A Technique to Automatically Determine Ad-hoc Communication Patterns at Runtime

Ana Moreton-Fernandez, Arturo Gonzalez-Escribano, and Diego R. Llanos

Departamento de Informática, Edif. Tecn. de la Información, Universidad de Valladolid, Campus Miguel Delibes, 47011 Valladolid, Spain.

Abstract

Current High Performance Computing (HPC) systems are typically built as interconnected clusters of shared-memory multicore computers. Several techniques to automatically generate parallel programs from high-level parallel languages or sequential codes have been proposed. To properly exploit the scalability of HPC clusters, these techniques should take into account the combination of data communication across distributed memory, and the exploitation of shared-memory models.

In this paper, we present a new communication calculation technique to be applied across different SPMD (Single Program Multiple Data) code blocks, containing several uniform data access expressions. We have implemented this technique in Trasgo, a programming model and compilation framework that transforms parallel programs from a high-level parallel specification that deals with parallelism in a unified, abstract, and portable way.

The proposed technique computes at runtime exact coarse-grained communications for distributed message-passing processes. Applying this technique at runtime has the advantage of being independent of compile-time decisions, such as the tile size chosen for each process. Our approach allows the automatic generation of pre-compiled multi-level parallel routines, libraries, or programs that can adapt their communication, synchronization, and optimization structures to the target system, even when computing nodes have different capabilities.

Our experimental results show that, despite our runtime calculation, our approach can automatically produce efficient programs compared with MPI reference codes, and with codes generated with auto-parallelizing compilers.

Keywords: SPMD models, Distributed communications, Parallel programming, Trasgo

1. Introduction

Parallel machines are becoming more heterogeneous, mixing devices with different capabilities in the context of hybrid clusters, with hierarchical shared- and distributed-memory levels. The focus on parallel applications is shifting to more diverse and complex solutions, exploiting

☆

Email address: {ana,arturo,diego}@infor.uva.es. (Ana Moreton-Fernandez, Arturo Gonzalez-Escribano, and Diego R. Llanos)

several levels of parallelism, with different strategies of parallelization. Programming in this kind of environment is challenging. Many approaches have been proposed over the last few years following two different paths: Programming using parallel programming models, or automatically generating parallel code from sequential programs.

Using current parallel programming models (e.g. MPI, OpenMP, Intel TBBs, Cilk, and PGAS languages such as Chapel, X10, or UPC), the application programmer still faces many important decisions not related with the parallel algorithms, but with implementation issues that are key for obtaining efficient programs. For example, decisions about partition and locality vs. synchronization/communication costs; grain selection and tiling; proper parallelization strategies for each grain level; or mapping, layout, and scheduling details.

Automatic code generation techniques can be used to automatically transform high-level parallel expressions or sequential codes to parallel programs that take into account many of these issues. Most of these techniques were designed to be applied at compile time [1, 2, 3, 4]. For example, the work presented in [2] proposes a technique that, from a sequential code, generates a low-level parallel code for distributed-memory systems using the Message Passing Interface (MPI) library. This technique improves previous schemes because the code it generates is parametric in the number of processes and problem sizes, reducing the communicated volume of data. However, it needs to fix a single tile size at compile time, even if the distributed system has nodes with different capabilities.

In this paper, we present a new communication calculation technique to be applied across different SPMD (Single Program Multiple Data) blocks of code, that contain several different data accesses expressions to the same data structure, whose indexes are calculate with uniform affine expressions in the indexes selectors. We consider as uniform affine expressions, those expressions that derive in accesses to a multi-dimensional paralelotope of the data structure domain. For two dimensions, this means rectangular shapes. The technique supports codes with several data accesses to the same data structure. Thus, the resulting domain accessed by a code block is a compound of paralelotope shapes, that can be non-convex.

We have implemented this technique in Trasgo [5], a programming model and compilation framework to generate parallel programs from a high-level parallel specification that deals with parallelism in a unified, abstract, and portable way. The proposed technique computes at runtime exact coarse-grained communications between two consecutive parallel blocks for distributed message-passing processes.

The main contributions of this paper are the following:

- A technique to determine at runtime communications across SPMD blocks for distributed-memory systems. The communications calculated are:
  - *Coarse-grained* in the sense that communication calculation across two parallel SPMD blocks is done once for the whole index space mapped to a process at runtime, independently of the number or sizes of tiles generated inside the process. This enables different tile sizes to be used in the same computation at the same hierarchical level, an important feature in achieving a good performance on heterogeneous systems that include machines with different architectures [6].
  - *Exact* because they are optimal in terms of communication volume. Our runtime calculation skips all the duplicated data elements in the communication. Thus, no data is communicated twice, and no unneeded or control data is communicated across any two distributed processes.

```
** Illustrative example
Inputs:
   a: 1st stencil parameter
   b: 2nd stencil parameter
   M[n][n]: Matrix with initial values
   limit: Number of iterations
Output:
   M[n][n] : Matrix with result values
Temporal variables:
   M_temp[n][n]: Auxiliar matrix.

** Time loop
Do iter = 1 to limit

   ** First SPMD block: Update M_temp
   Do i = a to n-b
      Do j = a to n-b
         M_temp[i][j] = M[i][j]

   ** Second SPMD block: Compute stencil operation
   Do i = a to n-b
      Do j = a to n-b
         M[i][j] = (M_temp[i-a][j] + M_temp[i+b][j] + M_temp[i][j-a] + M_temp[i][j+b])/4;
```

Figure 1: Sequential algorithm for the illustrative example.

- The implementation of the proposed communication calculation technique in a parallel programming framework, whose input language allows the programmer to reason in terms of logical processes, without facing decisions about granularity, thread management, or interprocess communication.
- An experimental study to evaluate our runtime proposal comparing it with a compile-time state-of-the-art tool that generates communication codes, and with pure-MPI references codes.

Our technique postpones to runtime part of the analysis and decisions needed to transform program abstractions to actual processes. Thus, the programs can adapt their behavior at runtime, dealing with different partitions, granularity, data-distribution, memory hierarchies, tile sizes, or synchronization and communication structures.

We start our discussion with an illustrative example based on a stencil computation in Sect. 2, showing the transformation techniques presented for the rest of the paper. Section 3 describes the Trasgo model and its tools. Section 4 introduces the new techniques applied in Trasgo. To show the applicability and efficiency of the approach, we include several experimental studies in Sect. 5, comparing performance on distributed- and shared-memory platforms with MPI reference codes, and codes generated with auto-parallelizing compilers. The results show that our approach can automatically produce efficient programs despite the overhead of the calculation performed at runtime. Section 6 discusses some related work. Section 7 presents the conclusion, and future work.

2. Illustrative example and Overview

This section presents an illustrative example that serves as a quick overview of the techniques presented in this paper. The example is a modification of a Jacobi PDE solver for Poisson's

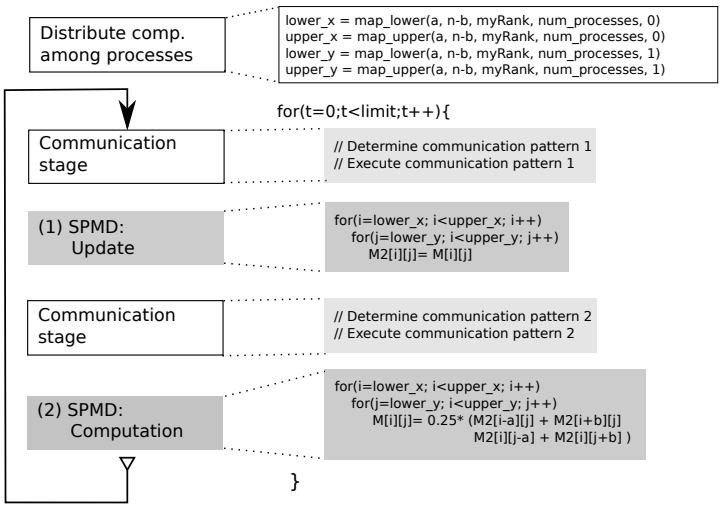

Figure 2: Scheme of the parallel algorithm following an SPMD model for the illustrative example (left), and code excerpts for the main blocks (right). The parameter a determines the halo sizes on the top and left sides, and the b parameter determines the same on the bottom and right sides.

equation to compute heat transfer in a discretized two-dimensional surface. This is a simple data-parallel example, that aims to introduce the readers to the basis of the approach. This example can be used to show basic concepts of MPI synchronization (see e.g. [7]). It contains clear computation and communication stages with uniform data access expressions. The sequential algorithm is shown in Fig. 1. In our example we also introduce two integer parameters a, b that are used in the access expressions to select at runtime the distance to the positions considered *neighbors* in terms of the stencil operation carried out for that particular invocation of the computation.

2.1. Programming with an SPMD model

We first present an overview of the typical approach used to program the illustrative example following an SPMD model. In the sequential algorithm (recall Fig. 1), we can distinguish two different blocks of code inside the time loop that can be parallelized independently without violating any data dependence (transforming them into SPMD blocks). Figure 2 (left) shows a distributed parallel programming scheme of the illustrative example following an SPMD model. To program this algorithm in parallel, the only data dependences that must be taken into account are those produced between the SPMD blocks. For this reason, a communication/synchronization stage is inserted between them.

When we execute this parallel example algorithm in shared-memory systems, a synchronization stage is enough to avoid data dependences between the two parallel structures. However, in distributed memory systems, the data written by a process in the first SPMD block should be sent to other processes that need these data to execute the second SPMD block. Notice that a synchronization is implicit in this communication.

