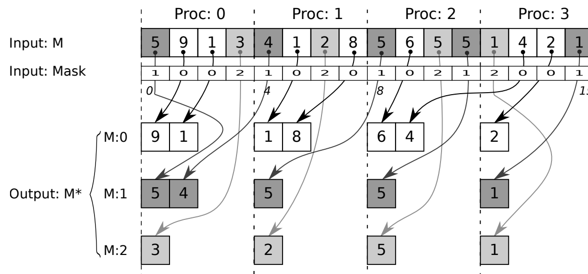

Fig. 4 Data redistribution performed by the *ArrayDivide* operator. In this case the call to the operator is $M^* = ArrayDivide(M, Mask, L, T)$. The chosen T function for this example assigns to every group all the processes.

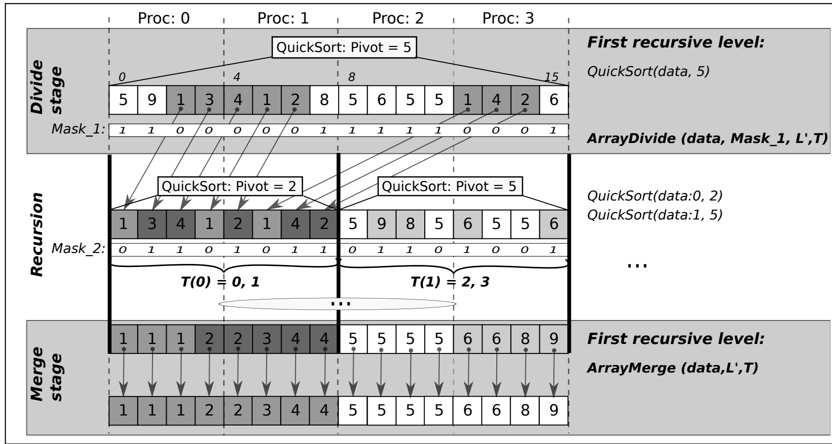

Fig. 5 Sequence of operations performed in the QuickSort algorithm in a distributed-memory system using the *ArrayDivide* and *ArrayMerge* operators. T function assigns to each group array a number of processes that is proportional to the size of the group array.

3.3 *ArrayDivide*: Dividing an array in several balanced parts using a mask

This operator is designed to tackle recursive, divide & conquer, and similar applications that need to split the data in several groups, with each group redistributed across processes independently. The operator receives an integer mask. The elements with the same natural value in the mask will be stored and redistributed in an independent output group array. Thus, the output is a collection of group arrays. The processes assigned to each group are determined by a new kind of function that assigns a subset of the processes indexes $[0..P-1]$ to each group, $T : \mathbb{N} \rightarrow P' \subset \{0, ..., P-1\}$. T is received as parameter by the operator. L' is used to redistribute each group across its assigned subset of processes. This operator is an extension of ArrayRemapMask with groups (equivalent to simultaneous instances of ArrayRemapMask), and adding the T parameter that introduces the possibility of mapping to a subset of processes. The definition of this operator is the following:

$$ArrayDivide : A_L\langle type\rangle, A_L\langle int\rangle, L', T \rightarrow \langle A^0_{L'}\langle type\rangle, ..., A^n_{L'}\langle type\rangle\rangle$$

In Fig. 4 we show an example of how this operator works for a specific mask with three group numbers (0, 1, 2), and a T function that assigns to every group all the processes.

As another example, the *QuickSort* algorithm can be programmed in parallel using this operator (see Fig. 5). The algorithm divides a large array into two smaller sub-arrays: the elements lower and the elements higher than a pivot element. The program creates a mask that keeps the information about if each element is lower or higher than the pivot (0 if the value is lower, or 1 if it is higher, as we see in the *Divide* stage). Using this mask, the program calls the *ArrayDivide* operator, which perform the data repartition. After that,

the algorithm recursively calls again the *QuickSort()* function for each group array (see *Recursion* stage). For proper load balance, in *QuickSort* we use a T function that assigns to each group array a number of processes that is proportional to its size. In the figure, for the first group array the function T returns the processes $0,1$ and for the second group array the processes $2,3$.

