Computer Science Department
University of Valladolid
Valladolid - Spain

Calculating coarse communication code for parallel distributed-memory systems

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

Computer Science Department, University of Valladolid, Spain.
ana.moreton@alumnos.uva.es, {arturo, diego}@infor.uva.es.

Abstract This paper presents a technique that automatically calculates at runtime, for affine expressions, the exact aggregated communications needed in a distributed-memory programming model. This technique generalizes and improves previous methods presented to compute aggregated communications. It avoids duplicated data communications, it reduces the number of communication operations, and allows dynamic loops with data-dependent conditions. It is based on intersections of remote and local footprints in terms of the mapping functions chosen. The technique works with information about data accesses obtained from the original input code, and relays on information about the mapping of the distributed data structures provided by the runtime-system.

Technical Report No. IT-DI-2015-0002

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. Also, the focus on parallel applications is shifting to more diverse and complex solutions, exploiting several levels of parallelism, with different strategies of parallelization. Therefore, it is increasingly interesting to generate application programs with the ability of automatically adapt their structure and computational load to any given target system. Many successful parallel programming models and tools have been proposed for specific environments. Message-passing paradigm (e.g. MPI libraries) have shown to be very efficient for distributed-memory systems. Global shared memory models, such as OpenMP, Intel TBBs, or Cilk, are commonly used in shared-memory environments to simplify thread and memory management. Polyhedral model provides a formal framework to develop automatic transformation techniques at the source code level [Bas04]. The polyhedral model is applicable to codes based on sequential static loops with affine expressions. Polyhedral model does not support dynamic loops dependent on information not known or parametrizable at compile-time. For distributed memory, the best communication calculation methods so far (see e.g. [Bon13]) are based on run-time analysis per tile (grouping data after applying a tilling technique). They compute tile footprints and use inverse mapping functions to locate remote data. Other previous polyhedral tools for distributed-memory systems such as [BGH$^+$06] generate also communication code. However, using these methods there are still cases for duplicated or unnecessary data communications and, the time complexity of the generated code, that do the analysis at runtime, is dependent on both the data size (tiles) and the number of processing elements [MFGEL15].

In this report we present a technique that, for affine expressions, generates code that computes exact aggregated communications at runtime, for the distributed runtime layer before applying tiling techniques. It uses intersection of remote and local footprints in terms of the mapping policies selected. It generalizes and improves previous methods presented to compute aggregate communications for code with affine expressions such as [Bon13,BGH$^+$06], also supporting dynamic loops with conditions dependent on data-values (or expressions dependent on runtime parameters). It avoids duplicated or unnecessary data communications and eliminates the data size dependence in the communication calculation. Our approach computes communications only once for the whole data space mapped to a local process, independently of the number of tiles generated inside, also allowing to use different tiling sizes for machines with different architectures in the same computation. The technique works with information about data accesses obtained from the original input code, and relays on information about the mapping of the distributed data structures provided by the runtime-system [GETFL13].

The rest of the paper is organized as follows: Section 2 discusses the tools and notations used in the proposal. Section 3 describes the technique. Section 4 presents the conclusions and future work.

2 Background and Notation

2.1 Definitions and notation

In this work we will focus on arrays with regular dense and strided domains. Their index *Domain* is a subspace of Z^n. Rectangular n-dimensional parallelotope domains, dense or strided, can be represented by a tuple of n *Signatures*. A Signature is a triplet of integer numbers $S < b, e, s >$ (meaning *begin*, *end*, and *stride*). The set of indexes expressed by a signature is $S < b, e, s >= \{b \leq i \leq e : ((i - b) \mod s) = 0\}$.

A data structure or *Tile* ($T : D \rightarrow type$) is an object that associates data elements of a given *type* to index elements of a domain. *Array tiles* associate one data element to each domain element. The domain of a tile is denoted as $D(T)$. Hierarchical tiling is a technique that allows to hierarchically declare new Tile structures with a subset of the domain of another Tile. The subtile maps the index elements of its subdomain to the same data elements of the root tile $r(T)$ (the ancestor of the subtiling chain). The *Selection* function $s : T \times D^\star \rightarrow T$, is used to declare a new subtile: $s(t, d) = t' : r(t) = r(t'), D(t') = D(t) \cap d$. A *Working set* ($W$) is a set of Tiles. A *Logical process* is a tuple $P < f, W_I, W_O >$ where f is a function or subprogram, W_I is the working set of tiles that P receives as input (data is read), and W_O is the set of tiles used as output (data is written). The intersection of W_I and W_O can be a set not empty. Logical processes may be composed in sequential, 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.