Programming for distributed-memory systems has the challenge for the programmer of dealing with the data and computation distribution between different processes. In our example (see Fig. 2, right), we have distributed the computation using the mapping functions *map_lower()* and

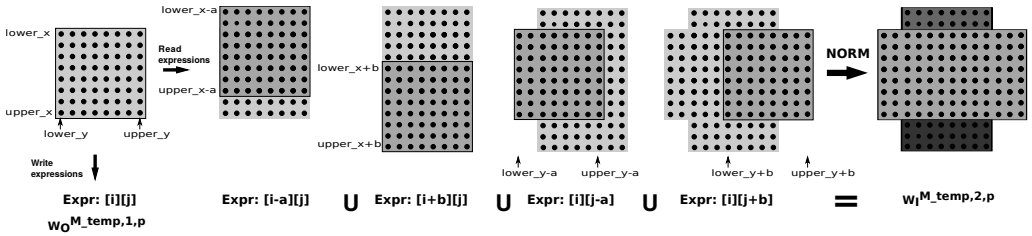

Figure 3: Using the read and write data-access expressions inside the parallel structure of the illustrative example to calculate the working input and output index sets (W_I^2,W_O^1) for M_temp for a generic process. This example assumes positive parameters $a, b \geq 0$. The *Norm* operation on the last stage reduces the number of boxes used to represent the union of domains. The drawings consider the particular case of $a = 2, b = 3$, for a domain with 8×8 contiguous indexes.

map_upper(). These mapping functions receive the first and last iteration of the loop to be distributed, the identifier of the current process, and the total number of processes. These functions return the loop limits corresponding to the chunk of iterations that should be executed by the current process, avoiding overlappings with other processes.

Each process is ready to perform the computation as soon as it has the corresponding data in its local memory. These data include not only the data in the positions that match the loop iterations, but also the data that correspond to their *halo*, which may be owned by other processes. A communication stage before each SPMD block should retrieve the corresponding halos, thus ensuring that each process has a local copy of all the needed data. In our particular example, before the execution of the first SPMD block, a communication stage is not necessary as each process already has all the data it needs. However, in the second SPMD block, each process needs data that have been updated by other processes (data in the halo). The communication phase in this case depends on execution parameters, such as the matrix size, the tile size, the number of processes, the partition policy (the way in which the data were partitioned among the processes), and the values of a and b parameters which define the halo sizes.

The technique presented in this paper determines automatically at runtime the communication patterns needed between two consecutive parallel structures, taking into account these parameters, regardless of their availability at compile or execution time, regardless of the application of other sequential or tiling optimization techniques inside each process.

2.2. Overview of the communication determination technique

We present here an overview of the proposed technique to determine automatically the communication patterns among different SPMD blocks. As we have seen in the previous description, we usually need a communication stage between two consecutive SPMD blocks to ensure a correct execution. In our example, we need to communicate some data written in the first SPMD block by each process to other processes that read these data in the second SPMD block. Our technique consists of two steps:

1. In order to determine the data read and written in each parallel structure, for each SPMD block in the program, we generate at compile time a parametrized function that, at runtime, returns the set of data being read or written for a given process identifier p.

   For the illustrative example, we show an example of both sets of indexes returned by these generated functions in Fig. 3. We name the sets of indexes of the matrix M read and written

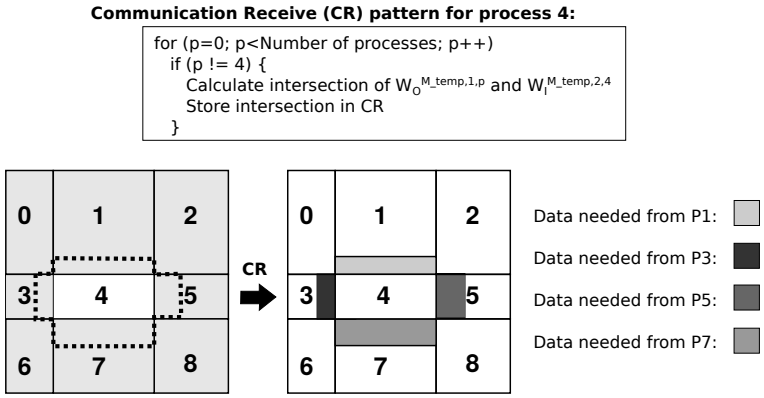

Figure 4: Calculation of Communication Receive (C_R) pattern between the two parallel structures of the illustrative example. Example for a number of processes $P = 9$, a process identifier 4, the previously generated functions $W_I^{M\_temp,2,*}$ and $W_O^{M\_temp,1,*}$, a mapping/partition function that returns irregular rectangular blocks, and particular values for symbolic parameters $a = 2, b = 3$.

by the k-th SPMD block in the code, at a given processor p as follows: *Input Working Set Indexes* $W_I^{M,k,p}$, and *Output Working Set Indexes* $W_O^{M,k,p}$. These functions are generated at compile time using the data-access expressions found in the input code. In Fig. 3, we see how the set $W_I^{\text{M\_temp},2,p}$ is calculated by applying the uniform access expressions found in the code to the calculated loop limits (lower_x, lower_y, upper_x, upper_y). The set of indexes is normalized to be represented with a set of rectangular shapes.

2. In the second step, we apply an algorithm at runtime to determine the communication patterns and to store a compact description of them in an object. To calculate the data that have to be received by a local process from a remote process p, the algorithm intersects the set of indexes read by the local process in the second SPMD block (that is, $W_I^{M,k+1,local}$) with the set of indexes written by the process p in the first SPMD block ($W_O^{M,k,p}$). Figure 4 shows a visual representation of our runtime algorithm for the illustrative example. The example shows the calculation of the Communication Receive (CR) pattern for process 4. On the left, we can see the M_temp matrix distributed among 9 processes with an arbitrary irregular partition policy, and the data set (dotted lines) to be read by process 4 in the following SPMD block. On the right, we see the data that should be received by process 4 from different remote processes. The patterns are calculated using the proper intersections of the data owned by different processes.

   To calculate the data to be sent by a local process to a process p, the algorithm performs the opposite intersection, between the set of indexes written by the local process in the first SPMD block (that is, $W_O^{M,k,local}$) and the set of indexes read by the p process in the second SPMD block ($W_I^{M,k+1,p}$). An empty intersection indicates that no send (or receive) operation is needed for that particular process p.

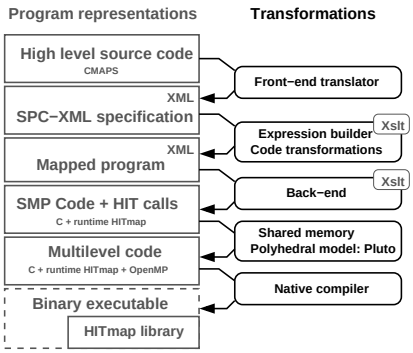

Figure 5: Structure of the Trasgo transformation framework.

After applying this technique for every process and every array we can apply the determined send and receive patterns to perform the actual communications.

Our technique requires that any symbolic parameter must have the same value on every process. Thus, the set of indexes accessed by a remote process p can be calculated by any process, with no inter-process communication. A deeper discussion about the constraints, the features used to reduce the complexity, and a formal definition of our technique is presented in Sect. 4.

3. The Trasgo Model

The previous section briefly describes our proposed technique. In this section, we review the Trasgo parallel programming and execution model, where our technique has been implemented.

The Trasgo model [5] proposes the use of an explicitly parallel, but high-level and structured representation of parallel algorithms. It uses restricted synchronization at the higher level, generating more efficient and less synchronized parallel structures at the low level. The original model is based on the SP (Series-Parallel) process model [8] and data-distribution algebras, providing clear and well-defined semantics [9]. The model is free of race conditions, unexpected dead-locks, or stochastic behaviors. The high-level code uses a global view approach in hierarchical decompositions. The semantics provide clear synchronization points and hierarchical global states that simplify testing and debugging.

3.1. Overview of the code transformation framework

Figure 5 shows the structure of the Trasgo transformation framework. The left column shows the program representations, and the right columns the transformation layers.

A front-end translates the input language (in our case CMAPS [10]) to an XML internal representation. The main part of the transformation layer transforms the global address space into a partitioned address space, building the functions to compute communications across virtual processes. The transformed code is rewritten by a back-end that generates C code with calls to the Hitmap run-time library (see Sect. 3.3). The resulting computation code, generated for the local distributed process, can finally be optimized through polyhedral tools to generate optimized parallel code for the shared memory level using OpenMP. Pluto compiler [11] has been shown

```
1  /* Second Sequential function */
2  void updateCell( in double up, in double down, in double left, in double right,
3   out double result) {
4   *result = ( up + down + left + right ) / 4 ;
5  }
6
7  /* First Sequential function  */
8  void updateData( in double Data, out double result) {
9   *result = Data ;
10 }
11
12 /* Parallel stencil code with parametric dependences */
13 coordination void caseA( inout tile double M[][], in int limit,
14                          in int a, in int b, in layout partition_policy) {
15  double M_temp[][];
16  /* Distribute arrays*/
17  ArrayMap( M, partition_policy );
18  ArrayMap( M_temp, partition_policy );
19
20  /* Time loop */
21  loop( t in [1:limit]  ) {
22   /* First SPMD block: Update M_temp */
23   parallel ( i,j in M) {
24    do: updateData( M[i ][ j ], M_temp[ i ][ j ] );
25   }
26   /* Second SPMD block: Compute stencil operation */
27   parallel ( i,j in M ) {
28    do: updateCell( M_temp[ i-a ][ j ], M_temp[ i+b ][ j ],
29                    M_temp[ i ][ j-a ], M_temp[ i ][ j+b ],
30                    M[ i ][ j ] );
31   }
32  }
33 }
```

Figure 6: CMAPS code for the illustrative example.

to be a good tool to auto parallelize sequential codes for shared-memory systems. As a proof of concept, we use Pluto on the codes obtained after applying the mapping policy. Thus, we make it optimize the local computation inside each MPI process, in order to efficiently exploit internally the multi-core processor of each node. The methodology used to integrate Pluto with the Hitmap toolchain was described in [12]. The final code is compiled with a native C compiler.

