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

Automatic Runtime Calculation of Communications for Data-Parallel Expressions with Periodic Conditions

Ana Moreton-Fernandez · Arturo Gonzalez-Escribano

Received: date / Accepted: date

Abstract Many real-world applications feature data accesses on periodic domains. Manually implementing the synchronizations and communications associated to the data dependences on each case, is cumbersome and error-prone. It is increasingly interesting to support these applications in high-level parallel programming languages or parallelizing compilers. In this paper we present a technique that, for distributed-memory systems, calculates the specific communication patterns derived from data-parallel codes with or without periodic boundary conditions on affine access expressions. It makes transparent to the programmer the management of aggregated communications for the chosen data partition. Our technique moves to runtime part of the compile-time analysis typically used to generate communication code for affine expressions, introducing a complete new technique that also supports the periodic boundary conditions. We present an experimental study to evaluate our proposal using several study cases. Our experimental results show that our approach can automatically obtain communication codes as efficient as those found in MPI reference codes, reducing the development effort.

Keywords Distributed memory · Periodic boundary condition · Communication patterns · Parallel programming

1 Introduction

Many real-world applications feature data accesses on periodic domains; domains on which at least one dimension is toroidal, and a walk through the

A. Moreton-Fernandez · A. Gonzalez-Escribano
Departamento de Informática, Universidad de Valladolid, Valladolid, Spain

A. Moreton-Fernandez E-mail: ana@infor.uva.es
A. Gonzalez-Escribano E-mail: arturo@infor.uva.es

dimension becomes an infinite tour on which the same elements of the domain are periodically traversed once and again. When this kind of domains are discretized and represented by a set of indexes, advancing past the last index in the periodic dimension, instead of getting out of the domain, means arriving at the first index. For example, physical phenomena can be modelled using spherical grids with periodic boundary conditions [25], using a stencil program over a periodic domain. Manually implementing the synchronizations and communications associated to the data dependences for each specific application, is cumbersome and error-prone. Thus, it is increasingly interesting to support these applications in high-level parallel programming languages or parallelizing compilers.

Current state-of-the art compilers focus on the transformation of high-level parallel programs (e.g. [10]), or sequential codes (e.g. [6]), to low-level parallel programs. Typically, they target static-control programs with affine expressions, using compile-time automatic techniques [11,4,17]. These techniques abstract many issues related to the execution platform, while they still deliver good performance. However, they produce a generic code that does not take into account some specific details of the execution machine. For example, the most sophisticated code generators for distributed memory [4], that only support affine expressions, reduce the volume of communicated data and are parametric in the number of processes and problem sizes. Nevertheless, they still need to fix a single tile size at compile time, even if the system has nodes with different capabilities. Moreover, the application of these techniques to periodic domains is more complicated. In general, they need to compute a transitive closure of dependencies, which is typically a very hard problem at compile time.

In this paper we present a technique that, automatically calculates at runtime the coarse-grained *communication patterns* [1] that are needed for a correct and efficient execution of SPMD (Single Program Multiple Data) programs derived from data-parallel codes with periodic affine expressions in data accesses. It makes transparent to the programmer the management of aggregated communications for the chosen data partition. This technique has been recently announced to the community [20]. It moves to runtime part of the compile-time analysis needed to generate the communication code. It produces programs more adaptable to different execution environments, allowing, for example, the use of different tile sizes at the same hierarchical level, a good approach for clusters that include machines with different architectures [19].

To show the applicability of our approach, we develop the technique in the Trasgo parallel programming system proposed in [12]. Our implementation automatically generates MPI programs from abstract data-parallel expressions. We test six benchmarks with periodic access expressions: An illustrative example based on the rotate routine of the STL library [28], the periodic versions of

[1] Communication patterns: Abstract structures determining which data should be moved across each pair of processes at a given program point. For example, in the MPI message-passing interface a specific instance of an MPI_Alltoallv function call can be represented as a communication pattern.

Heat 1-D, 2-D and 3-D applications [5], a matrix multiplication program using Cannon's algorithm [8] and a multi-grid V-cycle 3D-stencil application, the NAS MG benchmark [1]. Our experimental results, comparing pure MPI reference codes and the programs automatically generated, in both distributed- and shared-memory environments, show that the use of our approach can automatically obtain efficient codes while reducing the development effort. For example, the use of our solution leads to a reduction of 44.42% in the number of code lines and a reduction of 65.71% in the McCabe Cyclomatic Complexity for the NAS MG Benchmark.

2 Related Work

Many task-oriented programming approaches (like StarSs, OmpSs [7, 24]) are based on iterators to generate pools of tasks with explicit input and output working sets. The working sets analysis allows dependence graphs to be built. However, due to the task execution model based on dynamic scheduling with task-queues, synchronizations, and dynamic mapping information, these models cannot derive aggregated communication calculations for groups of tasks. Moreover, the use of these task-oriented approaches in distributed memory leads to performance penalties in the general case, due to the task creation and destruction, the management of distributed queues, the synchronization and load balancing mechanisms, and the data communications due to dynamic task scheduling and/or migration. Our approach minimizes synchronization overheads, by generating programs with static-scheduled processes that perform coarse-grained computation and communication phases.

PGAS (Partitioned Global Address Space) models present an abstraction to work with mixed distributed- and shared-memory environments similar to Trasgo. The PGAS language that is more closely related to our work is Chapel [9]. It proposes a separation of domain and mapping modules to generate distributed arrays. However, the best communication aggregation methods presented so far for Chapel abstractions are restricted to specific operations, or domain mapping properties. For example, the work in [26] is restricted to global array assignments with block or cyclic distributions. The work in [27] presents a symbolic substitution of mapping attributes in affine access expressions with the same inspiration as our approach. However, the Chapel runtime cannot aggregate several expressions across different loops to generate the full task footprint. Also, it needs to rely on non-aggregate communications when the whole set of data accessed by an expression is not fully allocated in the same remote processor. It only works for cyclic or block-cyclic distributions.