3.4 *ArrayMerge*: Merging array parts

This fourth operator is designed to merge several distributed arrays in one. This operator receives an array of group arrays and concatenates them in one distributed output array remapping the data, in terms of the results of a mapping function and a processes assignment function T. The definition of this operator is the following:

$$ArrayMerge : \langle A_L^0\langle type\rangle, ..., A_L^n\langle type\rangle\rangle, L', T \rightarrow A_{L'}\langle type\rangle$$

Such operation is needed, for example in the *Merge* stage of the *QuickSort* algorithm. In *QuickSort*, it merges all the sorted subarrays when exiting the recursion. Figure 6 shows a visual representation of how this operator works.

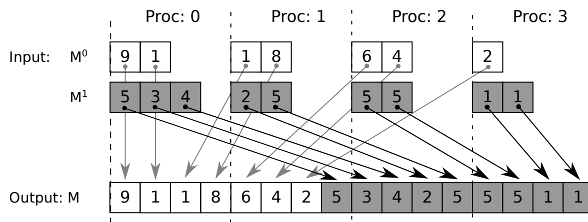

Fig. 6 Operation performed by the *ArrayMerge* operator. In this case the call to the operator is $M = ArrayMerge(M^*, L, T)$. The chosen T function for this example assigns to the output array all the processes.

4 Trasgo, CMAPS and Hitmap in a nutshell

In this section we briefly present Trasgo, the parallel programming model that we have chosen to implement our solution. Trasgo is adequate to implement our solution due to it has independent mapping functions, hierarchical data selections on distributed data, and aggregated data communications. The Trasgo model [10] proposes the use of an explicitly parallel but high-level and structured representation of parallel algorithms. The input language (currently CMAPS) is expressed in nested bulk synchronous steps, while the generated code can relax these synchronizations to improve efficiency using point-to-point communications when it is appropriated.

Trasgo transforms the global address space into a partitioned address space, building the functions to compute communications across virtual processes using a run-time library, named Hitmap [11].

4.1 CMAPS programming language

CMAPS [14] is the current input programming language for Trasgo. It provides both native data types, and *tile* types for more complex data structures, such as arrays. See an example of a CMAPS code for the motivating example in the left of Fig. 7. In this code, first, we define the sequential functions to apply to each element to reduce the sum and to multiply an array element by a value. The parallel function is defined using the *coordination* modifier. In its body, the *parallel* coordination statements spawn as many logical processes as indexes were defined in its control expression. In this example, the first parallel statement launches a logical process for each element in the M array accumulating the sum of all elements of M. The parallel statement contains *do:* clauses, with the functions to be executed by each logical processes and a *reduce* clause to consolidate the value of the variable. After that, the function *ArrayRemapRange()* redistributes the selected range of data across all the processes for load balancing. Range selections are expressed in CMAPS with a similar notation to the Fortran90 colon notation inside square brackets *[begin:end:stride]*. Finally, the code updates each element of M_out in parallel.

4.2 Run-time system: Hitmap

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 automatizes the partition, mapping, and communication of hierarchies of tiles, while still delivering good performance [11].

An object *hit_shape* represents a subspace of domain indexes defined as an n-dimensional rectangular parallelotope. The limits on each dimension are represented with a *hit_sig* object, containing the range limits (begin, end, and stride).

A *hit_tile* maps actual data elements to the index subspace defined by a shape. New allocated tiles internally 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 be also allocated to have their own memory space.