A *Virtual Topology* $V(N, R)$ is a graph where the vertices N represent virtual processes, associated to computational resources (groups of processors), and the edges R represent neighbor relations. A *Layout* $L : D \rightarrow V$ is a function that maps domain subspaces to the virtual processes in a virtual topology. These functions can be used to map indexes, of logical processes or tiles, to virtual processes.

2.2 Hitmap

Hitmap [GETFL13] is a library for management, and run-time mapping, of hierarchical tiling arrays. It is based on a SPMD model, and the message-passing paradigm. Hitmap has three main functionality modules: (a) Domains and tiles management; (b) Mapping modules; and (c) Communication patterns. Hitmap defines objects to declare and manipulate multidimensional index domains, and different types of indexed data structures [FGEL13]. Hitmap defines a plug-in system to include new mapping modules: Virtual topology constructors that arrange the physical processes in meshes or graphs defining neighborhood relations, and mapping functions named *Layouts* that distribute tile domains across

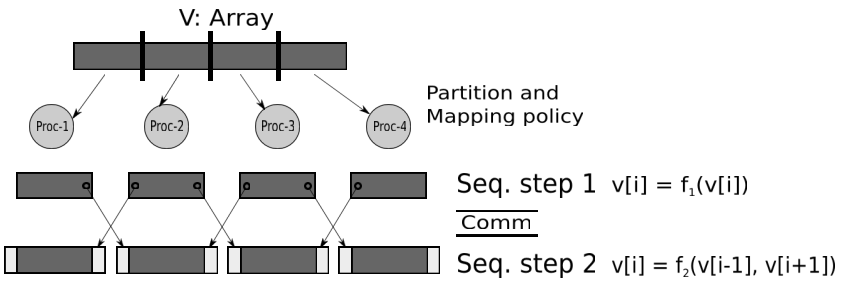

Figure 1. A representation of the distribution of an array among the processors and the data propagation between two sequential stages.

the processes of a virtual topology. These modules generate objects that can be queried at run-time to obtain information about the result of the mapping to know which part of a domain is assigned to the local or a 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.

3 Proposal

Using the functionalities given by Hitmap is possible to create some heuristics to calculate at runtime the necessary communications between two or more stages of a parallel computation in a distributed-memory system. These communications redistribute the data written in one parallel computation step to the following steps where data, recently modified, is read. The information needed to calculate these communications is the data dependences among sequential computations. We can obtain data dependences from explicit details expressed in specific parallel languages, or using dependence analysis techniques on a sequential code.

3.1 Illustrating example

Distributed programming models, based on message-passing, divide the data domain among the different processors. Dependences only exist when the local process needs to read data, which has been written by other process.

Figure 1 represents an array V distributed using a blocks partition and mapping policy across four processes. The first sequential step updates each element of the array applying an arbitrary function f_1 to each element. Each process updates its part directly. However, in a second stage each element is updated on function of the previous and next elements in the domain (Indexed $i-1$ and $i+1$). The updates to the elements in the border of the blocks depend on data

```
** Case 1: JACOBI SOLVER
1. While not converge and iterations < limit
  1.1. For each i,j in M.domain
      M2[i][j] = M[i][j]
  1.2. For each i,j in M.domain
      M[i][j] = ( M2[i-1][j] + M2[i+1][j]
       + M2[i][j-1] + M2[i][j+1] ) / 4;
```

Figure 2. Sequential algorithm of Jacobi solver.

```
** Case 1: JACOBI SOLVER
1. While not converge and iterations < limit
  1.1. For each i,j in M.domain
      M2[i][j] = M[i][j]
  1.1C  Communication stage (WI_1.2, WO_1.1, M2)
  1.2. For each i,j in M.domain
      M[i][j] = ( M2[i-1][j] + M2[i+1][j]
       + M2[i][j-1] + M2[i][j+1] ) / 4;
  1.2C  Communication stage (WI_1.1, WO_1.2, M)
```

Figure 3. Parallel Algorithm of Jacobi solver with the communications stages.

updated in neighbor processes. Thus, communication is needed, and the exact data to be communicated is defined by the data dependences. The first step to generate these communications is to calculate the domains of the data written (named Output Working Set or W_O in this work) and read (named Input Working Set or W_I in this work) in each sequential phase.