Fortran-D compiler techniques for calculating communication in SPMD programs [15] use domain calculations to generate, at compile time, different communication code depending on the data partition selected. This constrain avoids to change data partition features at runtime. However, performing a data partition based on the details of the target machines is key to achieve a good performance and a balanced workload, especially in distributed-memory systems that include machines with different architectures.

The polyhedral model provides a formal framework to develop automatic transformation techniques at the source code level to generate low-level code for shared- or distributed-memory systems [4,2,31,16]. Using the current distributed-memory approaches, communications cannot be calculated across different affine loop nests sections unless loop fusion can be done. Moreover, using these methods there are still cases for duplicated or unnecessary data communications, that our proposal can avoid, as it will be shown in Sect. 6. The work in [17] presents a hybrid compiler-runtime translator scheme similar to our approach that calculates the communication pattern needed in an SPMD programming model. However, they only support *regular and repetitive* applications where the communication pattern is the same in all the iterations of an outer serial loop that encloses the SPMD block. Our technique is applicable to programs that change their communication structure on each iteration.

Applying the tiling technique to a set of indexes enables a medium-grained parallelization. Some approaches have been presented to solve the problem of tiling with periodic boundary conditions. The closest method to our approach is a cutting-and-pasting technique. In this technique, the dependencies which are affected by the periodic conditions are broken and displaced. It is similar to a circular loop skewing. This approach needs a computation of the transitive closure of dependencies to determine the set of iterations which another tile depends on. Some libraries such as ISL [30] are capable, for some problems, of computing at compile time the transitive closure of dependencies efficiently. However, the transitive closure computation typically is a very hard problem for solutions that work at compile time. Recently, another method to tile and optimize time-iterated computations over periodic domains was presented in [5]. This technique first splits the iteration domain, cutting close to the mid-point what their authors call *long dependences*. After this cut, they apply a separate affine transformation on each half of the space. All these approaches target shared-memory systems. Programming for distributed-memory systems is more challenging due to the management of the distributed data structures. Moreover, communications among processes should be devised in terms of, for example, message-passing operations, taking into account all the potential combinations of proper data partitions or matrix sizes. Our technique makes transparent all these issues.

3 Illustrative Example

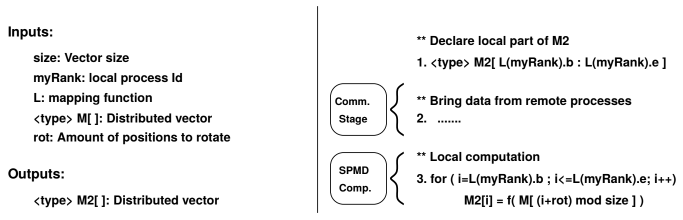
This section presents an illustrative application to show an example of how a parametrizable algorithm can be tackled with our proposed solution. We selected an example based on the *rotate* routine of the STL library [28]. Refer to the sequential algorithm in Fig. 1. The routine applies a rotation to the vector elements while applying a function $f()$ on each one of them. The routine receives an integer parameter *size*, a vector of elements (with the size indicated

```
** Sequential rotate algorithm
Inputs:   size: Vector size
          <type> M[size]: Vector with initial values
          rot: Amount of positions to rotate the elements
          f: function for computing each element

Outputs:  <type> M2[size]: Vector with result values

   1. <type> M2[size];
   2. For i = 0 to size-1
         M2[i] = f(M[(i+rot) mod size])
```

Fig. 1 Sequential algorithm for the illustrative example assuming a positive value of *rot*.



```
Inputs:
    size: Vector size
    myRank: local process Id
    L: mapping function
    <type> M[ ]: Distributed vector
    rot: Amount of positions to rotate

Outputs:
    <type> M2[ ]: Distributed vector

             ** Declare local part of M2
             1. <type> M2[ L(myRank).b : L(myRank).e ]

Comm. Stage  ** Bring data from remote processes
             2.  .......

SPMD Comp.   ** Local computation
             3. for ( i=L(myRank).b ; i<=L(myRank).e; i++)
                    M2[i] = f( M[ (i+rot) mod size ] )
```

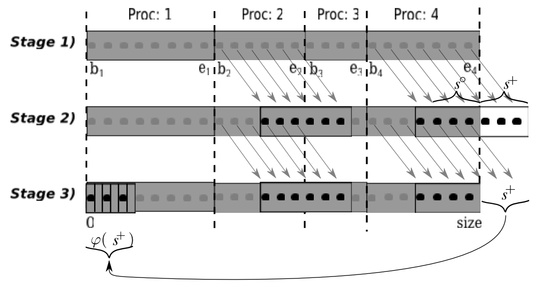
Fig. 2 Parallel algorithm for the illustrative example in a SPMD model assuming a previously distributed input array. Boxes indicate the logical steps in SPMD computations.

by the previous parameter), and another integer parameter *rot*. The parameter *rot* is used in the access expressions to select, at runtime, the amount of positions that the elements in the vector should be shifted.

A distributed parallel version of this algorithm is shown in Fig. 2. In this case, the input parameters are: (1) The size of the original input vector, *size*; (2) the local process identifier, *myRank*; (3) a mapping function L that receives a process identifier and returns the set of indexes, in the range [0:*size*-1], assigned to that process (b and e will indicate the limits of the set for this process) ; (4) An input vector M that was previously distributed among the different processes using the mapping function L; (5) The integer parameter *rot* that is assumed to be positive in this example.

First, the local part of the distributed output vector $M2$ is defined, also using the mapping function L. Thus, each process stores the same part of both arrays, M and $M2$. After that, each process participates in the rotation of the whole vector elements, and applies the function $f()$ on its part (SPMD block). When a process applies the access expression $(i + rot)\ mod\ size$ to its assigned index domain, it is possible that part of the resulted indexes are out of the assigned domain. Hence, we insert a communication stage before the SPMD block to ensure that every process has the necessary data to compute $f()$. After the communication and computation, processes will be able to update its part of the $M2$ output vector. Our technique calculates automatically these necessary communication patterns.