Figure 6 shows the illustrative example coded in CMAPS, the current Trasgo input language. First, we define the sequential functions to apply to each element, specifying the output or input role of the parameters (lines 2,8). The parallel function is defined using the *coordination* modifier, and also specifying the role of the parameters (line 13). In its body, the function *ArrayMap()* distributively allocates both arrays *M_temp* and *M* in terms of the results of a mapping/partition policy named *partition_policy*, which is also a parameter (line 17-18). After the distribution, the code updates and computes each element of *M_temp* in parallel using the sequential functions previously defined (lines 23, 28). The *parallel* structure in CMAPS maps the computation indicated in the *do:* code to each indexes pair in the domain specified in the clause inside the parallel brackets.

3.2. Notations and definitions

In this section, we present definitions used in the rest of the paper. In this work, we focus on arrays with regular dense and strided domains.

Signature is a triplet of integer numbers $S\langle b, e, s\rangle$ (meaning *begin*, *end*, and *stride*). The set of indexes expressed by a signature is $S\langle b, e, s\rangle = \{b \leq i \leq e : (i - b) \mod s = 0\}$.

Domain is a subspace of $\mathbb{Z}^n$. Rectangular n-dimensional parallelotope domains, dense or strided, can be represented by a tuple of *n Signatures*. Let us consider an n-dimensional domain $D_n\langle s_0, s_1, ..., s_{n-1}\rangle \in \mathbb{Z}^n$, where $s_0 \in S^*$, .., $s_{n-1} \in S^*$ are the signatures whose Cartesian product defines the domain. This kind of structures only represent rectangular shapes.

Working set is a generic set of indexes. We represent generic sets as unions of signature domains, $W^M = \cup_{i=0}^{q} d_i : q \in \theta(N), d_i \in D_n$.

Tile is an object that associates data elements of a given *type* to index elements of a domain. The domain of a tile is denoted as $D(T)$, $T : D \rightarrow type$.

Logical process is a tuple $P\langle f, W_I^M, W_O^M\rangle$, where f is a function or subprogram, W_I^M is the working set that P receives as input (the data indexes of M that are read), and W_O^M is the set used as output (the data indexes of M that are written). Logical processes may be composed in sequence, or in parallel. A sequential composition $P_1 \triangleright P_2$ indicates that f_1 is executed before f_2, and that data modifications introduced in the output tiles of P_1 are propagated in the corresponding input tiles of P_2. Sequential composition is associative but not commutative. A parallel composition $P_1 \circ P_2$ indicates that f_1 and f_2 can be executed in parallel. Parallel composition is associative and commutative.

Wave-front composition $(P_1 \bullet (W_f) P_2)$, is a parallel composition with explicitly added order dependences for arbitrary tiles, across processes. If there is overlapping of the shapes of W_f with the input working set of P_2 and the output working set of P_1, the function f_2 cannot be started until f_1 has finished. The data represented by the overlapping shapes should be propagated. This does not allow generic data-flow compositions to be expressed, with transitive cycles that could lead to dead-locks. This composition operation allows parallel structures, such as wave-front computations, and macropipelines to be expressed when used inside a loop.

Virtual Topology $V(N, R)$ is a graph where the vertices N represent virtual processes, associated with computational resources (groups of processors), and the edges R represent neighbor relations.

Layout $L : D \rightarrow V$ is a function that maps domain subspaces (indexes of tiles or logical processes) to the virtual processes in a virtual topology.

3.3. Hitmap library

Hitmap [13] is a library for the management and run-time mapping of hierarchically tiled arrays used in Trasgo. It is based on an SPMD model, and on the message-passing paradigm. Hitmap has three main functionality modules: (a) Domain and tile management; (b) Mapping modules; and (c) Communication patterns. Hitmap defines objects to declare and manipulate index sets as multidimensional parallelotopes with optional stride, or as sparse sets. Hitmap defines a plug-in system to include new mapping modules: Virtual topology constructors and mapping functions named *layouts*. The modules generate objects that can be queried at runtime to obtain information about the result of the mapping for the local, or any remote process. Finally, it contains functionalities to build reusable communication patterns for tiles, or subtiles, across

virtual processes. These functions internally use the MPI standard, exploiting efficient techniques like derived data types, and asynchronous communications.

Several new functionalities have been added to Hitmap to support the new techniques presented in Sect. 4. To represent generic domains and working-set indexes in Hitmap, we have created a new data structure, called *Domain*. We have implemented functions for efficient domain set operations (intersection $\cap$, union $\cup$, and subtraction $\setminus$) on parallelotope shape structures. A *Norm* function is implemented to transform any domain, represented as a set of signature-based parallelotope shapes, in an union of a set of signature-based parallelotope shapes with empty pair-to-pair intersections. The resulting normalized domain is a set of parallelotope areas with optional stride inside, which can be directly translated to MPI derived data-types for efficient marshalling/unmarshalling operations. The runtime asymptotic complexity of this function is directly related to the amount of shapes that compose the input set. Also, the actual bound of the q value for a given application (reminding the representation of the working sets) is directly related with the runtime complexity of applying our technique. See an example of the use of the *Norm* function in the final step of Fig. 3. More advanced representations will be presented in a future work.

4. Implementation of the technique to determine communication patterns

This section describes in depth the new technique we propose in the context of the Trasgo framework. It allows to determine at runtime the exact aggregated communications across distributed processes, for codes with data-access expressions which are affine transformations of the data indexes used in an SPMD block. Let $i_x : x = 0 \dots n-1$ be the data indexes in an SPMD block (parallel indexes). The parallel indexes in a CMAPS code are those in the clause inside the brackets of a *parallel* structure. An affine access expression $\rho(x)$ is defined as:

$$\rho(x) = \alpha_0 \times i_0 + \alpha_1 \times i_1 + ... + \alpha_{n-1} \times i_{n-1} + \beta$$

where the coefficients α_x, β can be general expressions using constants and parameters whose values can be unknown at compile time, but are invariant in the body of the parallel structure (SPMD block). In the current prototype we only support uniform affine expressions, whose resulting index domain is a multidimensional parallelotope (hyperrectangular shapes). A uniform expression is defined as:

$$\varrho(x) = i_y + \beta$$

The proposed technique also supports the composition of several blocks that come from the application of several uniform expressions. The resulting domain can be a non-convex domain that is represented by a set of non-overlapped hyperrectangular blocks, that represent the exact subspace of the domain that is accessed. See an example on Fig. 3.

This section describes in detail the proposed technique. As shown in the illustrative example, we divide our technique into two steps.

4.1. Functions to calculate working set indexes

As previously discussed in section 2.2, we should generate functions $W_I^k(\dots)$, and $W_O^k(\dots)$, which calculate at runtime the input/output working set indexes of each data structure, at each k-th parallel structure (SPMD block). These functions are generated from the expressions found in the *do:* clauses of the CMAPS codes. For parallel structures with *wave-front* expressions, we

also generate functions to compute the Input-Flow Working Set Indexes $W_F^k(\ldots)$. The *wave-front* expressions are found in CMAPS in a specific clause that determine the *wave-front* dependences. These functions ($W_F^k(\ldots)$) are generated like other working-set index functions, but using the *wave-front* expressions as if they were read accesses to the data structure.

The generated functions have the following parameters: A process index p, a mapping function $L(p)$ to obtain the subdomain of parallel indexes mapped to p, and the symbolic parameters that appear in the data read/write accesses to the chosen data structure. The code of the function applies, to the index subdomain $L(p)$, all the affine transformations found in the read/write accesses for the data structure, inside the k-th parallel structure. Figure 7 shows the functions generated for the working sets $W_I^{\text{M\_temp},2,p}$ and $W_O^{\text{M\_temp},1,p}$ in the illustrative example (they are prefixed by `calculateWI`, and `calculateWO` in the code). The *HitLayout* objects implement the $L(p)$ methods applied at runtime. The resulting shape domain is transformed according to the code expressions found in CMAPS codes, and the union of domains is computed. The example of Fig. 7, uses specialized Hitmap functions for the case of expressions with only one parallel index on each dimension scope. Let $[\alpha_x * i_y + \beta_x]$ be an expression to access the x-th dimension of the data structure. Notice that the parallel index y does not need to be the same as the dimension that is accessed. The *hit_shapeAffine2(...)* function transforms a 2-dimensional shape to another 2-dimensional shape. For each dimension, the parameters are the identifier of the parallel index y, and the subexpressions α_x, β_x, which are literally copied from the data access expression.

In the current implementation of the Trasgo prototype, we have only taken into account uniform access expressions (According to [4], approximately 84% of the benchmarks of the Polybench [14] can be fully uniformized). The implementation of the transformation functions (such as *hit_shapeAffine2()*) is based on signature (domain) algebras. These functions simply apply the access expressions to the index-space limits at runtime to calculate the resulting shapes. The transformations can be applied independently to each shape, and each dimension. Let us consider a shape domain $L(p) = \langle s_0, s_1, \ldots, s_{n-1} \rangle$. Let $s_y\langle b, e, s \rangle$ be its signature in the dimension y. For an access expression $[\alpha_x * i_y + \beta_x]$, the signature representing the transformed working-set index domain in the x-th dimension can be calculated as $T_1(L(p), y, \alpha_x, \beta_x) = \langle b', e', s' \rangle$, with:

$$
\begin{aligned}
b' &= min(\alpha_x \times s_y.b + \beta_x, \alpha_x \times s_y.e + \beta_x) \\
e' &= max(\alpha_x \times s_y.b + \beta_x, \alpha_x \times s_y.e + \beta_x) \\
s' &= \alpha_x \times s_y.s
\end{aligned}
$$

The transformation functions are applied one by one, according with the data accesses expressions. Their resulting domains are compounded using the *hit_shapeUnion* function. This composition can result in a non-convex domain, that is normalized to eliminate the overlapped parts. The result is a set of non-overlapped rectangular shapes (see Fig. 3). These functions are used to calculate the domains, W_I^{M} and W_O^{M}, independently on each process, with no interprocess communication.