We are going to analyse the pseudo-code of a PDE solver using a Jacobi iterative method to compute the heat transfer equation in a discretized two-dimensional space as an illustrating example (see Fig. 2).

We should add a potential communication stage between each two consecutive computation phases. In the case of this example there should be communications between sequential stages 1.1 and 1.2 on each iteration, and between 1.2 and the execution of stage 1.1 on next iteration.

Each communication stage takes into account the W_O of the previous sequential step and the W_I of the next computation stages.

The own or local domain is the domain assigned to each process during the application of the distribution policy at the beginning of the program. We represent the own domain on an array M by $d \in D$, where d is the set of two signatures that represent the part of M assigned to the local process. Let $d < s_0, s_1 > \in D$, be a two dimensional domain, being $s_0 < b, e, s > \in S$ and $s_1 < b, e, s > \in S$ the signatures that define the domain. The Output Working sets (W_O) of the two computation stages (1.1 and 1.2) are always the local domain (writes only occur in the elements associated to the local domain, $M2[i][j]$ and $M[i][j]$, $\forall (i,j) : i \in [s_0.b, s_0.e], j \in [s_1.b, s_1.e]$, see Fig. 4 left side).

We also have to calculate the input Working set (W_I) for each computation stage. Stage 1.1 only reads over its own domain, $M[i][j]$ being $s_0.b \leq i \leq s_0.e$

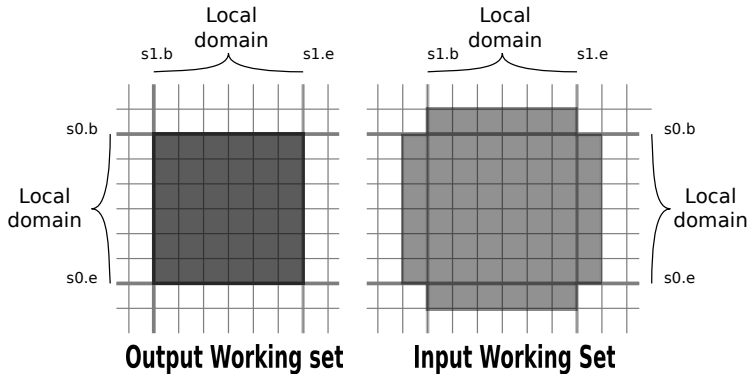

Figure 4. Example of output and input working sets of the stage 1.2, for a given local domain

and $s_1.b \leq j \leq s_1.e$. Thus, $W_I = d$. However, stage 1.2 needs a deeper discussion. In stage 1.2 each process reads data defined by the access expressions:

$\rho_1 = M2[i-1][j],\ \rho_2 = M2[i+1][j],\ \rho_3 = M2[i][j-1],\ \rho_4 = M2[i][j+1]\ \ \forall(i,j) : i \in [s_0.b, s_0.e],\ \ j \in [s_1.b, s_1.e]$

We apply these expressions to create a *shape* for each access expression from the local domain that represent the set of indexes accessed due to the expressions applied to the local subset of indexes (as wee see in Figure 5):

$$\rho_1 = [i-1][j] \rightarrow W_I^1 = (i,j)\ \forall(i,j) : i \in [s_0.b-1, s_0.e-1], j \in [s_1.b, s_1.e]$$
$$\rho_2 = [i+1][j] \rightarrow W_I^2 = (i,j)\ \forall(i,j) : i \in [s_0.b+1, s_0.e+1], j \in [s_1.b, s_1.e]$$
$$\rho_3 = [i][j-1] \rightarrow W_I^3 = (i,j)\ \forall(i,j) : i \in [s_0.b, s_0.e], j \in [s_1.b-1, s_1.e-1]$$
$$\rho_4 = [i][j+1] \rightarrow W_I^4 = (i,j)\ \forall(i,j) : i \in [s_0.b, s_0.e], j \in [s_1.b+1, s_1.e+1]$$