We illustrate the communication calculation for this example in Fig. 3. In the left of the figure, we see in Stage 1 the set of indexes assigned by the mapping function L to each process. In Stage 2, we see the set of indexes



Stage 1) Illustrative example:

$$L(p) = [b_p : e_p : 1]$$

Stage 2) Applying Expr: [(i+rot)]:

$$W_I^{M,k}(2, L, rot) = \begin{cases} s^- = \emptyset \\ s^\circ = [b_2 + rot : e_2 + rot : 1] \\ s^+ = \emptyset \end{cases}$$

$$\ldots$$

$$W_I^{M,k}(4, L, rot) = \begin{cases} s^- = \emptyset \\ s^\circ = [b_4 + rot : e_4 : 1] \\ s^+ = [e_4 + 1 : e_4 + rot : 1] \end{cases}$$

Stage 3) Applying Expr: [mod size]:

$$T(W_I^{M,k}(2, L, rot)) = \begin{cases} \psi(s^-) = \emptyset \\ s^\circ = [b_2 + rot : e_2 + rot : 1] \\ \varphi(s^+) = \emptyset \end{cases}$$

$$\ldots$$

$$T(W_I^{M,k}(4, L, rot)) = \begin{cases} \psi(s^-) = \emptyset \\ s^\circ = [b_4 + rot : e_4 : 1] \\ \varphi(s^+) = [0 : 0 + rot - 1 : 1] \end{cases}$$

Fig. 3 Communication structures calculation. Using the read data-access expression inside the parallel computation of the illustrative example to build the functions that calculate the working input indexes set (W_I) for M for the processes 2 and 4. This example uses contiguous rectangular shapes, an irregular data partition policy, and considers the particular case of $rot = 3$, for a domain with $|M_0| = 25$.

resulted by applying the expression $(i + rot)$ to the indexes assigned to the processes 2 and 4. In our example, the domain accessed by process 2 overlaps with the domain owned by process 3. Thus, in order to compute, process 2 needs to receive those data from process 3. For process 4 we obtain two sets of indexes: (a) A set located into the original array domain (it will be named s°), and (b) a set located outside of the original array domain. This set is represented in white and it will be named s^+. Notice that s^+ is a set of indexes higher than the end of the array domain, due to the positive value of rot in the example. If the value of rot had been negative, we would have had a set of indexes lower than the start of the array domain, named s^-.

The application of the periodic boundary condition (*mod size*) over the set of indexes resulted of applying $i + rot$ in process 4, transform them to the set of indexes shown in Stage 3. The white region on Stage 2 corresponds to the stripped region that belongs to process 1 at Stage 3. We will need an extra communication to receive the stripped region from process 1 in process 4.

4 Aggregated-Communication Model

In this section, we present a generic model that calculates at runtime the necessary communication patterns to compute SPMD blocks, that contain affine access expressions on the indexes of the parallel loops (SPMD blocks) with optional periodic boundary conditions. In this paper we assume an *owner-computes* paradigm, with the result of a runtime mapping function indicating the ownership. We describe the model to calculate communication patterns to move data from owner processes to the processes where these data are accessed to execute computation. The same model can be applied to move computed data to destination positions, in the case of write accesses which are generic expressions that transform the domain indexes.

In this section, first, we present the notation used in this paper. After that, a simplified one-dimensional overview of the communication calculation is presented. Finally, the generic multi-dimensional model is described.

4.1 Definitions

In this section, we define the terms used during the model description.

- **Number of elements** (N): number of elements in an array M.
- **Number of dimensions** (n): number of dimensions of an array M.
- **Cardinality** ($|M_x|$): number of indexes of the array M in the x-th dimension.
- **Signature** ($S\langle b, e, z\rangle$): is a triplet of three integer numbers *begin* (b), *end* (e), and *stride* (z). A signature defines a subset of integer one-dimensional indexes from the *begin* to the *end*, using the *stride* as step. We will use the classical Fortran90 notation $[b : e : z]$ for simplicity in our discussion. The set of all possible signatures is S^*.
- **Domain** (D_n): An n-dimensional domain formed by the Cartesian product of n signatures (a hyperrectangle). The set of all possible domains in n dimensions is denoted as D^*_n.
- **Computation indexes** ($\overrightarrow{i}$): The set of indexes where a parallel computation will be performed.
- **Affine expression** ($\rho_x(\overrightarrow{i})$): In our technique we consider affine expressions on the x-th dimension with the form:

$$\rho_x(\overrightarrow{i}) = \alpha_0 \times i_0 + ... + \alpha_{n-1} \times i_{n-1} + \beta$$

  where the coefficients α_x, β are invariant in the body of the SPMD block. We can also apply an affine expression to a whole set of indexes described by a signature ($\rho_x(s)$).
- **Periodic expression** ($cyc(\rho_x(\overrightarrow{i}))$): It denotes periodic boundary conditions on the result of an affine expression.
- **Mapping function** ($L(p)$): It is a function that receives the index of a process and returns the domain representing the set of indexes mapped to that process.
- **Access expression** ($M[cyc(\rho_0(\overrightarrow{i}))]...[cyc(\rho_{n-1}(\overrightarrow{i}))]$): An expression in the code that accesses the data in a data structure M. A periodic boundary condition is applied on each dimension of the data structure.
- **Set of affine expressions** ($\phi(\overrightarrow{i})$): The set of affine expressions (one for each dimension) of a single access expression.

4.2 Model for Calculating Communication Patterns in 1-D Applications

In this section we present an overview of the proposed technique to calculate the communication patterns for only one-dimensional index domains. Recall the parallel algorithm presented in Fig. 1, that performs the rotation of the elements of a vector also applying a function $f()$. A communication phase is needed before the SPMD block to reallocate some data across processes, ensuring that each process has the necessary data to compute.

The following analysis is done independently for each data structure, and for each SPMD block. The *Input Working Set of Indexes* ($W_I^{A,k}(p, L, \overrightarrow{\delta})$) is a function built based on the access expressions on a SPMD block. It returns the set of indexes of the data structure A, read by a given processor p, during the k-th SPMD block in the code (remind Stage 2 of Fig. 3). The parameters of the function are: The processor identifier p, a mapping function L that returns the set of indexes mapped to any processor, and $\overrightarrow{\delta}$, the values of the symbolic parameters that appear in the expressions. The function applies at runtime the affine expressions (ϕ) found in read accesses to A inside the k-th SPMD block, one by one, to the indexes set returned by the mapping function ($L(p)$). See a calculation example in Stage 2 of Fig. 3.