Future work includes the implementation of functions to support multi-domains, allowing expressions that involve more than one parallel index. For example, a transformation for expressions such as $[i_0 + i_1][i_0 - i_1]$ would lead to a non-rectangular, rhomboidal shape, that can be represented as a set of rectangular shape structures. Moreover, more complex representations based on octagons [15] can be considered for more general shape representations.

```
/** Calculate W_I for M_temp in SPMD 2 */
HitDomain calculateWI_M_temp_2(HitRank p, HitLayout lay, int a, int b){
   HitShape _TT_mapIdx = hit_layOtherShape( lay, p );                // L(p)

   HitDomain _TT_inWS_matrix = hit_shapeUnion(
     // 2D Affine transformations ( domain, index_x, alpha_0, beta_0, index_y, alpha_1, beta_1)
     hit_shapeAffine2( _TT_mapIdx, 0, 1, -a, 1, 1, 0  ),
     hit_shapeAffine2( _TT_mapIdx, 0, 1, +b, 1, 1, 0  ),
     hit_shapeAffine2( _TT_mapIdx, 0, 1, 0, 1, 1, -a  ),
     hit_shapeAffine2( _TT_mapIdx, 0, 1, 0, 1, 1, +b  ) );

   return _TT_inWS_matrix
}

/** Calculate W_O for M_temp in SPMD 1 */
HitDomain calculateWO_M_temp_1(HitRank p, HitLayout lay){
   HitShape _TT_mapIdx = hit_layOtherShape( lay, p );                // L(p)
   HitDomain _TT_outWS_matrix = hit_shapeTodomain( _TT_mapIdx );     // No transformations
   return _TT_outWS_matrix;
}

/** Calculate Communication Pattern: Between SPMD 1 and 2 */
HitPattern calculateCommunications_M_temp_1_2( HitTile _TT_Tile1, HitLayout _TT_lay1,
                                               int a, int b){

   HitRank local = hit_laySelfRanks ( _TT_lay1 );                    // myRank
   HitPattern _TT_patternComm=HIT_PATTERN_NULL;                      // CR,CS = Empty
   HitDomain WO_L = calculateWO_M_temp_1( local , _TT_lay1);             // W_O(myRank,L)
   HitDomain WI_L = calculateWI_M_temp_2( local , _TT_lay1);             // W_I(myRank,L)

   for (_TT_1 = 0; _TT_1 < hit_layNumActives(_TT_lay1) ; _TT_1++ ){  // For p = 0..P
      if( hit_layToActiveRanks(_TT_lay1, _TT_1) != local ){          // If p != myRank
         // Send tuple
         HitDomain WI_P = calculateWI_M_temp_2( _TT_1 , _TT_lay1, a, b); // W_I(p,L,a,b)
         HitDomain _TT_aux = hit_domainIntersect(WO_L, WI_P);        // Intersection
         _TT_aux = hit_NormalizeDomain (_TT_aux);                    // Normalize
         hit_patternAdd(&_TT_patternComm,                            // Add CS tuple
            hit_comSendSelect(_TT_lay1, hit_layToActiveRanks(_TT_1), &_TT_Tile1, _TT_aux,
                              HIT_COM_TILECOORDS, HIT_DOUBLE) );
         // Receive tuple
         HitDomain WO_P = calculateWO_M_temp_1( _TT_1 , _TT_lay1);       // W_O(p,L)
         HitDomain _TT_aux = hit_domainIntersect(WI_L, WO_P);        // Intersection
         _TT_aux = hit_NormalizeDomain (_TT_aux);                    // Normalize
         hit_patternAdd(&_TT_patternComm,                            // Add CR tuple
            hit_comRecvSelect(_TT_lay1, hit_layToActiveRanks(_TT_1), &_TT_Tile1, _TT_aux,
                              HIT_COM_TILECOORDS, HIT_DOUBLE) );
      }
   }
   return _TT_patternComm;
}
```

Figure 7: Generated code for illustrative example: Communication constructor functions for the parallel structure inside the time loop. Hitmap library is used for tiling management and message passing. Communication is encapsulated on *HitPattern* objects. Reading a, b values from program arguments is skipped.

ALGORITHM 1: Model to calculate the communication pattern across parallel structures, for a given data structure, in terms of intersections of input/output working-set indexes.

Input: P : Number of processes,
myRank : Local process id,
$W_O^k(p)$: Function to compute output working-set
$W_I^{k+1}(p)$: Function to compute input working-set
Output: $\langle C_S, C_R \rangle$: Sets of communication tuples

$C_S \leftarrow \emptyset,\ C_R \leftarrow \emptyset$
for *p = 1 to P* **do**
  if $p \neq myRank$ **then**
    $C_S \leftarrow C_S\ \cup \langle p, W_O^k(myRank) \cap W_I^{k+1}(p) \rangle$
    $C_R \leftarrow C_R\ \cup \langle p, W_O^k(p) \cap W_I^{k+1}(myRank) \rangle$
  end
end

ALGORITHM 2: Model to calculate communication pattern from parallel structure k to itself, after satisfying wave-front flow dependences.

Input: P : Number of processes,
myRank : Local process id,
$W_I^k(p)$: Function to compute input working-set of k
$W_O^k(p)$: Function to compute output working-set of k
$W_F^k(p)$: Function to compute input-flow working-set of k
Output: $\langle C_S, C_R \rangle$: Sets of communication tuples

$C_S \leftarrow \emptyset,\ C_R \leftarrow \emptyset$
$W_{tmp1} \leftarrow W_I^k(myRank)\ \setminus\ W_F^k(myRank)$
for *p = 1 to P* **do**
  if $p \neq myRank$ **then**
    $W_{tmp2} \leftarrow W_I^k(p)\ \setminus\ W_F^k(p)$
    $C_S \leftarrow$
    $C_S\ \cup \langle p, W_O^k(myRank) \cap W_{tmp2} \rangle$
    $C_R \leftarrow C_R\ \cup \langle p, W_O^k(p) \cap W_{tmp1} \rangle$
  end
end

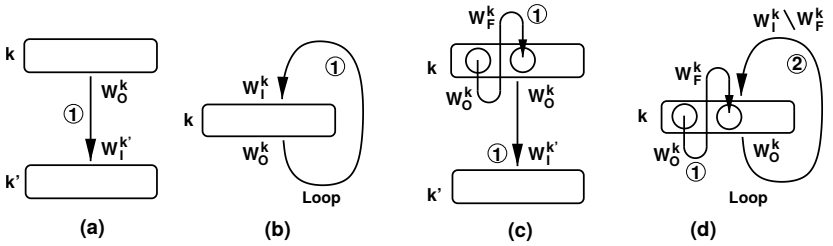

Figure 8: Working-set index functions used to tailor the communication constructor algorithms for the four possible situations. Cases (a) and (c) calculate communications across two parallel structures. Cases (b) and (d) calculate communications from one parallel structure to itself, when it is inside a loop. In cases (a) and (b), the parallel structure k does not have *wave-front* expressions. The encircled number represents the algorithm implemented by the generated function.

4.2. Determining Communications patterns

As described in the overview on Sect. 2.2, communications should be executed between parallel structures, or between a parallel structure and itself if it is inside a loop.

Communication patterns are formed by two subsets $\langle C_R, C_S \rangle$ of comm-tuples (communication tuples). C_R tuples indicate receive operations. C_S tuples indicate send operations. A comm-tuple $\langle p, W^M \rangle$ contains the index of the remote process p, and the working-set indexes W^M of the structure whose data values will be communicated.

Our solution determines the communication patterns needed across SPMD blocks combining two algorithms. Algorithms 1 and 2 are generic models that traverse at runtime the remote active $P-1$ processes intersecting output and input working-set indexes for the same data structure in order to determine the communication tuples needed between two SPMD blocks. Comm-tuples with empty sets, resulting from the intersections, are discarded and not internally stored by the Hitmap objects. For clarity, the data-structure name, the mapping functions $L(p)$ of the indexes at each parallel structure, and the specific parameters are omitted. These algorithms should be implemented on tailored functions for each communication calculation stage, adding the extra parameters needed as inputs in the calls to each specific working-set function.

We distinguish four different cases for using the working-set index functions in order to build the communication constructor functions (See Fig. 8). The cases for parallel structures without *wave-front* expressions (namely, (a) and (b)), are simpler. A function is built using Alg. 1. It is tailored to use $W_O^k(\ldots)$ for output working-set indexes. For input working-set indexes, we use $W_I^{k'}(\ldots)$ in case (a), or $W_I^k(\ldots)$ in case (b), which only has a single SPMD block inside a loop. An example of a tailored function to calculate the communication of two consecutive parallel structures in the illustrative example, is shown in Fig. 7, with the name `calculateCommunications_M_temp_1_2()`.