Topology and *Layout* abstract classes are interfaces for two different plug-in systems. These plug-in modules are selected by name in the invocation of the constructor method. Topology plug-ins implement simple functionalities to arrange physical processors in virtual topologies, with their own rules

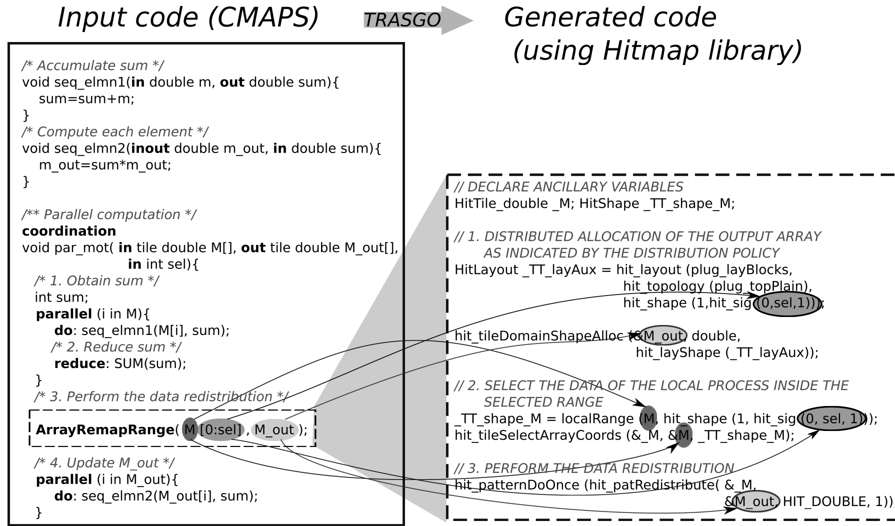

Fig. 7 Left: Trasgo input code for the motivating example (CMAPS). Right: Generated code for the *ArrayRemapRange* operator (Hitmap) in the motivating example.

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.

Finally, *hit_comm* and *hit_patt* 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.

5 Implementation of the operators

In this section we present the extensions developed in our runtime system, Hitmap, and in Trasgo to support the new operators proposed.

5.1 Supporting data redistributions at Hitmap runtime level

Trasgo framework transforms a high-level code with a global index space in a low-level code where each process has its own local index space. We have introduced in Hitmap 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 Fig. 2, 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 have also developed in Hitmap a generic redistribution communication pattern constructor (*hit_patRedistribute()*). It receives two already distributed arrays (that in Hitmap contains 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 indexes of data 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 loops traverse the process identifiers in a cyclic way, starting at the local identifier plus 1; $(myRank + 1)$ mod P. This generates a skewed communication scheme, that helps in reducing communication saturation bottlenecks on specific processes.

5.2 Implementation and code generation of the new operators in Trasgo

In this section we describe how we have implemented the support for the high-level operators in Trasgo as CMAPS functions.

We have introduced in CMAPS a *ArrayRemapRange(in_tileSelection, out_tile)* function. Figure 2 shows a diagram of how this operator works when it is used as in the expression: *ArrayRemapRange (M[2:10], M_out)*. Figure 7 (right) shows the code generated by Trasgo for this operator applied in the motivating example. It first calculates and allocates the local part of the output array (step 1). Then, it calculates and selects from the input array, the part of the selection range in the local process (step 2). The last step creates and executes the pattern containing the needed communications.

Similarly, the *ArrayRemapMask* operator has been designed for the CMAPS language with the following syntax: *ArrayRemapMask(in_tile, in_mask, out_tile)*. The code generated by Trasgo, when this operator is used, follows the same structure than the previous operator. It selects for each process the data elements that the local process has (the data elements whose mask value is 1). In this case, to select the data, we generate a loop that traverses the local domain analysing the mask (to know the selected elements), copying contiguously the needed data elements in an auxiliary array with contiguous memory, in order to perform the redistribution using ranges as in the first operator.

The *ArrayDivide* operator is designed in CMAPS as: *ArrayDivide(tile, mask, T function)*. The function creates an array of arrays, where each array stores the elements that belong to the same group. The new CMAPS notation $tile$: $group$ selects the desired group array. The code generated is similar to the second operator, but, in this case, every element is stored separately in its

corresponding array of the group after traversing the mask. A redistribution call is issued for each group.

The *ArrayMerge* operator is designed in CMAPS as: *ArrayMerge(tile, T function)*. The generated code for this operator calls the Hitmap redistribution function for each group array, concatenating the groups in consecutive ranges of the single output data.