For the one-dimensional case, we obtain a set of indexes that can be represented by a set of signatures. These signatures can be classified as:

- Set of signatures whose indexes are lower than the begin of the original array domain (S_0^-).
- Set of signatures whose indexes intersect with the original domain (S_0°).
- Set of signatures whose indexes are higher than the end of the original array domain (S_0^+).

We generate the code of the functions that compute the set of indexes read per each SPMD block, and each data structure ($W_I^{A,k}(p, L, \overrightarrow{\delta})$). These functions return a set of signatures representing the set of indexes read. With these functions, a process can calculate the input working set of a data structure, of any process, once the parametric values are known. These functions simply apply the access expressions to the index-space limits at runtime to calculate the working-sets (see the implementation of these functions in Sect. 5).

Once we have the set of signatures that result of applying the affine expressions, we can apply the periodic boundary conditions, where it is required. This relocates the indexes that after the application of the affine expression are out of the original array domain, into the domain. See Stage 3 in Fig. 3. We define two functions to apply the periodic conditions to signatures. Remind that $|M_0|$ is the number of elements of M in the first dimension.

- For signatures in S^- we define $\psi : S^* \to S^*$ where

$$\psi(s) = s' : \begin{cases} s'.b = -((-s.b) \ mod \ |M_0|) + |M_0|, \\ s'.e = -((-s.e) \ mod \ |M_0|) + |M_0|, \\ s'.z = s.z \end{cases} \tag{1}$$

- For signatures in S^+ we define $\varphi : S^* \to S^*$ where

$$\varphi(s) = s' : \begin{cases} s'.b = s.b \ mod \ |M_0|, \\ s'.e = s.e \ mod \ |M_0|, \\ s'.z = s.z \end{cases} \tag{2}$$

ALGORITHM 1: Model to calculate the receive communication pattern for a SPMD block, for a given data structure A.

```
Input:
$P$: Number of processes;
$myRank$: Local process id;
$L()$: Mapping function;
$\overrightarrow{\delta}$ : Symbolic parameters;
$|A_0|$: Cardinality of the 1-D array;
$W_I^{A,k}()$: Function to compute the working input set;
$\psi(),\ \varphi()$: Periodic transforming functions
Output: $C_R$ : Set of comm-tuples that indicates data to be received
$C_R \leftarrow \emptyset$
for $p$: 1 to $P$ do
    $lp \leftarrow L(p)$
    for all $s \in T(W_I^{A,k}(myRank, L, \overrightarrow{\delta}))$ do
        $s' \leftarrow s \cap lp.S_0$
        if $s' \neq \emptyset$ then
            $tmp_0 \leftarrow \emptyset$
            if $p \neq myRank$ and $s \in S^{\circ}$ then
                $tmp_0 \leftarrow s'$
            end
            if $s \in \psi(S^-)$ then
                $tmp_0 \leftarrow s' - |A_0|$
            end
            if $s \in \varphi(S^+)$ then
                $tmp_0 \leftarrow s' + |A_0|$
            end
            $C_R \leftarrow C_R \cup \langle p, \langle tmp_0 \rangle \rangle$
        end
    end
end
```

A function $T : D_n^* \to D_n^*$ transforms the working input set by applying the two previous functions to relocate all the indexes back into the original array domain, as it was showed in Stage 3 of Fig. 3. We can express the transformation with the following function:

$$T(W_I^{A,k}(p, L, \overrightarrow{\delta})) = \begin{cases} if\ \ s \in S^-; s' = \psi(s), \\ if\ \ s \in S^{\circ}; s' = s, \\ if\ \ s \in S^+; s' = \varphi(s) \end{cases} \tag{3}$$

An example of the application of $T(W_I^{A,k}(p, L, \overrightarrow{\delta})$ for the illustrative example can be seen also in Stage 3 on the right of Fig. 3.

Algorithm 1 uses a simplified one-dimensional model to calculate the data to be received at any process. The output is a set of communication tuples (C_R). A comm-tuple $\langle p, D^* \rangle$ associates the index of the remote process p, with the set of indexes D^* of the structure whose data values should be communicated. For each data structure, the local process, named *myRank*, calculates the exact data to be received from a remote process p. In order to do that, local process intersects $L(p)$, which is the domain assigned to that remote process by the mapping function, with the data positions needed in read accesses, which are represented by the transformed input set at the local process, $T(W_I^{A,k}(myRank, L, \overrightarrow{\delta}))$. If the intersection is not empty, a receive communication should be performed. The data have to be received in the positions accessed by the local process, so the applied boundary conditions are reverted. The data to be sent to process p can be calculated by the opposite intersection: The assigned domain to the local process $(L(myRank))$, with the transformed input sets at the remote process $(T(W_I^{A,k}(p, L, \overrightarrow{\delta})))$. Empty intersections indicate that no send or no receive operation is needed for that particular process p. In the illustrative example of Sect. 3, the groups S^- is empty for any process p, due to *rot* being positive, and simply added to the parallel loop index in the expression. See the details in Fig. 3.

4.3 Multi-dimensional Model

As we have seen in the previous section, for the one-dimensional case, we have three groups of signatures, depending on the access expressions (S^-, S°, S^+). However, in a general case with n dimensions, the number of classes for the transformed domains, before applying boundary conditions, is 3^n. For example, let $A[cyc(\rho_0(\overrightarrow{i}))][cyc(\rho_1(\overrightarrow{i}))]$ be a two-dimensional periodic access expression. The input working set is calculated similarly as in the 1-dimensional case, simply applying the access expressions to the index-space limits at runtime for every dimension. As we use rectangular tiles, no over-approximation is performed, so the calculated input working set only contains the index space obtained by applying the access expressions on the different dimensions.