Our communication calculation and execution are performed just before the computation of the SPMD block. Nevertheless, if the access-expressions is not dependent on an index of an outer loop, the communication calculations can be inserted as soon as the parameters and L functions are known (even in initialization time in many cases), and the communication execution is inserted before the computation of the parallel structure. For example, in the generated code for the illustrative example (see Fig. 9), the invocation to calculate the communications is at line 2 on the main program, before the time loop iteration. Communications are calculated only once, because the a and b parameters are not modified during the computation. The execution of the

```
1   /* 1. Building comm. pattern A (1 to 2) that is invariant in the whole execution */
2   HitPattern _TT_comA = calculateCommunications_M_temp_1_2(M_temp, M_temp.Layout a, b);
3
4   /* 2. Time loop */
5   for( i=0; i<iterations; i++ ) {
6
7      /* 3. SPMD block: Loops to update M_temp */
8      ...
9
10     /* 4. Communication, execute pattern A */
11     hit_patternDo( _TT_comA );
12
13     /* 5. SPMD block: Loops to traverse the logical processes */
14     for ( _TT_i0=0; _TT_i0 < hit_tileDimCard(M,0); _TT_i0++ ) {
15        for ( _TT_i1=0; _TT_i1 < hit_tileDimCard(M,1); _TT_i1++ ) {
16           /* 5.1. Call functions with selections */
17           updateCell(
18              hit_tileElemAt(M_temp, 2, _TT_i0-a, _TT_i1),
19              hit_tileElemAt(M_temp, 2, _TT_i0+b, _TT_i1),
20              hit_tileElemAt(M_temp, 2, _TT_i0, _TT_i1-a),
21              hit_tileElemAt(M_temp, 2, _TT_i0, _TT_i1+b),
22              hit_tileElemAt(M, 2, _TT_i0, _TT_i1)
23           );
24        }
25     }
26  }
```

Figure 9: Excerpt of generated code for illustrative example: main program. Hitmap library is used for tiling management, and message passing. Communication is encapsulated on *HitPattern* objects. Reading a, b values from program arguments is skipped.

communications is at line 11, before the second parallel block. It is executed on each iteration.

When the parallel structure k has *wave-front* expressions, we should generate two different communication patterns. The first one is to satisfy the dependences indicated by the *wave-front* expressions across processes during the first SPMD block. In cases (c) and (d), Alg. 1 is tailored to use $W_O^k(\ldots)$ for output working-set indexes, and for input working-set indexes, we use $W_F^k(\ldots)$. The C_R comm-tuples are executed *before* the parallel structure, blocking the process until the dependences from other processes are satisfied. The C_S comm-tuples are executed *after* the parallel structure.

The second communication pattern aims to communicate with the following parallel structure. If it is another different parallel structure, case (c), we use Alg. 1 in the standard way. However, in the case of a parallel with *wave-front* expressions inside a loop (d), we should skip the data already communicated due to the flow dependences. For this particular case, we build a tailored function following Alg.2, using functions $W_O^k(\ldots)$, $W_I^k(\ldots)$, and $W_F^k(\ldots)$.

In the same way than this communication calculation is performed, it would be possible to calculate at runtime the part of the data domain where it is possible to overlap computation with the communication of other parts. The calculation can be performed also at runtime, by operating (intersecting, subtracting, etc.) with the HitShape objects that contain information about the domains of: the data to be sent, the data to be received, and the data to be computed. Future work includes the implementation of this feature in the framework.

4.3. Communication patterns for specific applications

In the general case, determining the communication structures involves the comparison of the local working sets with the working sets of the rest of the active distributed processes. Thus, the

Table 1: Input data sizes ($N \times N$), time loop iterations (T), and threshold parameter, for the different benchmarks in the experimental studies conducted in Heracles and CETA.

Machine	**Heracles**	**CETA**
Benchmarks	**Sizes, iterations, threshold**	**Sizes, iterations, threshold**
Illustrative example	N = 7500, T = 200	N = 7500, T = 200
Stencil-Opt	N = 5000, Threshold = 0.001	N = 5000, Threshold = 0.001
Cannon's algorithm	N = 7680	N = 7680
Matmul	N = 4000	N = 4000
Jacobi-2d	N = 7000, T = 1000	N = 5000, T = 800
Gauss-Seidel	N = 7000, T = 1000	N = 5000, T = 800
Blur-Roberts	N = 13000	N = 13000
Gemver	N = 8000	N = 600

computation cost of the communication calculation at runtime grows linearly with the number of virtual processes P in the topology. In many applications, the calculation of the communication patterns can be moved out of the loops and computed at initialization time. However, in applications where the communication expressions are parametrized with loop indexes or other parameters, the expressions are not invariant, and the communication patterns should be computed at every loop iteration.

The new Trasgo prototype allows the addition of specialized transformation modules that, by input code inspection, can detect parallelism patterns, and substitute the generic communication calculation by specific optimized functions that do not traverse all the other processes to compute working-set index intersections. The time to compute communications in these cases does not grow with the number of processes. For example, by checking the expressions used in the tile selections, it is possible to detect stencil computations, that derive in neighbor synchronization structures. Similarly, a circular shift pattern is also detectable in Cannon's algorithm for matrix multiplication. Our Trasgo prototype includes modules for some simple stencil and shift patterns, which substitute the generic communication calculation by code that calculates the intersections only with the needed neighbors for both input and output working sets.

These modules to detect specific patterns reduce the calculation communication times. Nevertheless, they cannot be generalized to any dependences pattern or mapping policy chosen. More well-known applications or design patterns can be analyzed and implemented. It is an interesting research question if any well-defined pattern can be detected, and its corresponding communication code can be generated for any mapping policy chosen at runtime.

5. Experimental study

We have conducted an experimental study to validate our approach, and to verify the efficiency of the resulting codes. We present four different performance studies:

- One of our main contributions is the ability of our technique to automatically calculate communications on a distributed-memory programming model without a fixed tile size at compile time. For this reason, the experimental study starts with a performance study to show the performance improvement achieved when the tile size is independently tuned at runtime for each machine involved in the computation.

- Second, we evaluate the potential overhead that our runtime technique can introduce when adding more computing elements. The simulation study shows a comparison of the runtime cost of the general communication determination using the described algorithms with respect to using communication patterns for specific applications already included in Trasgo.

- Third, we perform an end-to-end measure, including creation of data structures, data initialization, and the rest of Trasgo potential overheads to compare the programs generated by our Trasgo prototype with MPI programs manually developed and optimized.

- The last study presents a comparison in terms of computation and communication (determination plus execution) times with a state-of-the-art polyhedral code generator for distributed-memory systems, the Pluto-MPI compiler [2], using several benchmarks of the PolyBench [14].

A more concise description of each application and benchmark used in the studies can be found in the supplementary material of this paper.

5.1. Experimental platforms and setup

Two clusters have been used in the different experimental studies. The first one is a homogeneous distributed-memory system called *CETA*. It is a hybrid cluster that belongs to CIEMAT and the Spanish government. The cluster nodes are connected by Infiniband technology, and they have two Intel Nehalem-based 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 other cluster, *Atlas*, is composed by two multicore machines (*Heracles* and *Zeus*), that acts as a distributed-memory cluster. *Heracles* is a Dell PowerEdge R815 server, with 4 AMD Opteron 6376 processors at 2.3 GHz, with 16 cores each, and 64 cores in total. *Zeus* is a 6-core Intel E5-2620 v3 at 2.40GHz (up to 12 threads with hyperthreading). Heracles is used in all the experiments to test the scalability of the generated message-passing codes in shared-memory platforms.

In the experimental studies, all the codes, including the MPI reference codes and the programs generated by Trasgo or Pluto, are compiled with GCC v4.8.3 with `-O3` flag. We use *mpich3* v3.1.3 as MPI implementation.

Table 1 shows the benchmarks, input sizes of the matrices, threshold, and number of iterations for the experimental studies using Heracles alone and CETA, except for the first experimental study, where we use the full Atlas cluster with matrices of $1\,500x1\,500$ data elements. These problem sizes have been chosen to generate enough computational load to obtain significant results for our experimental platforms. In some cases, like Gemver in CETA, the size is the maximum supported in the platform by the distributed version of Pluto-MPI that replicates the memory footprint of the whole global data structures for each process.

As the focus of our work is the efficient automatic calculation and execution of communications among processes, we have launched, in all the experiments, a distributed process for each computational unit, without exploiting the shared memory of the machines. All the results presented are the minimum execution time on ten repetitions of each experiment, to eliminate the outliers produced by stochastic delays in the communication systems.

Table 2: Computation times (seconds), of the matrix multiplication benchmark with a size of $1500x1500$ on the cluster *Atlas* with different tunning of the tile size. *TS-Heracles*, when applying the best tile size for the Heracles machine in both machines. *TS-Zeus*, when applying the best tile size for the Zeus machine in both machines. *Best TS* represents the program that chooses the best tile size found empirically for each machine.

Processes	**TS-Heracles**	**TS-Zeus**	**Best TS**
1+1	11.32	6.23	6.23
2+2	5.69	3.17	3.17
4+4	2.87	2.24	1.78
6+6	2.00	2.11	1.75
12+12	1.16	1.28	0.99

Table 3: Execution time for the communication determination for Jacobi-2D solver (seconds).

Processes	**General Model**	**Specific Model**	**Hierarchical mapping policy**
256	2.27×10^{-3}	3.08×10^{-5}	7.60×10^{-4}
1 024	3.92×10^{-3}	3.12×10^{-5}	1.48×10^{-3}
16 384	0.12	2.97×10^{-5}	1.54×10^{-3}
262 144	2.41	3.27×10^{-5}	2.38×10^{-1}
1 048 576	53.22	3.21×10^{-5}	9.53×10^{-1}

5.2. Improvement achieved by tuning the tile size for each process

We have developed an experimental study to show the positive impact on the performance of tuning the tile size at runtime based on the execution machine details [6].