6 Experimental studies

We have conducted experimental studies to validate our approach, and to verify the efficiency of the resulting codes that use the new operators. We present three different experimental studies. The first one, using a shared-memory system, aims to show the overhead of our distributed-memory portable approach compared with the main-trend parallel STL libstdc++ implementation. The second and the third studies, using a distributed-memory system, aim to show the scalability delivered by our proposal. We have chosen as benchmarks many routines of the Standard Template Library (STL) for one dimensional arrays (summarized in Tab. 1). During the last years, many works have presented parallel versions of this library [9, 15, 18], as well as new parallel programming models that support the development or use of this library in parallel [20]. It includes many useful algorithms, and it is a well-known supporting tool for developers [17]. For all the routines and examples tested, we have always used a mapping policy of contiguous balanced blocks.

6.1 Experimental platforms and setup

The experiments were executed in two platforms. The first one is a pure shared-memory machine (Heracles), a Dell PowerEdge R815 server, with 4 AMD Opteron 6376 processors at 2.3 GHz, with 16 cores each, and 64 cores in total. The second platform (CETA) 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 4 nodes of the cluster, we exploit up to 32 computational units. We have compiled the codes with the GCC v4.8.3 compiler, using the optimization flag *-O3*, and the flag *-fopenmp* for the codes generated for shared-memory systems. We use *mpich3* v3.0.4 as MPI implementation for the codes generated for distributed-memory systems.

6.2 Performance study on a shared-memory system

In this performance study, we show the overhead of our distributed-memory portable approach, compared with the main-trend parallel STL libstdc++ v6.0.19 implementation [16]. Due to space restrictions, in this paper we only show the results of three specific routines. They cover the different kinds of

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

Algorithm Class	Function Call(s)
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

data redistributions that together can support all the other implemented STL algorithms. These representative results can be extrapolated to the other STL routines tested. The examples we have chosen are:

- **for_each**: This example performs 1 000 floating point operations for each element on the selected range of an array. The kind of data redistribution used in this algorithm appears in most of the algorithms in the STL, to remap a selected range of a distributed array. We use the *ArrayRemapRange* operator.
- **unique_copy**: This example copies the elements of a range of an array to a range in a second array, skipping the consecutive duplicates. The size of the second data structure is created in function of the amount of elements copied from the first data structure. We use the *ArrayRemapMask* operator to select data elements that have to be eliminated.
- **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 repartitioning and merging the pivoted subarrays. We use the *ArrayDivide* and *ArrayMerge* operators.

Table 2 shows the comparison in terms of execution time, between the parallel shared-memory version included in the C++ standard template library of libstdc++ (based on OpenMP), and our approach (based on Hitmap). We see that, for algorithms like *for_each*, where the number of data redistributions is low compared with the computational load, we obtain similar performance in both codes. As expected, in cases such as the *quickSort*, we observe some penalty performance, due to the data movements and buffer replications associated to the message-passing programming. It is worth while to note that, in shared-memory programming models, communicating pointer values is enough to recursively assign the subarrays to the working threads. Even so, we see that we achieve a good scalability in the execution time when we increase the number of processes. The extreme case is the *unique_copy* routine. Due to the low

Table 2 Performance results (in seconds) for experiments in Heracles, the shared-memory system, using 10^7 elements.

	for_each		**quickSort**		**unique_copy**	
Processors	**Ref C++**	**Trasgo**	**Ref C++**	**Trasgo**	**Ref C++**	**Trasgo**
1	5.43	5.45	1.57	3.26	0.025	0.72
4	1.36	1.38	0.39	1.13	0.023	0.19
8	0.68	0.74	0.22	0.91	0.029	0.21
16	0.34	0.40	0.13	0.65	0.029	0.14
32	0.18	0.21	0.08	0.37	0.03	0.06
64	0.10	0.11	0.07	0.21	0.035	0.04

computational load and calculations performed only by the master in the libstdc++ implementation, it does not reach a good scalability. Our approach introduces a penalty due to extra calculations and communications inherent to the distributed-memory management. These extra costs make a noticeable difference when the number of processes is low, although when the number of processes increases the gap is quickly reduced.