The final W_I is the union of the working sets derived from each read expression: $W_I = W_I^1 \cup W_I^2 \cup W_I^3 \cup W_I^4$. According with the access expressions of the Jacobi solver algorithm we obtain W_O and W_I of the stage 1.2. See an example in Fig. 4. For the 1.1 stage the W_O and W_I are equal to the local domain .

After calculating the shapes of W_I and W_O of each computation stage, we have all the needed information to generate the communications code. Data should be sent from process P_1 to process P_2 between stages 1.1 and 1.2 if the intersection between W_O of stage 1.1 at P_1 and W_I of stage 1.2 at P_2 is not empty. A communication pattern is formed by: (1) The sends that the local process has to issue to other processes in order to allow them to compute its next stage, and (2) the data that the local process needs to receive in order to perform its next computation. The general technique used to calculate a communication stage is presented in the section below.

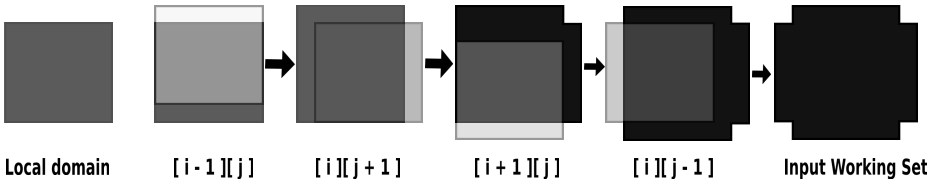

Figure 5. Calculate Input working set from access expressions for Jacobi Solver.

3.2 General model

We present here a generic model for calculating coarse grain communications for affine expressions in distributed-memory systems. First, we generate code that (1) queries the corresponding Hitmap layout objects to obtain the local assigned domain for a given tile; and (2) applies at runtime the affine expressions used in the sequential computations, to transform the domain signatures and obtain the input and output working sets (W_I, W_O). Each communication stage takes into account the W_O of the previous sequential step (stage j) and the W_I of the next computation stage (stage j+1).

For simplicity the model will be first described for unidimensional array domains. It is straightforward to extend it for domains and expressions involving more dimensions, as it will be shown in section 3.3. Let A be an array distributed using the map M_A. Let $d < s_0, s_1 > \in D$ be the local domain of A obtained from M_A at run-time. Let $\rho_x = A[\alpha * i + \beta]$ be the x-th affine access expression used as a read access in a computation stage. The resulting index domain space needed by the whole local process due to that expression is:

$W_I^x = i, \forall i : i \in [min(\alpha * s_0.b + \beta, \alpha * s_0.e + \beta), max(\alpha * s_0.b + \beta, \alpha * s_0.e + \beta)]$.

Output domains due to write access expressions will be calculated in the same way for expressions used in write accesses. Let k' be the number of write access expressions and k'' is the number of read access expressions in a computation stage. The final output and input working sets will be the union of all domains obtained from all the write and read access expressions in the computation stage, respectively.

$$W_O = \bigcup_{x=1}^{k'} W_O^x \qquad W_I = \bigcup_{x=1}^{k''} W_I^x$$

We can symbolically generate code to compute the aggregated domain corresponding to the working sets of all the logical processes assigned to a whole real distributed process. The complexity to compute them at run-time is bounded by the amount of code expressions $k = k'' + k'$ in the code. It is $O(\log k)$ for non-strided or block domains and distributions.