The domains that compose the 2-dimensional input working set can be classified in 3^2 possible groups of domains. The classification groups are defined as the combination of the S^-, S°, and S^+ sets of each dimension:

$$\langle S_0^\circ, S_1^\circ\rangle, \langle S_0^\circ, S_1^-\rangle, \langle S_0^\circ, S_1^+\rangle, \langle S_0^-, S_1^\circ\rangle, \langle S_0^-, S_1^-\rangle,$$
$$\langle S_0^-, S_1^+\rangle, \langle S_0^+, S_1^\circ\rangle, \langle S_0^+, S_1^-\rangle, \langle S_0^+, S_1^+\rangle$$

Figure 4 shows an example based on a Stencil-2D application with periodic boundary conditions. For simplicity in this figure, the matrix M has been divided in four equal parts that correspond to 4 processes. Thus, each process computes a part of the matrix. In this example each process needs data of its neighbors for computing. On the left of the figure, we show the domains accessed by process 3, classifying also the domains in their corresponding groups (e.g $\langle S_0^\circ, S_1^\circ\rangle, \langle S_0^\circ, S_1^+\rangle$). The groups of domains not represented in the figure are empty. On the right of the figure, we represent using arrows the corresponding receive communication operations that should be performed on each iteration by process 3, and which are automatically calculated by our proposal.

The multidimensional model uses an extension of the Alg. 1. In this case, the computation of the transformed input working set $(T(W_I^{A,k}(p, L, \overrightarrow{\delta})))$, and

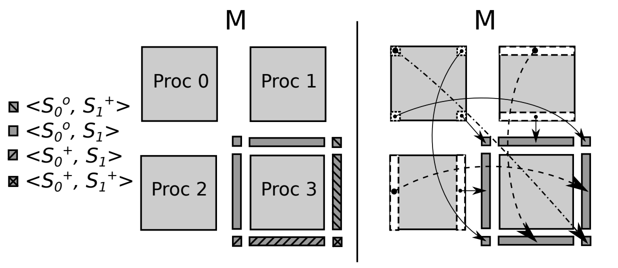


Fig. 4 Stencil-2D application: Input matrix has been divided among four processes. Left: Set of signatures accessed by process 3 and their classification. Right: Communications needed, to enable the computation at process 3, represented by arrows.

```
 1  /* Function to rotate one element */
 2  void rotateElem(in double m, out double m2 ) {  m2=f(m); }
 3
 4  /* Rotate: Parallel function */
 5  coordination void rotate( in tile double M[], in int rot,
 6                            in int size, in Map L, out tile double M2[]) {
 7   ArrayMap( M2, L );
 8   parallel(i in [0 :size-1]){
 9    rotateElem(M[cyc(i+rot)], M2[i]);
10   }
11  }
```

Fig. 5 Trasgo input code for the illustrative example.

the reversion of the resulting domains are done by applying the transforming functions (ψ and φ) independently on each dimension.

5 Implementation on a Parallel Programming Framework

In this section we briefly describe the programming tool-chain where we implemented our proposal, the Trasgo system proposed in [12].

The frontend of the Trasgo framework receives as input a high-level code developed in an explicit parallel language that simplifies the data dependencies extraction [22]. Figure 5 shows a simplified example of how to program in this parallel language the illustrative example presented in Sect. 3. First, we define the sequential function to apply to each data element, specifying the output or input role of the parameters (line 2). The parallel function is defined using the *coordination* modifier, and also specifying the role of the parameters (lines 5 to 6). In its body, the function *ArrayMap()* distributively allocates the array $M2$ in terms of the results of the mapping policy L, that is also a parameter (line 7). After the distribution, the code updates each element of $M2$ in parallel invoking, inside the *parallel* statement, the sequential function previously defined (lines 8-9).

We implemented the proposed technique on top of the Hitmap library, that works in Trasgo as a runtime system [13]. Hitmap is a library for the

```
/** Calculate W_I for M in SPMD */
HitDomain calculateWI_1_M( HitRank p, HitLayout lay, HitTile Tile1, int rot){
   // L(p)
   HitShape remote = hitLayOtherShape( lay, p );
   // 1*begin+rot, 1*end+rot, 1*stride
   HitDomain inWS = hitShapeAffine1( remote, 1,+rot);
   //Apply boundary conditions
   hitApplyBoundary(inWS, lay, Tile1);
   return inWS;
}
```

Fig. 6 Excerpt of the generated function that applies the input-code affine access expressions and periodic conditions to compute $T(W_I^{A,k}(p, L, rot))$ for the illustrative example.

management and runtime mapping of hierarchically distributed tiled arrays. It defines objects to declare and manipulate indexes sets as multidimensional parallelotopes, and provides a plug-in system that can be queried at runtime to obtain information about the result of a mapping, for both local and remote processes. Hitmap also contains functionalities to build reusable communication patterns for tiles, or subtiles, across virtual processes. They are internally implemented as collections of asynchronous MPI send/receive operations that also exploit derived data types for efficient marshalling/unmarshalling.

Mapping functions are represented in Hitmap as *HitLayout* objects; The signatures by *HitSig* objects; The hyperrectangular domains by *HitShape* objects; And the sets of domains by *HitDomain* objects. The handlers containing pointers to the actual data structures are expressed by *HitTile* objects.

We implemented the multidimensional model of the proposed technique on Trasgo, using the Hitmap features, and pre-implementing some new functions in the Hitmap library for efficient domain set operations on hyperrectangular shape structures such as intersection $\cap$, union $\cup$, and subtraction $\setminus$.

Figures 6 to 7 present an example of the functions automatically generated following the proposed technique to calculate the receive communication pattern (C_R) for the illustrative example, and the lines to be inserted in the main code to invoke these functions.

The function named *calculateWI_1_M* (see Fig. 6) is generated at compile time by Trasgo. It uses three new Hitmap functions: (1) *hitLayOtherShape* returns the domain assigned to a given remote process; (2) *hitShapeAffine1* applies an affine access expressions of the form ($\alpha_0 \times i_0 \beta$) to the index-space limits of the first dimension at runtime. (It performs the computation of Stage 2 of Fig. 3); (3) *hitApplyBoundary* applies the boundary conditions, and returns a set of signature domains bounded to the original array domain. It performs the transformation of Stage 3 of Fig. 3.