6.3 Redistribution load balancing impact on scalability

In this section, we study the performance obtained on a distributed-memory system when comparing our redistribution approach with a solution that does not apply data redistributions for load balancing. We test the *for_each* routine with 1 000 floating point operations like in the previous experimental study on an array of 10^7 elements (*for_each* routine contains the operator that is more used in the rest of the STL routines implemented). The first CMAPS implementation redistributes the data in the selected range across all the processes to balance the computational load (*Redis*). The second CMAPS implementation does not include data redistributions (*Without-Redis*), and each process works with its originally mapped data that are in the selected range (if any). Both 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, that is the focus of our study.

We have designed experiments to study the impact of two parameters in the data-redistribution operations:

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

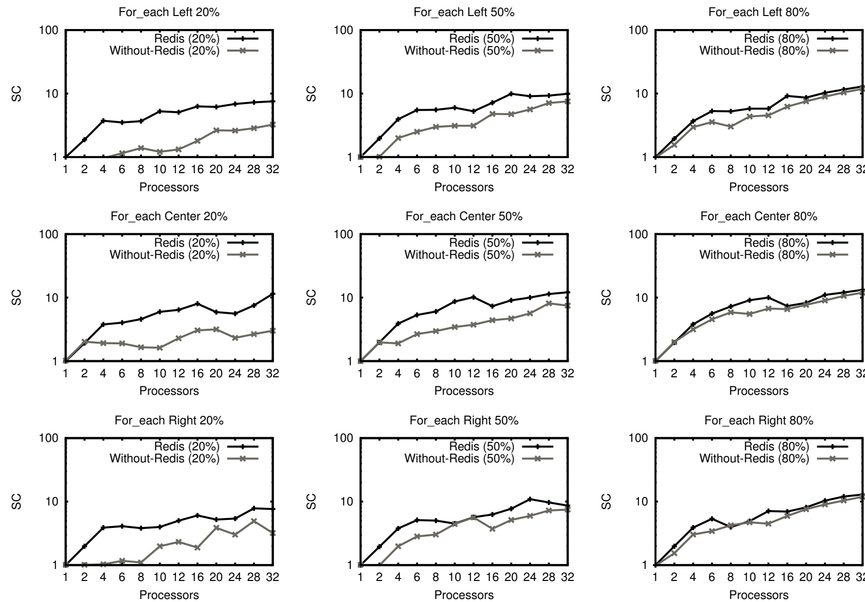

Fig. 8 Performance scalability results for the *for_each* algorithm in CETA, the distributed-memory system (logarithmic scale). $SC(p) = time(p)/T_1$ being T_1 the time spent for one process in *Redis* implementation.

Figure 8 shows the performance obtained for the different versions and parameters of the *for_each* routine in CETA, the distributed-memory system. We observe that, when the data selection is 80%, both codes obtain approximately same performance. However, when the amount of data selection is low (20 or 50%) the performance obtained using our approach is significantly better than the codes which do not use it. Our approach redistributes the range of data that needs computation, avoiding idle processors and creating load balance that alleviates the extra communication costs, only increasing the development effort in one line.

6.4 Scalability of a combination of routines/operators

The last experimental study analyzes the performance obtained by an application that use a combination of several STL routines on a distributed-memory system. We have selected a simple algorithm that combines the three previous routines discussed, together with a *find* routine that implies a reduction. This covers the main redistribution combinations using the four operators. See the algorithm in Fig. 9 (left). First, the application sorts an array. Then, it applies the *unique_copy* routine to eliminate the duplicated elements. After that, the program applies to all the non-duplicated elements a function (executing 1 000 floating point operations for each element). It finally finds the first element that fulfils a condition using the *find* routine.