We use as benchmark the classical algorithm for matrix multiplication (in CMAPS). We have executed the program with different tile sizes in the two different machines of the Atlas cluster, to empirically determine which is the best tile size for each machine. Then, we execute the program distributing the processes across both machines at the same time. Table 2 shows the runtime execution times when the matrix multiplication is executed (1) using the best tile size found for Heracles in both machines (TS-Heracles), (2) using the best tile size found for Zeus in both machines (TS-Zeus), and (3) using, on each machine, its best tile size (Best TS),

We observe that the best performance is achieved when the tile size is tuned for each machine independently. Unlike previous techniques, our solution does not analyze the index domain and nor does it generate communication code at compile time. This allows the tile size to be adjusted at runtime using different approaches. For example, fixing a parametric tile size in an already tiled input code or using works, such as [16], that provide different ways to choose the best tile size at runtime. Our method allows the application of this kind of optimization techniques without changing the communication codes. This feature is not found in other techniques of the related work, where the tiling is also used as a main feature to generate the communication code, and thus the tile size cannot be changed at runtime.

5.3. General communications model vs. patterns for specific applications

In this study, we evaluate the potential overhead that our runtime technique can introduce when the number of processes is really high. Each process computes its communication structure independently. Thus, we can isolate and run the code to compute the communication structures on a single process with the proper parameters to simulate the calculations that would be done if

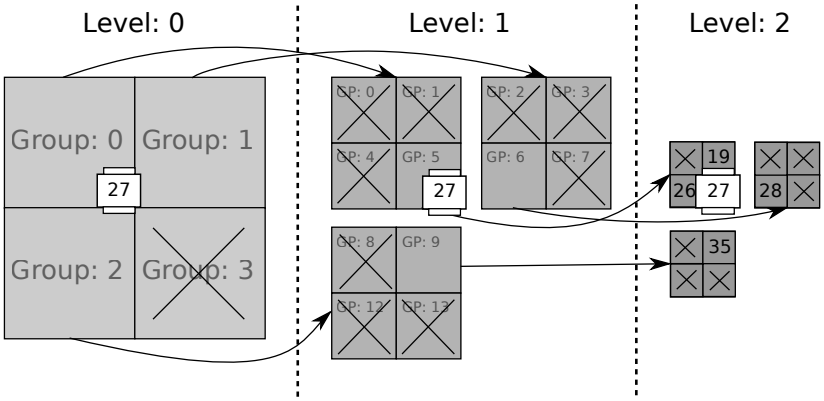

Figure 10: Application of the proposed communication calculation technique when using a hierarchical QuadTree mapping policy to distribute a matrix on 64 processes. White rectangular shapes represent the W_I of the 28-th process (id=27) for the Jacobi-2D benchmark. Crosses point out the processes or groups of processes whose mapped data do not intersect with the locally accessed domain at a given level, detecting that no communication is needed. Thus, they are not checked at lower levels.

the application were to be launched with any given number of processes. This allows us to obtain accurate measures of the cost of the communication determination alone for a huge number of processes. This study has been carried out with one MPI process in Heracles.

Table 3 shows the execution times obtained to compute the communications structure of the Jacobi-2D solver. It compares the actual time to calculate communications using the general model, described in Sect. 4.2, with the optimized pattern described in Sect. 4.3. Moreover, we also compare the time spent to calculate communications using the general model when a Qtree hierarchical mapping policy is used to distribute the data. This kind of mapping policies define the distribution of data in several levels (see Fig. 10). Our technique is performed at each hierarchical level recursively. The calculation operations (such as intersections or subtractions among domains) are only performed in the next lower/finer level for the parts of the current level whose intersection with the local accessed domain is not empty. Thus, in the example shown in the figure with 64 processes, instead of comparing with all the processes, the comparison is only performed with 4 domains at level 0, 12 at level 1 and 12 at the last level. Using this kind of hierarchical mapping techniques, we obtain a calculation time bounded by $O(log(P))$ for patterns which imply communication with a constant number of processes.

The time of the general model grows linearly with the number of processors as expected (P). For the Jacobi-2D example, we can see on Tab. 3 that it is less than two seconds, even for hundreds of thousands of processors. But it may be a hindrance for millions of processors, or if the pattern needs to be recomputed at different iterations of a loop. In these cases, a hierarchical decomposition and nested parallel structures can alleviate the problem, as we can see in the third column of the table. This simulation verifies that the communication calculation time when using the general model on a hierarchical mapping policy, is reduced to be proportional to $log(P)$ instead of P.

On the other hand, the specific pattern for the Jacobi-2D stencil always computes neighbor synchronization pairs, independently of the data sizes or the number of processes. The time in this case is bounded and negligible.

Table 4: Performance (in seconds) obtained for the three benchmarks chosen, for MPI reference versions, and for Trasgo generated codes. Cannon's algorithm by design requires a number of processes with a perfect square root.

	Illust. case		Stencil-Opt		Cannon's	
Machine	**MPI**	**Trasgo**	**MPI**	**Trasgo**	**MPI**	**Trasgo**
Heracles-4	14.48	18.04	163.47	196.86	174.31	186.01
Heracles-8	17.51	20.28	116.76	118.68	-	-
Heracles-16	9.79	11.07	57.34	67.43	56.66	62.94
Heracles-32	4.42	5.43	35.83	43.61	-	-
Heracles-64	4.00	5.08	31.62	35.89	16.10	17.95
CETA-4	25.22	27.69	147.79	166.72	173.46	173.71
CETA-8	22.68	23.20	108.94	114.01	-	-
CETA-16	11.03	12.72	100.97	111.19	48.89	58.46
CETA-32	6.04	6.64	80.11	86.08	-	-
CETA-64	3.37	3.63	57.54	60.20	18.54	21.37

The use of hierarchical mapping policies is an interesting feature to scale the use of the proposed technique to really huge amounts of processes, with the target of exascale computing in mind. However, for the small sizes of the machines used in the rest of our current experimental studies, we use the general model with a simple one level mapping policy, because determining the communication patterns for these cases has an unnoticeable impact on the performance, for any model or application.

5.4. Comparison with MPI references

In this study, we compare the programs generated by our Trasgo prototype with MPI programs manually developed and optimized. In order to do that, we perform an end-to-end measure, including creation of data structures, data initialization, and the rest of Trasgo overheads. For this study we use: the Illustrative example, an implementation of Cannon's algorithm for matrix multiplication [17], which is specially devised for distributed-memory systems in order to minimize the memory footprint, and an optimized stencil computation (Stencil-Opt) whose communication pattern is data-dependent and redetermined on each time iteration of the stencil program. These benchmarks are described in the supplementary material.

Table 4 shows the performance obtained by the MPI reference versions, and the Trasgo generated programs for the cases of study. We execute the applications on both, shared-memory and distributed-memory machines. We see that Trasgo programs scale quite well, but losing some performance in comparison with the manually optimized MPI codes (less than 20% in the worst cases). This shows that the implementation of the proposed technique in Trasgo produces efficient parallel programs at the communication level.

5.5. Comparison with a state-of-the-art tool

In this study, we have performed an in-depth comparison of the performance between the codes generated by Trasgo, and those generated for distributed-memory by Pluto-MPI (dist-mem), a polyhedral model compiler that includes state-of-the-art techniques for generating communication code [3]. We choose Pluto-MPI because: (1) According to the authors, it is the first work that reported an end-to-end fully automatic distributed-memory parallelization and code generation for input programs and transformation techniques; (2) It is a free available tool easy to install, which supports all the benchmarks tested in the paper; (3) Many research works have appeared that use Pluto as baseline, for both shared- and distributed-memory systems, such as [18, 19, 20]; (4) To the best of our knowledge, the methods of Pluto for code generation in

Table 5: Maximum variation in the execution times for each benchmark in Heracles and CETA, when using Trasgo and Pluto. The variation ratio is defined using the following formula: ((*max time* − *min time*)/*min time*).

Machine	**Heracles**	
Benchmarks	**Trasgo**	**Pluto**
Matmul	0.0231	0.0575
Jacobi-2d	0.0490	0.0707
Gauss-Seidel	0.0079	0.0160
Blur-Roberts	0.2670	0.2591
Gemver	0.1192	0.1965

Machine	**CETA**	
Benchmarks	**Trasgo**	**Pluto**
Matmul	0.0262	0.0260
Jacobi-2d	0.0196	0.0177
Gauss-Seidel	0.0119	0.0056
Blur-Roberts	0.1695	1.1790
Gemver	0.5304	0.4878

distributed-memory systems, comparing with others, are the ones that reduce the communicated volume of data, being the generated code parametric in the number of processes and problem sizes.

We have selected five examples from Polybench [14], a collection of examples to be used for testing and developing polyhedral model compilation techniques. The examples are Jacobi-2D, Gauss-Seidel, Blur-Roberts, the classical Matrix Multiplication algorithm and the Gemver algorithm. We have slightly modified the stencil in the Jacobi-2D, and the Gauss-Seidel examples. Instead of computing a 5-point star stencil, we compute the 4-point star stencil of Poisson's equation. It reduces the computation load per process, leading to a slightly bigger impact of the communications, which is the focus of this study. Gauss-Seidel has been selected because it presents *wave-front* dependences, deriving in a macropipeline computation. Matrix Multiplication is a good case for simple linear algebra problems with a nice computation/communication balance. On the other hand, Gemver presents a communication/computation balance that is not adequate to distributed-memory systems (communication time typically is higher than computation). The Blur-Roberts filter is a kind of stencil with a single iteration. They were chosen to show the performance of this kind of applications when the techniques discussed in this paper are applied.

We compile the codes with the *Makefiles* provided by default with Pluto-MPI, which include a *distopt* option enabling the use of the FOP communications model [2], and a set of default flags and tile size values for each example. The *Makefiles* for the stencil examples do not include the flag *-l2tiles* to enable multi-level tiling. The internal tools used in Pluto-MPI do not handle it in an affordable compilation time.