We implement the algorithm presented in section 4 into the Hitmap library as a function. It receives as parameters the mapping function used to part the data structure (a HitLayout object), and the generated for the specific piece of code to calculate the transformed input working set ($T(W_I^{A,k}(p, L, \overrightarrow{\delta}))$). The result of the new function `calcComms`, is a HitPattern object where the

```
/* Building comm. pattern */
HitPattern _TT_comA = calcComms(M, calculateWI_1_M, M.Layout, rot);
/* Communication, execute pattern */
hit_patternDo( _TT_comA );
```

Fig. 7 Calling both the communication calculation and execution functions in the target program of the illustrative example.

calculated comm-tuples have been inserted. Figure 7 shows how both the communication calculation and the execution functions are called in the target main program. A similar piece of code is automatically inserted before the execution of each SPMD block.

6 Discussion: Analyzing the Technique

Communication optimality: For a given input working set, our technique calculates the corresponding exact data communication. In the current prototype frontend, the construction of the exact input working sets is limited for some kind of programs. Future work includes the integration of polyhedral frameworks, which use more sophisticated representations [29], to extend the application range of our implementation.

In our implementation, the domains used to represent the indexes accessed in a given piece of code should be represented as hyperrectangles (signature domains). The representation of any other shape should be done as a union of signature domains. The number of signature domains needed is directly related with the runtime complexity of applying our technique. We developed a function that, after a union of domains, eliminates the redundant elements in the communication object. It is represented in the Alg. 1 by the $\cup$ symbol. This function will be applied at runtime before the communication execution. The asymptotic complexity of this function is dependent on the number of signature domains stored in the communication object, whose data will be communicated. Using this function we avoid redundant communication. This is one of the main advantages of our technique with respect to the previous work. However, depending on the program, the complexity of this function can penalize the performance of the application in some cases. A future work will determine the best option between applying the function to eliminate the redundant communications, or communicating extra data on each case.

Scalability on processing elements: Analyzing Alg. 1 we observe that the time spent by the communication calculation grows linearly with the number of processes. This has been verified in the experimental study in Sect. 7.4. This trend also appears in other previous distributed-memory approaches used to derive communications code [23]. For scalability in target platforms with high orders of magnitude of processing elements, these techniques should be combined with other ones, such as hierarchical groups of processes, or the detection and application of specific techniques for application patterns.

Table 1 Input data sizes (N) and time loop iterations (T), for benchmarks in the experimental studies.

Study 1 Bench.	**Sizes (N)**
Rotate	N $=3 * 10^7$, rot=2
Cannon	N=7680 x 7680
MG	Class D

Study 2 Bench.	**Sizes (N), iterations (T)**
Heat-1d	N = 2000000, T = 6000
Heat-2d	N = 8000 x 8000, T = 500
Heat-3d	N = 500x500x500, T = 100

7 Experimental Study

We performed an experimental study to validate our approach, to verify the efficiency of the resulting codes, and to study the potential overheads introduced by our runtime calculation. The section is divided into: (1) the experiments design details, (2) a study of several study cases, comparing our proposal with optimized MPI reference programs in terms of performance and code complexity; and (3) a breakdown of the performance measures of our codes in computation, communication calculation, and communication execution times.

7.1 Design and Setup of the Experimental Study

The experiments were executed in two platforms. The first one (CETA) is a hybrid cluster that belongs to CETA-CIEMAT[2] and the Spanish government. The cluster nodes are connected by Infiniband, and they have two Intel Xeon 5520 CPUs at 2.27 GHz, with 4 cores each. Using 8 nodes of the cluster, we exploit up to 64 computational units. The second platform is a pure shared-memory machine (Atlas), a Dell PowerEdge R815 server, with 4 AMD Opteron 6376 processors at 2.3 GHz, with 16 cores each, and 64 cores in total. We compiled the codes with the GCC v6.2 compiler, using the optimization flag *-O3*. As MPI implementation, we use *mpich3* v3.1.3, with device *ch*4, that also improves communication performance on shared memory.

We use static mapping, launching one MPI process for each processing element. The main contribution of this paper is the technique to calculate automatically the data communications needed when an application with periodic array accesses is executed on a distributed-memory system. Thus, in this paper we use only MPI processes for a fair comparison, without exploiting threads programming. We focus on studying only the potential overheads introduced by this calculation, and the benefits obtained due to the aggregated communications. The foundation of how OpenMP threads can be efficiently composed inside MPI processes when using Hitmap was presented in [21].

We executed the programs in CETA and Atlas with number of processes P = 1, 4, 8, 16, 32, and 64. We performed ten executions per each test, taking the average time. We selected big enough input data sizes to produce a minimum computational load that remains significant when computation is distributed across 64 computational units. The input data sizes (N) and time

[2] Extremadura Research Center for Advanced Technologies, Spain.

Table 2 Study 1: Performance (in seconds) for the illustrative example, Cannon's algorithm, and the MG real-world application. Comparison of MPI references and Trasgo generated codes. Cannon's algorithm requires a number of processes with a perfect square root.

	Rotate		**Cannon**		**MG**	
Machine	**MPI**	**Trasgo**	**MPI**	**Trasgo**	**MPI**	**Trasgo**
Atlas-4	1.00	1.26	148.42	146.13	512.05	519.05
Atlas-8	0.66	0.84	–	–	364.47	367.80
Atlas-16	0.35	0.43	42.43	41.69	208.59	214.40
Atlas-32	0.24	0.32	–	–	135.44	138.46
Atlas-64	0.09	0.11	11.95	10.66	105.28	100.16
CETA-4	0.94	1.07	101.16	106.20	–	–
CETA-8	0.50	0.57	–	–	–	–
CETA-16	0.27	0.31	29.89	28.60	210.92	231.83
CETA-32	0.14	0.16	–	–	194.14	229.99
CETA-64	0.08	0.08	10.68	12.80	151.66	197.64

loop iterations (T), for the different benchmarks in the experimental studies are presented in Tab. 1. All the codes use a data partition policy that splits the input data structure in as many 1D, 2D, or 3D blocks as number of processes.