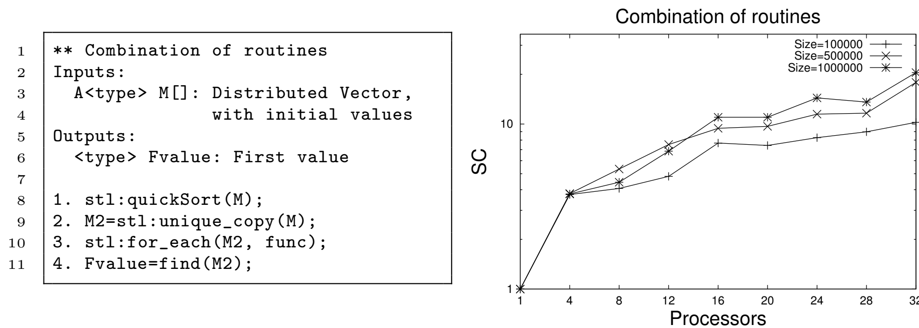

Fig. 9 Algorithm (left) and scalability results (right) for an application combining different STL routines in CETA, the distributed-memory system (logarithmic scale). *func* applies 1000 floating point dummy operations to simulate real applications load. $SC(p) = time(p)/time(1)$.

Figure 9 (right) shows the scalability obtained by this application in CETA, the distributed-memory system, for different data input sizes, and using different number of processes. We observe that our approach, despite the overhead produced by dealing with distributed data, and applying irregular data-dependent operations, allows to transparently obtain a good scalability.

7 Related Work

Many parallel programming languages and models have arisen following the task-parallelism approach [3,6,7,13] for both shared- and distributed-memory systems, with the goal of abstracting to the programmer decisions about load-balancing, granularity, etc. as our proposal. All these programming models based on task parallelism generate performance penalties, especially in distributed-memory systems due to, for example, the task creation and destruction, the management of distributed queues, or the synchronization and load balancing mechanisms. In data-flow approaches, such as the distributed-memory extension for FastFlow [1], the task construction implies a data partition and a dynamic control of task that leads to load balance. However, this dynamic scheduling prevents the exploitation of affinities and data locality across tasks. In our proposal, we use a fixed scheduled SPMD model, focused on exploiting data localities, and redistributing only the data when it is necessary. PGAS (Partitioned Global Address Space) languages, such as Chapel [5], X10 [6], or UPC [8], present a middle point approach by explicitly managing local and global memory spaces. Some PGAS models like Chapel, present an abstraction to work with mixed distributed- and shared-memory environments at the same level than Trasgo. However, they cannot aggregate communications derived from irregular data accesses. Our proposed operands perform only one aggregated communication step for each operator, deriving in a reduced number of coarse-grained computations and communications.

There are many parallel code generators based on data-parallelism that transform code to statically scheduled coarse-grain processes at compile time. Most of them are based on the polyhedral model [4]. All the techniques presented so far need to parametrize the iteration space polyhedra and to analyse dependencies at compile time. Unlike these approaches, our approach can tackle nested, recursive, or data-dependent applications with dynamic behavior, such as Quicksort, etc.

There are many works that provide external libraries to ease the parallel programming or to demonstrate the power of a parallel language. In particular, the STL library has been one of the most parallelized and studied libraries in the literature. For example, the work in [20] presents an implementation for multicore architectures of the STL library using Cilk++. Works like [15, 18] developed parallel versions of this library for shared- and distributed-memory systems using OpenMP and MPI respectively. Another works, such as HPC++ [12], also includes more sophisticated distributed versions of the STL containers, iterators and algorithms. STAPL [2] provides, through the usage of the STL library, a model of parallelism that supports recursive parallelism and recursive data decomposition, generating a data dependence graph to distribute tasks among the processes ensuring the right execution order. Our proposed operators provide similar programming expressiveness as these STL approaches, while also can be used to program new routines to cover different applications with new synchronization and communication structures.

8 Conclusion and future work

This paper presents four array data-redistribution operators to efficiently implement distributed-memory algorithms, making the data partition, relocation and data movement transparent to the programmer. Experimental results show that our implementation of the STL routines for distributed memory in terms of the four proposed operators achieves good scalability for data-dependent, irregular, or recursive parallel applications in distributed memory. Future work includes the automatic generation of parallel codes from sequential ones, exploiting the presented four operators, and the implementation of the four operators as library functions to be used in other SPMD models.