For time measuring, in this study, we have taken into account only the main computation and communication times, named *Seq.* and *Comm.* in the result tables. In Pluto-MPI, the full matrices are allocated and initialized in all processes, although only the parts needed for the local computation are used. At the end of a parallelized affine loop nest, Pluto needs to create again a common global state communicating local results for each process. We have skipped all these times in our Pluto-MPI measures. For fair comparison, we measure in all the codes as communication times only: The cost of packing and unpacking the data, the cost of communicating control information needed for the communications (only in Pluto-MPI), network latencies and synchronization waits. In the computation parts, we have selected the same tile sizes on each example for both, Trasgo and Pluto codes. In addition, in order to provide a measure of the stochastic delays, we also show in Tab. 5 the maximum variation of the execution times, for the different benchmarks, for each different execution platform, and for each different tool tested, Trasgo and Pluto. The maximum variation is represented as a ratio of the time difference between the minimum and the maximum execution times. We observe that in the applications Blur-Roberts and Gemver, where the communication time is much higher than the computation

Table 6: Main execution times (in seconds) for the five benchmarks chosen from the Polybench.

	Jacobi-2d		Gauss-Seidel		matmul		Gemver		Blur-Roberts	
Machine	**Trasgo**	**Pluto**	**Trasgo**	**Pluto**	**Trasgo**	**Pluto**	**Trasgo**	**Pluto**	**Trasgo**	**Pluto**
Heracles-4	142.65	101.67	428.36	274.04	41.34	28.46	0.65	0.75	0.38	1.36
Heracles-8	83.35	77.74	253.40	196.52	23.23	14.41	1.15	0.76	0.29	1.49
Heracles-16	46.61	58.26	142.41	156.02	13.24	8.26	1.60	0.78	0.24	1.75
Heracles-32	24.18	59.47	113.21	138.58	7.23	4.67	1.85	0.83	0.14	2.05
Heracles-64	18.51	61.32	64.91	128.11	4.05	2.39	2.52	0.86	0.11	2.77
CETA-4	53.44	38.20	122.13	72.08	26.19	30.31	0.0025	0.0049	0.48	1.61
CETA-8	37.03	28.73	71.08	59.87	13.33	14.36	0.0999	0.0046	0.38	5.67
CETA-16	24.28	33.20	62.86	44.93	7.84	12.73	0.1291	0.0787	0.41	20.88
CETA-32	14.62	21.13	40.25	46.65	7.05	6.84	0.4696	0.1673	0.35	47.05
CETA-64	14.56	24.36	35.09	70.30	7.56	4.13	0.4641	0.1310	0.20	59.78

time, the variation of the execution times is really high because of the stochastic delays of the network. In order not to take into account this stochastic effects of the network infrastructure, in the rest of tables, we show the minimum execution time achieved in the experiments.

In Table 6, we present the total execution times (*Seq+Comm*) considered for each benchmark. In Table 7, we present independently the accumulated time expended in the computation stages, and the accumulated time expended in the communication calculation, execution, and synchronization waits. Each result is the measure obtained for the process that expended more time in the corresponding stages. Thus, we can observe in which cases the communication cost is higher or lower, independently of the computation code.

The results for the Jacobi-2d, and the Gauss-Seidel examples indicate that the Pluto code is more efficient for a low number of processes, although it does not scale as well as Trasgo. The transformations performed by Pluto, including skewing the time loop to parallelize, derive in a lot of re-utilization and exploitation of memory hierarchies inside the processes. Trasgo codes exploit only spatial parallelism at the distributed-level, as in classical manual message-passing approaches. This derives in coarse-grained communications, but fewer opportunities to exploit computation code optimizations inside the processes due to extra synchronizations. It is specially noticeable for the macro-pipeline structure from which Gauss-Seidel derives. However, as the number of processes grows, Pluto reveals its more clumsy communication calculations, while the granularity of the Trasgo communications decreases, and its reduced costs for communications become much more relevant. See the communication cost in Table 7 for these examples.

In the case of Pluto codes, the full matrices are allocated and initialized in all the processes. On the other hand, Trasgo use actually distributed arrays, with much lower memory footprint. However, Trasgo needs a communication stage before the execution of each SPMD block. In the case of the matrix multiplication, there is only one execution of an SPMD block. Thus, the communication times in Table 7 for Trasgo only include the time to redistribute the data needed for each process to compute its local part. On the other hand, Pluto does not need a communication stage beyond the global state consolidation, which we do not consider in our study. The communication times for the matrix multiplication in Pluto codes in Table 7 are due mainly to the synchronization times to exchange control information in order to determine that no communication is necessary for any process.

As expected, the Gemver example shows poor scalability for both Trasgo and Pluto distributed-

memory programs. The computational load is really low, with several high-volume communication stages. The cost of executing the communications is higher than the computation. The sequential algorithm in the Gemver benchmark is not a good candidate for distributed-memory programming in general. The performance in this case can be improved in both approaches using a different mapping policy to distribute the computational load among the processes. However, this study is beyond the scope of this paper.

The Blur-Roberts filter is a kind of stencil program with a single iteration with two SPMD blocks. The results show that the transformation of SPMD blocks into an affine loop nest performed by Pluto sometimes implies poor performance, specially in distributed-memory systems, due to the need for multiple communication stages in the generated pipeline (although locality is improved). This effect is highly noticeable for CETA, the distributed-memory cluster (see *Comm.* times in Table 7). Using solutions such as diamond-tiling can alleviate this problem in Pluto. On the other hand, Trasgo issues a single communication stage for each SPMD block, with the expected scalability.

We conclude that Trasgo codes scale very well due to their very efficient communication structures, despite the fact that the computation code can still be optimized further.

6. Related Work

The polyhedral model provides a formal framework to develop automatic transformation techniques at the source code level [21]. It is applicable to codes based on sequential static loops with affine expressions (SCoP). All the polyhedral techniques presented so far for distributed memory need to parametrize the iteration space polyhedra and analyze dependences at compile time. There are many similarities between our work and specific polyhedral model techniques. For distributed memory, the best communication calculation methods so far (see e.g. [2, 22, 3]) compute communications for sequences of arbitrary nested loops with regular (affine) accesses, also known as affine loop nests. The loops are transformed, tiled and finally parallelized. Communications cannot be calculated across different sections of affine loop nests unless loop fusion can be done. These techniques analyze at compile time the footprint of tile data used by other tiles. This implies that the tile size must be fixed at compile time and must be the same for all the machines involved in the computation. Moreover, using these methods, there are still cases for duplicated or unnecessary data communications [2]. The run-time complexity of the generated code is dependent on both the data size (number of tiles) and the number of processing elements [23]. Multi-level tiling techniques can be used to alleviate the problem. However, this introduces more tile-size decisions at compile time. Communication across tiles of different levels is harder, and communicating across tiles of bigger sizes increases the cases of redundant communications.

There are tools which bring together the advantages of the polyhedral model and the task-oriented programming proposals [19]. These approaches reduce global barriers in shared-memory parallel codes by launching tasks and computing their dependences and footprints. However, this approach also performs a data partition after a tiling technique is applied with predefined tile sizes at compile time [11]. The best tile size depends, among other things, on the architecture details of the target machine where the program will be executed [6]. Choosing tile sizes at compile time prevents automatic tuning for different devices in heterogeneous environments. There are some proposals such as [24] that generate loops that iterate over full rectangular tiles, with unknown parametric tile size. However, it has not been demonstrated that these techniques can be applied with the current communication code generators for distributed-memory systems.

Table 7: Performance (in seconds) of Polybench codes, generated for distributed-memory by Trasgo, and by Pluto-MPI, broken down into computation and communication times (including calculation and execution).

	Jacobi-2d				**Gauss-Seidel**				**Matmul**			
	Trasgo		**Pluto**		**Trasgo**		**Pluto**		**Trasgo**		**Pluto**	
Machine	**Seq.**	**Comm.**	**Seq.**	**Comm.**	**Seq.**	**Comm.**	**Seq.**	**Comm.**	**Seq.**	**Comm.**	**Seq.**	**Comm.**
Heracles-4	141.94	2.24	85.35	49.43	226.32	283.24	192.41	116.34	41.24	0.09	28.46	2.73
Heracles-8	80.56	16.51	52.52	54.18	129.52	182.18	100.02	123.70	23.05	0.18	14.41	2.69
Heracles-16	45.93	15.24	30.04	48.48	70.61	108.35	53.80	122.33	13.05	0.20	8.26	3.30
Heracles-32	22.46	7.83	22.23	54.78	52.88	95.59	33.25	123.64	7.05	0.18	4.67	3.49
Heracles-64	15.57	7.86	22.96	63.83	27.73	55.60	22.11	123.51	3.79	0.26	2.39	2.39
CETA-4	53.04	8.09	36.88	15.98	62.93	75.75	55.65	29.37	26.12	0.07	30.31	2.62
CETA-8	36.17	13.19	22.15	17.87	32.89	52.95	28.60	36.43	13.22	0.13	14.36	2.66
CETA-16	20.24	13.14	16.49	30.66	22.83	53.43	14.39	35.78	6.97	0.89	12.72	6.16
CETA-32	8.63	9.40	8.54	22.41	19.62	36.24	10.79	43.36	5.32	1.79	6.81	5.85
CETA-64	5.77	12.29	8.55	25.39	8.81	33.09	9.35	70.47	5.60	1.69	4.07	4.13

	Gemver				**Blur-Roberts**			
	Trasgo		**Pluto**		**Trasgo**		**Pluto**	
Machine	**Seq.**	**Comm.**	**Seq.**	**Comm.**	**Seq.**	**Comm.**	**Seq.**	**Comm.**
Heracles-4	0.1816	0.4648	0.7514	0.7538	0.35	0.03	0.70	0.93
Heracles-8	0.1607	0.9864	0.7553	0.7585	0.21	0.06	0.47	1.41
Heracles-16	0.1417	1.4545	0.7790	0.7848	0.19	0.07	0.41	1.78
Heracles-32	0.0504	1.7966	0.8158	0.8286	0.08	0.05	0.39	2.19
Heracles-64	0.0383	2.4794	0.8442	0.8679	0.09	0.05	0.39	3.25
CETA-4	0.0010	0.0019	0.0026	0.0033	0.38	0.10	0.75	1.01
CETA-8	0.0005	0.0989	0.0031	0.0045	0.38	0.15	0.74	4.92
CETA-16	0.0003	0.1203	0.0037	0.0462	0.22	0.33	0.70	19.48
CETA-32	0.0003	0.3767	0.0036	0.1987	0.11	0.30	0.37	45.20
CETA-64	0.0002	0.4633	0.0035	0.1967	0.05	0.16	0.41	53.93

Other similar approaches based on compile-time intersections of parametric polyhedra have been proposed to reduce data transfers in accelerators, such as FPGAS [25], where communications are calculated and optimized only between the host and the accelerator. Distributed-memory programs introduce the complexity of dealing with data partition policies, and different communication patterns across a number of processes only known at runtime.