7.2 Study 1: Performance comparison with MPI Reference Codes

In this section we present a performance study using: (1) The simple illustrative example presented in Sect. 3; (2) The well-known Cannon's distributed-memory parallel algorithm for matrix multiplication [8], which is specially devised for distributed-memory systems in order to minimize the memory footprint; and (3) The real-world application MG of the NAS Benchamrks, implementing a multi-grid v-cycle method for a 3D-stencil computation [1].

In this study we perform an end-to-end time measure including all program stages where Trasgo could introduce overheads. Table 2 shows the performance obtained when we compare Trasgo generated codes with reference MPI versions. Our programs for the illustrative example and Cannon's algorithm scale similarly to the optimized MPI codes.

The NAS MG benchmark requires further discussion. The distribution of the data structures is key to execute this computationally-intensive application. The MG data structures for the D input class (defined by NAS benchmarks) need to be distributed at least among 16 processes in order to fit in the local memories of the nodes of our distributed-memory machine, CETA. Thus, we only present the results for 16, 32 and 64 processes on Tab. 2. The MPI reference code of the NAS MG benchmark contains a manual optimization to communicate data across different levels of the v-cycle. This optimization cannot be directly derived from the access-expression analysis of the SPMD blocks that traverse the multi-grid during the v-cycle. Our implementation issues the extra communication stage that this optimization eliminates, incurring thus in a performance loss up to 30% in our measures.

In summary, our technique allows the automatic calculation of communication stages for codes with affine expressions with periodic boundary conditions at runtime efficiently. As we stated above for MG, the drawback of this kind

Table 3 Comparison of development effort measures for three case studies.

		KDSI	**McCabe's C.C.**	**Halstead D.E.**
Rotate	Trasgo	24	6	74K
	C+MPI	62	21	1 890K
	Reduction	61.29%	71.43%	96.08%
Cannon's MM	Trasgo	57	4	19K
	C+MPI	175	4	122K
	Reduction	67.43%	0.00%	84.43%
NAS MG	Trasgo	772	72	19 477K
	C+MPI	1389	210	29 568K
	Reduction	44.42%	65.71%	34.13%

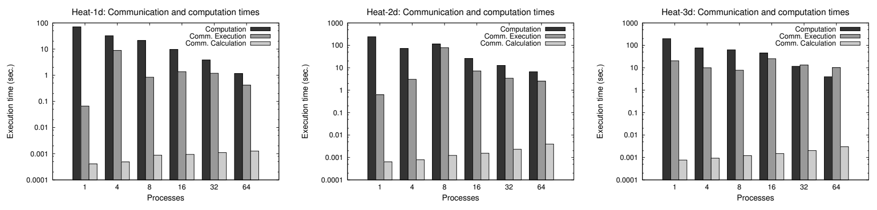


Fig. 8 Computation, communication calculation, and communication execution times in seconds for the Heat examples on the distributed-memory machine (log scale). using the problem sizes of Tab. 1.

of approaches is that these automatic calculations do not generate certain communication optimizations across SPMD blocks that could positively impact performance. An open question is whether these particular optimizations could be applied after the use of this kind of techniques.

7.3 Study 2: Ease of programming

Our technique avoids to the programmer the management of the communication and/or data partition codes. This leads to a reduction on the parallel programming complexity. Table 3 shows, for our study cases, several complexity and development effort metrics, including KDSI metric used in the COCOMO model [3] (number of lines), McCabes cyclomatic complexity [18], and Halstead development effort [14]. These metrics are used to compare the potential programming effort needed when using the different alternatives considered. We observe that the development effort needed is highly reduced when using our approach. As can be seen in Tab. 3, the reductions for the different metrics used range from 44% to 96% in all cases, except in the McCabe complexity for the Cannon's matrix multiplication, where this measure is extremely low in both codes.

7.4 Study 3: Relative cost of calculating communications

Our technique to calculate the communication patterns is performed at runtime. In this section we show an experimental study where we focus on the cost

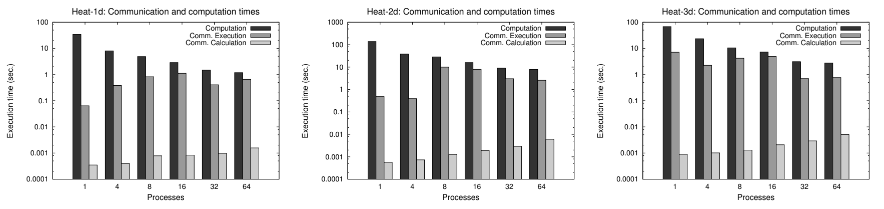


Fig. 9 Computation, communication calculation, and communication execution times in seconds for the Heat examples on the shared-memory machine (log scale) using the problem sizes of Tab. 1.

of our calculation and synchronization times with respect to the main computation times. The experiment was executed in two different architectures, the distributed- and the shared-memory machines, *CETA* and *Atlas*.

Figure 8 and 9 show the measures of the computation and the communication times, also separating in communication calculation and communication execution times. We show results for the periodic versions of the Heat-1d, Heat-2d and, Heat-3d benchmarks [5], for different number of MPI processes launched (notice the logarithmic scale in the plots). We see that the computation time decreases when the number of processes increases, except in one situation (Heat-2D with 8 processes in the distributed-memory system). This phenomenon is not related to the data communication that is the focus of this study. We also observe that the time spent by our technique in the runtime calculation of the communication patterns increases with the number of processes, as expected. However, these times can be consider negligible, as they are several orders of magnitude smaller than the computation and the communication execution times.

In summary, our technique automatically and efficiently calculates at runtime the communication patterns needed in a distributed-memory parallel program with periodic access expressions, allowing the selection at runtime of the partition policies, and the choice of the proper tile sizes for the current execution platform.