Let P be the number of active processes at run-time. The Hitmap layout objects can be also queried to obtain the domain mapped to any other remote process in $O(1)$. It is possible to generate the working sets W_{I_i} and W_{O_i} of any process p with exactly the same asymptotic time cost than for the local

ALGORITHM 1: General model to calculate communication patterns

Input: Number of processes (P),
Working Set of outputs of each process ($W_{O_p}; p = 1..P$) at stage j,
Working Set of inputs of each process ($W_{I_p}; p = 1..P$) at stage j+1,
Local process id: *local*,
Output: Pattern communication (Set to send (CS), Set to receive (CR) for each process)

```
for each p in [1..P] do
    if p != local then
        CS = CS ∪ < p, (W_{I_p} ∩ W_{O_local}) >
    end
end
for each p in [1..P] do
    if p != local then
        CR = CR ∪ < p, (W_{O_p} ∩ W_{I_local}) >
    end
end
```

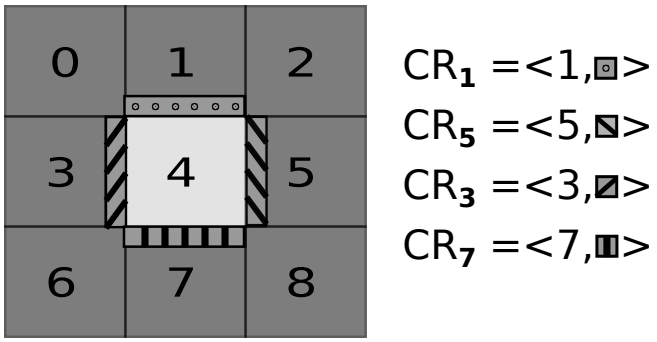

Figure 6. Calculation of receives according to algorithm 1 for Jacobi Solver.

process. As it is shown in Alg. 1, we can generate a loop in the code that traverses all the other $P-1$ processes, computing the intersections of the local process input and output working sets with the opposite sets in the remote process, generating an exact communication pattern in $O(P \times (k \log k))$. In the algorithm, communication patterns are represented by a set of comm-tuples (communication tuples). A comm-tuple $< p, W >$ contains: The index p of the process to which local data has to be sent to, or received from; and the working set W that identifies the domain of the data that should be transferred. The result of the algorithm includes up to two sets of comm-tuples for each remote process, one for the data to be sent, and another for the data to be received.

However, this algorithm is not valid for expressions with data-flow dependences. We consider data-flow dependences when a computation step need to wait for values of data updated at other process in the same computational

```
** Case 4: GAUSS SEIDEL SOLVER
1. While iterations < limit
     For i = 0 .. M.rows
       For j = 0 .. M.columns
          M[i][j] = ( M[i-1][j] + M[i+1][j]
           + M[i][j-1] + M[i][j+1] ) / 4;
```

Figure 7. Sequential algorithm expression with data-flow dependences.

```
** Case 4: GAUSS SEIDEL SOLVER
1. While iterations < limit
   1.1CW Data-flow Communication of stage (WIF_1.1, WO_1.1, M)
   1.1   For i = 0 .. M.rows
          For j = 0 .. M.columns
            M[i][j] = ( M[i-1][j] + M[i+1][j]
             + M[i][j-1] + M[i][j+1] ) / 4;
   1.1C  Communication stage (WI_1.1 \ WIF_1.1, WO_1.1, M)
```

Figure 8. Parallel algorithm expression with data-flow dependences.

stage. This situation appears for example in codes that parallelize a loop, but inherit form sequential semantics a loop-carried dependence due to a given expression. See an example of an algorithm with data-flow dependences in Fig 7. It is a simple Gauss Seidel solver for the Poisson Equation. In codes with such expressions, we use Alg. 2. In this case, we calculate a pattern to be executed before computation (CRF), and another to be executed after computation (CR, CS), see Fig 8.

The set of comm-tuples CRF contains the communications corresponding to the receives derived from the data-flow dependences. This pattern is executed before the computation to wait for the needed values. The set of comm-tuples CR contains the communications associated to all data that has to be received, excluding data corresponding to the data-flow dependences already involved in CRF. We obtain this information subtracting the working set of the former data-flow dependences, from the total input working set. This pattern is executed after the computation to relocate data before the next sequential computation.

Calculating these patterns at run-time and always just before computation allow us to support dynamic loops with conditions evaluated in terms of data values (e.g. convergence checking for the Jacobi solver (see Fig. 2)) unlike other parallelizing-compilers based on the polyhedral model (see e.g. [Bon13]).