The work in [26] presents a hybrid compiler-runtime translator scheme, similar to our approach, that calculates the communication pattern needed among the SPMD blocks. However, they only support *regular and repetitive* applications where the communication pattern is the same in all the iterations of the outer serial loop that encloses the SPMD blocks. This constraint is also found in other distributed-memory approaches that integrate classical polyhedral techniques for regular codes, with inspector/executor techniques [27] to support irregular or indirect data access expressions. This inspector/executor technique exchanges control data before actual communications to avoid traversing the whole iteration space of the parallelized loop on every process. Unlike our approach, these solutions cannot be used, for example, in our *Stencil-Opt* benchmark, where the communication pattern is recalculated on each iteration.

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 [28]. 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 [29] is restricted to global array assignments with block or cyclic distributions. The work in [30] 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.

7. Conclusion

This paper presents an extension of the Trasgo parallel programming and compiling framework. This extension includes techniques that, for affine expressions of parallelism on data accesses, automatically determine at runtime ad-hoc communication patterns for distributed-memory processes across two consecutive SPMD blocks. This new technique uses the results of a partition policy to compute at runtime exact coarse-grained communication patterns for distributed message-passing processes. It is based on intersections of remote and local footprints in terms of the results of the mapping functions chosen. Our approach allows the automatic generation of pre-compiled multi-level parallel libraries or programs, that can adapt their communication and synchronization structures to the target system. Experimental results, for several representative cases of study, show that our technique produces efficient codes, despite the overhead of our runtime communication calculation, compared with a compile-time state-of-the-art tool that generates communication codes, and with manually implemented and optimized pure-MPI references codes.

Future work includes the applicability of the transformation model in the context of current polyhedral model frameworks, using more irregular domains, or extending it for non-completely affine expressions. The new Trasgo framework is available at `http://trasgo.infor.uva.es`.

Acknowledgments

This research has been partially supported by MICINN (Spain) and ERDF program of the European Union: HomProg-HetSys project (TIN2014-58876-P), CAPAP-H6 (TIN2016-81840-REDT), COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS), and 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] M. Claßen, M. Griebl, Automatic code generation for distributed memory architectures in the polytope model, in: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, IEEE, 7–pp, 2006.

[2] U. Bondhugula, Compiling Affine Loop Nests for Distributed-Memory Parallel Architectures, in: Proc. SC'2014, ACM, Denver, CO, USA, 2014.

[3] R. Dathathri, C. Reddy, T. Ramashekar, U. Bondhugula, Generating efficient data movement code for heterogeneous architectures with distributed-memory, in: Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on, IEEE, 375–386, 2013.

[4] T. Yuki, S. Rajopadhye, Parametrically Tiled Distributed Memory Parallelization of Polyhedral Programs, Tech. Rep. CS13-105, Colorado State University, 2013.

[5] A. Gonzalez-Escribano, D. Llanos, Trasgo: A Nested-Parallel Programming System, The Journal of Supercomputing 58 (2) (2011) 226–234.

[6] S. Mehta, G. Beeraka, P.-C. Yew, Tile size selection revisited, ACM Transactions on Architecture and Code Optimization (TACO) 10 (4) (2013) 35.

[7] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT Press, 2014.

[8] A. v. Gemund, The Importance of Synchronization Structure in Parallel Program Optimization, in: Proc. *11th* ACM ICS, Vienna, ISBN ACM ISBN: 0-89791-902-5, 164–171, 1997.

[9] K. Lodaya, P. Weil, Series-Parallel Posets: Algebra, automata, and languages, in: Proc. STACS'98, vol. 1373 of *LNCS*, Springer-Verlag, Paris, France, 555–565, 1998.

[10] A. Moreton-Fernandez, A. Gonzalez-Escribano, D. R. Llanos, A New High-Level Parallel Portable Language for Hierarchical Systems in Trasgo, in: Computational and Mathematical Methods in Science and Engineering (CMMSE), 2015.

[11] U. Bondhugula, A. Hartono, J. Ramanujam, P. Sadayappan, A Practical Automatic Polyhedral Program Optimization System, in: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2008.

[12] A. Moreton-Fernandez, A. Gonzalez-Escribano, D. Llanos, Exploiting distributed and shared memory hierarchies with Hitmap, in: Proc. HPCS'2014, Bologna (Italy), 278–286, 2014.

[13] A. Gonzalez-Escribano, Y. Torres, J. Fresno, D. Llanos, An extensible system for multilevel automatic data partition and mapping, IEEE TPDS 25 (5) (2013) 1145–1154, (doi:10.1109/TPDS.2013.83).

[14] L.-N. Pouchet, Polybench: The polyhedral benchmark suite, URL: http://www. cs. ucla. edu/˜ pouchet/software/polybench/[cited July,] .

[15] R. Upadrasta, A. Cohen, Sub-polyhedral scheduling using (unit-) two-variable-per-inequality polyhedra, in: ACM SIGPLAN Notices, vol. 48, ACM, 483–496, 2013.

[16] B. Pradelle, P. Clauss, V. Loechner, Adaptive runtime selection of parallel schedules in the polytope model, in: Proceedings of the 19th High Performance Computing Symposia, Society for Computer Simulation International, 81–88, 2011.

[17] L. Cannon, A cellular computer to implement the kalman filter algorithm, Doctoral dissertation, Montana State University Bozeman, 1969.

[18] Y. Barigou, E. Gabriel, Maximizing Communication–Computation Overlap Through Automatic Parallelization and Run-time Tuning of Non-blocking Collective Operations, International Journal of Parallel Programming (2016) 1–27.

[19] M. Kong, A. Pop, L.-N. Pouchet, R. Govindarajan, A. Cohen, P. Sadayappan, Compiler/runtime framework for dynamic dataflow parallelization of tiled programs, ACM Transactions on Architecture and Code Optimization (TACO) 11 (4) (2015) 61.

[20] M. Kong, L.-N. Pouchet, P. Sadayappan, V. Sarkar, 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, IEEE Press, 39, 2016.

[21] C. Bastoul, Code Generation in the Polyhedral Model Is Easier Than You Think, in: Proc. PACT'04, ACM Press, 7–16, 2004.

[22] C. Reddy, U. Bondhugula, Effective automatic computation placement and data allocation for parallelization of regular programs, in: Proceedings of the 28th ACM international conference on Supercomputing, ACM, 13–22, 2014.

[23] A. Moreton-Fernandez, A. Gonzalez-Escribano, D. R. Llanos, On the run-time cost of distributed-memory communications generated using the polyhedral model, in: High Performance Computing & Simulation (HPCS), 2015 International Conference on, IEEE, 151–159, 2015.

[24] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, P. Sadayappan, Parametric multi-level tiling of imperfectly nested loops, in: Proceedings of the 23rd international conference on Supercomputing, ACM, 147–157, 2009.

[25] L.-N. Pouchet, P. Zhang, P. Sadayappan, J. Cong, Polyhedral-Based Data Reuse Optimization for Configurable Computing, in: ACM/SIGDA FPGA'13, 29–38, 2013.

[26] O. Kwon, F. Jubair, R. Eigenmann, S. Midkiff, A hybrid approach of OpenMP for clusters, in: ACM SIGPLAN Notices, vol. 47, ACM, 75–84, 2012.

[27] M. Ravishankar, R. Dathathri, V. Elango, L.-N. Pouchet, J. Ramanujam, A. Rountev, P. Sadayappan, Distributed memory code generation for mixed Irregular/Regular computations, in: Proc. PPoPP'2015, ACM, 65–75, 2015.

[28] B. Chamberlain, S. Deitz, D. Iten, S.-E. Choi, User-Defined Distributions and Layouts in Chapel: Philosophy and Framework, in: 2nd USENIX Workshop on Hot Topics in Parallelism, 2010.

[29] A. Sanz, R. Asenjo, J. López, R. Larrosa, A. Navarro, V. Litvinov, S.-E. Choi, B. Chamberlain, Global data reallocation via communication aggregation in Chapel, in: Proc. SBAC-PAD'2012, IEEE, 2012.

[30] A. Sharma, D. Smith, M. Ferguson, J. Koehler, R. Barua, Affine Loop Optimization Based on Modulo Unrolling in Chapel, in: Proc. PGAS'2014, ACM, Eugene, OR USA, 2014.