8 Conclusion

This paper describes a technique that calculates at runtime exact aggregated coarse-grained distributed-memory communications, for algorithms with affine expressions with periodic boundary conditions. It is based on: (1) calculating at runtime different footprints through cutting-and-pasting methods in terms of the mapping functions chosen and, (2) intersecting at runtime the remote and local footprints. Performance results for six cases of study, including a real-world benchmark, indicate that using our technique, we obtain similar efficiency to optimized MPI codes, while the development effort is reduced. Future work includes the applicability of the proposed technique in current polyhedral model frameworks.

Acknowledgements This research has been partially supported by MICINN (Spain) and ERDF program of the European Union: HomProg-HetSys project (TIN2014-58876-P), CAPAP-H6 Network (TIN2016-81840-REDT), 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. Bailey, D., Harris, T., Saphir, W., van der Winjgaart, R., Woo, A., Yarrow, M.: The NAS Parallel Benchmarks 2.0. Report RNR-95-020, NASA Advanced Supercomputing (NAS) Division (1995)
2. Bikshandi, G., Guo, J., Hoeflinger, D., Almasi, G., Fraguela, B.B., Garzarn, M.J., Padua, D., von Praun, C.: Programming for parallelism and locality with hierarchically tiled arrays. In: Proc. of the ACM SIGPLAN PPoPP, pp. 48–57. ACM, New York, New York, USA (2006)
3. Boehm, B.W., et al.: Software engineering economics, vol. 197. Prentice-hall Englewood Cliffs (NJ) (1981)
4. Bondhugula, U.: Compiling affine loop nests for distributed-memory parallel architectures. In: Proc. SC'2014. ACM, Denver, CO, USA (2013)
5. Bondhugula, U., Bandishti, V., Cohen, A., Potron, G., Vasilache, N.: Tiling and optimizing time-iterated computations on periodic domains. In: Proceedings of the 23rd international conference on Parallel architectures and compilation, pp. 39–50. ACM (2014)
6. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral program optimization system. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2008)
7. Bueno, J., Martorell, X., Badia, R., Ayguadé, E., Labarta, J.: Implementing OmpSs support for regions of data in architectures with multiple address spaces. In: Proc. ICS'13, pp. 359–368. ACM (2013)
8. Cannon, L.: A cellular computer to implement the Kalman filter algorithm. Doctoral dissertation, Montana State University Bozeman (1969)
9. 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)
10. Chatarasi, P., Shirako, J., Sarkar, V.: Polyhedral optimizations of explicitly parallel programs. In: 2015 International Conference on Parallel Architecture and Compilation (PACT), pp. 213–226. IEEE (2015)
11. Claßen, M., Griebl, M.: Automatic code generation for distributed memory architectures in the polytope model. In: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, pp. 7–pp. IEEE (2006)
12. Gonzalez-Escribano, A., Llanos, D.: Trasgo: A nested-parallel programming system. The Journal of Supercomputing **58**(2), 226–234 (2011)
13. 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)
14. Halstead, M.H.: Elements of Software Science (Operating and programming systems series). Elsevier Science Inc. (1977)
15. Hiranandani, S., Kennedy, K., Tseng, C.W.: Compiling fortran d for mimd distributed-memory machines. Communications of the ACM **35**(8), 66–80 (1992)
16. Kong, M., Pouchet, L.N., Sadayappan, P., Sarkar, V.: Pipes: a language and compiler for task-based programming on distributed-memory clusters. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, p. 39. IEEE Press (2016)
17. Kwon, O., Jubair, F., Eigenmann, R., Midkiff, S.: A hybrid approach of openmp for clusters. ACM SIGPLAN Notices **47**(8), 75–84 (2012)

18. McCabe, T.J.: A complexity measure. Software Engineering, IEEE Transactions on **4**, 308–320 (1976)
19. Mehta, S., Beeraka, G., Yew, P.C.: Tile size selection revisited. ACM Transactions on Architecture and Code Optimization (TACO) **10**(4), 35 (2013)
20. Moreton, A., Gonzalez-Escribano, A., Llanos, D.R.: A runtime analysis for communication calculation. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO) (poster, 2017)
21. Moreton-Fernandez, A., Gonzalez-Escribano, A., Llanos, D.: Exploiting distributed and shared memory hierarchies with Hitmap. In: Proc. HPCS'2014, pp. 278–286. Bologna (Italy) (2014)
22. 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)
23. Moreton-Fernandez, A., Gonzalez-Escribano, A., Llanos, D.R.: On the run-time cost of distributed-memory communications generated using the polyhedral model. In: High Performance Computing & Simulation (HPCS), 2015 International Conference on, pp. 151–159. IEEE (2015)
24. Planas, J., Badia, R., Labarta, E.A.J.: Hierarchical task-based programming with StarSs. IJHPCA **23**(3), 1145–1154 (2009)
25. Randall, D.A., Ringler, T.D., Heikes, R.P., Jones, P., Baumgardner, J.: Climate modeling with spherical geodesic grids. Computing in Science and Engineering **4**(5), 32–41 (2002)
26. Sanz, A., Asenjo, R., López, J., Larrosa, R., Navarro, A., Litvinov, V., Choi, S.E., Chamberlain, B.: Global data re-allocation via communication aggregation in chapel. In: Proc. SBAC-PAD'2012. IEEE (2012)
27. Sharma, A., Smith, D., Ferguson, M., Koehler, J., Barua, R.: Affine loop optimization based on modulo unrolling in chapel. In: Proc. PGAS'2014. ACM, Eugene, OR USA (2014)
28. Stepanov, A., Lee, M.: The Standard Template Library. Tech. Rep. 95-11(R.1), HP Laboratories (1995)
29. Upadrasta, R., Cohen, A.: Sub-polyhedral scheduling using (unit-) two-variable-per-inequality polyhedra. ACM SIGPLAN Notices **48**(1), 483–496 (2013)
30. Verdoolaege, S.: Isl: An integer set library for the polyhedral model. In: Mathematical Software–ICMS 2010, pp. 299–302. Springer (2010)
31. Yuki, T., Rajopadhye, S.: Parametrically tiled distributed memory parallelization of polyhedral programs. Tech. Rep. CS13-105, Colorado State University (2013)