3.3 Applying general model to Jacobi Solver

In this section we show how to apply the algorithms presented in section 3 for the illustrating example of the Jacobi 2-D solver.

In section 3.1 we show how to place the communications stages, and how to calculate the input and output Working set which are taken into account for each communication stage. With this information is possible to automatically calculate the communications. For the Jacobi solver example, the communication

ALGORITHM 2: General model to calculate communication patterns including wave-fronts

Input: Number of processes (P),
Working Set of outputs of each process ($W_{O_p}; p = 1..P$) at stage j,
Working Set of inputs of each process ($W_{I_p}; p = 1..P$) at stage j+1,
Working Set of inputs of which implies data-flow dependencies of each process ($W_{IF_p}; p = 1..P$) at stage j,
Local process id: *local*
Output: Pattern communication (Set to send (CS), Set to receive (CR) and Set to receive before computation (CRF) for each process)

for *each p in [1..P]* **do**
  if *p != local* **then**
    $CS = CS \cup < p, (W_{I_p} \cap WO_{local}) >$
  end
end
$W'_{I_{local}} = W_{I_{local}} \setminus W_{IF_{local}}$
for *each p in [1..P]* **do**
  if *p != local* **then**
    $CR = CR \cup < p, (W_{O_p} \cap W'_{I_{local}}) >$
    $CRF = CRF \cup < p, (W_{O_p} \cap W_{IF_{local}}) >$
  end
end

step added after Stage 1.2 must look forward W_I of the following computation phases, in this case, the phase 1.1 at the next iteration of the outer loop. In this phase, there is no dependences because of 1.2 stage writes only in its domain, and 1.1 stage reads also only in its domain. Then, there is no communication stage after computation stage 1.2. However, in the communication between the 1.1 and 1.2 stages this situation does not occur, so we have to apply the Alg. 1.

We show a pseudo-code of the communication stage 1.1C in figure 9. Here, input and output working sets are calculated with the functions named *calculateWI* and *calculateWO*, which are generated at compile-time from the access expressions in the sequential code. This functions are evaluated at runtime allowing the use of expressions dependent on runtime parameters.

According to Alg. 1 the first step is to calculate the set of comm-tuples to send (CS). In order to do this, we inspect all the other processes to know which one needs data that belongs to the output working set of the local process. Each process intersects its output working set with the input working set of the rest of the processes. When the intersection is not null, the process, which is being inspected, needs data from the local process to perform its next computations (see Fig.10). The data that exactly needs to be sent is the data in the shape delimited by the intersection previously computed. Thus, we add to CS the comm-tuple formed by the index of the process that will receive the data and the domain resulted from the intersection (see 1.1C.1 in figure 9).

```
** Case 1: JACOBI SOLVER
1. While not converge and iterations < limit
  1.1. For each i,j in M.domain
      M2[i][j] = M[i][j]
  1.1C  ** Communication stage
      1.1C.1   WO_1.1_local = calculateWO_1.1(local)
               For each p in Processors
                if p != local
                     WI_1.2_p = calculateWI_1.2(p)
                     W_Aux= intersect(WI_1.2_p, WO_1.1_local)
                     CS=union(CS,W_Aux,p)
      1.1C.2   WI_1.2_local = calculateWI_1.2(local)
               For each p in Processors
                if p != local
                     WO_1.1_p = calculateWO_1.1(p)
                     W_Aux= intersect(WI_1.2_local, WO_1.1_p)
                     CR=union(CR,W_Aux,p)

      1.1C.3   Execute(CS,CR)

  1.2. For each i,j in M.domain
        M[i][j] = ( M2[i-1][j] + M2[i+1][j]
                  + M2[i][j-1] + M2[i][j+1] ) / 4;
  1.2C  ** Communication NULL


  **  WI Calculation of stage 1.2
  calculateWI_1.2(process p)
     ** Return the domain assigned to process p
     shapeOtherP = hit_shape(p)
     ** Applying affine expressions
     Aux= union(
                Affine(shapeOtherP, 1, -1, 1, 0) // Dim 0, alpha 1, beta -1
                                                 // Dim 1, alpha 1, beta 0
                Affine(shapeOtherP, 1, 1, 1, 0)  // Dim 0, alpha 1, beta 1
                                                 // Dim 1, alpha 1, beta 0
                Affine(shapeOtherP, 1, 0, 1, -1) // Dim 0, alpha 1, beta 0
                                                 // Dim 1, alpha 1, beta -1
                Affine(shapeOtherP, 1, 0, 1, 1)  // Dim 0, alpha 1, beta 0
                                                 // Dim 1, alpha 1, beta 1
     )

     return Aux
```