Acknowledgment

This research has been partially supported by MICINN (Spain) and ERDF program of the European Union: HomProg-HetSys project (TIN2014-58876-P), and COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS). By 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. Aldinucci, M., Campa, S., Danelutto, M., Kilpatrick, P., Torquati, M.: Targeting distributed systems in fastflow. In: European Conference on Parallel Processing, pp. 47–56. Springer (2012)
2. An, P., Jula, A., Rus, S., Saunders, S., Smith, T., Tanase, G., Thomas, N., Amato, N., Rauchwerger, L.: STAPL: an adaptive, generic parallel C++ library. In: Languages and Compilers for Parallel Computing, pp. 193–208. Springer (2001)
3. Barik, R., Budimlic, Z., Cave, V., Chatterjee, S., Guo, Y., Peixotto, D., Raman, R., Shirako, J., Taşırlar, S., Yan, Y., et al.: The Habanero multicore software research project. In: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, pp. 735–736. ACM (2009)
4. Bondhugula, U.: Compiling affine loop nests for distributed-memory parallel architectures. In: Proc. SC'2014. ACM, Denver, CO, USA (2013)
5. Chamberlain, B., Deitz, S., Iten, D., Choi, S.E.: User-defined distributions and layouts in Chapel: Philosophy and framework. In: 2nd USENIX Workshop on Hot Topics in Parallelism (2010)
6. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., Von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. Acm Sigplan Notices **40**(10), 519–538 (2005)
7. Chatterjee, S., Tasırlar, S., Budimlic, Z., Cavé, V., Chabbi, M., Grossman, M., Sarkar, V., Yan, Y.: Integrating asynchronous task parallelism with MPI. In: Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, pp. 712–725. IEEE (2013)
8. El-Ghazawi, T., Carlson, W., Sterling, T., Yelick, K.: UPC: distributed shared-memory programming. Wiley-Interscience (2003)
9. Frias, L., Singler, J.: Parallelization of bulk operations for STL dictionaries. In: Euro-Par 2007 Workshops: Parallel Processing, pp. 49–58. Springer (2007)
10. Gonzalez-Escribano, A., Llanos, D.: Trasgo: A nested-parallel programming system. The Journal of Supercomputing **58**(2), 226–234 (2011)
11. Gonzalez-Escribano, A., Torres, Y., Fresno, J., Llanos, D.: An extensible system for multilevel automatic data partition and mapping. IEEE TPDS **25**(5), 1145–1154 (2013). (doi:10.1109/TPDS.2013.83)
12. Johnson, E., Gannon, D.: HPC++: Experiments with the parallel standard template library. In: Proceedings of the 11th international conference on Supercomputing, pp. 124–131. ACM (1997)
13. Kumar, V., Zheng, Y., Cavé, V., Budimlić, Z., Sarkar, V.: HabaneroUPC++: A compiler-free PGAS library. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, p. 5. ACM (2014)
14. Moreton-Fernandez, A., Gonzalez-Escribano, A., Llanos, D.R.: A new high-level parallel portable language for hierarchical systems in Trasgo. In: Computational and Mathematical Methods in Science and Engineering (CMMSE) (2015)
15. Sheffler, T.J.: A portable MPI-based parallel vector template library. Tech. Rep. RIACS-TR-95.04, Research Institute for Advanced Computer Science (1995)
16. Singler, J., Konsik, B.: The GNU libstdc++ parallel mode: software engineering considerations. In: Proceedings of the 1st international workshop on Multicore software engineering, pp. 15–22. ACM (2008)
17. Singler, J., Sanders, P.: The GNU libstdc++ parallel mode: Benefit from Multi-Core using the STL
18. Singler, J., Sanders, P., Putze, F.: MCSTL: The multi-core standard template library. In: Euro-Par 2007 Parallel Processing, pp. 682–694. Springer (2007)
19. Stepanov, A., Lee, M.: The Standard Template Library. Tech. Rep. 95-11(R.1), HP Laboratories
20. Szugyi, Z., Török, M., Pataki, N.: Towards a multicore C++ standard template library. In: Proc. of Workshop on Generative Technologies (WGT 2011), pp. 38–48 (2011)

View publication stats