Figure 9. Parallel Algorithm of Jacobi solver after applying Alg 1.

The second part of Alg. 1 calculates the data which the local process must receive before performing its next computation (set of comm-tuples to receive, CR). In the same way that CS calculation, we inspect all the other processes to know which ones have in their W_O data needed by the local process. Each process intersects its input working set with the output working set of the other processes. When the intersection is not null, the process, which is being inspected, has data which the local process needs to perform its next computation (see Fig.6). The data that needs to be received locally is the data in the shape delimited by the intersection previously performed. Thus, we add to CR the comm-tuple formed by the index of the process that will send the data to the local process and the domain resulted from the intersection (see 1.1C.2 in figure 9).

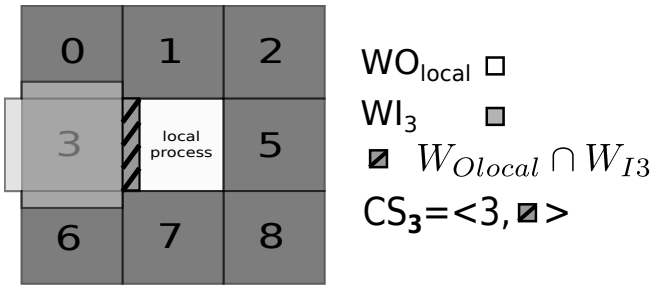

Figure 10. Calculation of sends from P_4 (local process) to P_3 according to Alg. 1, for the Jacobi Solver.

The execution of the communications indicated by CS and CR allows to relocate the data before the next sequential computation (see 1.1C.3 in figure 9).

4 Conclusions

This report describes a transformation technique for codes based on affine expressions to generate communications code for distributed-memory systems, generating exact aggregated communications at the distributed level. This technique generalizes and improves previous methods proposed to compute aggregated communications. It is based on intersections of remote and local footprints at run-time in terms of the mapping functions chosen. Future work includes the improvement of this technique to support periodic and different types of non-affine expressions.

Acknowledgement

This research has been partially supported by the Ministerio de Economía y Competitividad (Spain) and the ERDF program of the European Union: CAPAP-H5 network (TIN2014-53522), MOGECOPP project (TIN2011-25639), HomProg-HetSys project (TIN2014-58876-P); the Junta de Castilla y León (Spain): ATLAS project (VA172A12-2).

References

[Bas04] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proc. PACT'04*, pages 7–16. ACM Press, 2004.

[BGH+06] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua, and Christoph von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proc. of the ACM SIGPLAN PPoPP*, pages 48–57, New York, New York, USA, 2006. ACM.

[Bon13] U. Bondhugula. Compiling affine loop nests for distributed-memory parallel architectures. In *Proc. SC'2014*, Denver, CO, USA, 2013. ACM.

[FGEL13] J. Fresno, A. Gonzalez-Escribano, and D.R. Llanos. Blending extensibility and performance in dense and sparse parallel data management. *IEEE TPDS*, 25(10):2509 – 2519, Oct 2013.

[GETFL13] A. Gonzalez-Escribano, Y. Torres, J. Fresno, and D.R. Llanos. An extensible system for multilevel automatic data partition and mapping. *IEEE TPDS*, 25(5):1145–1154, 2013. (doi:10.1109/TPDS.2013.83).

[MFGEL15] A. Moreton-Fernandez, A. Gonzalez-Escribano, and D.R. Llanos. On the run-time cost of distributed-memory communications generated using the polyhedral model (to appear). In *Proc. HPCS'2015*, 2015.