Departamento de
Informática
Universidad de Valladolid

Computer Science Department
University of Valladolid
Valladolid - Spain

A New High Level Parallel Portable Language for hierarchical systems in Trasgo

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 Currently, the generation of parallel codes which are portable to different kinds of parallel computers is a challenge. Many approaches have been proposed during the last years following two different paths. Programming from scratch using new programming languages and models that deal with parallelism explicitly, or automatically generating parallel codes from already existing sequential programs. Using the current main-trend parallel languages, the programmer deals with mapping and optimization problems that forces to take into account details of the execution platform to obtain a good performance. In code generators that automatically transform sequential programs to parallel ones, programmers cannot control basic mapping decisions, and many times the programmer needs to transform the original sequential code to expose to the compiler information needed to leverage important optimizations.
This paper presents a new high-level parallel programming language named CMAPS, designed to be used with the Trasgo parallel programming framework. This language provides a simple and explicit way to express parallelism at a highly abstract level. The programmer does not face decisions about granularity, thread management, or interprocess communication. Thus, the programmer can express different parallel paradigms

in an easy, unified, abstract, and portable form. The language supports the necessary features imposed by transformation models such as Trasgo, to generate parallel codes that adapt their communication and synchronization structures for target machines composed by mixed distributed- and shared-memory parallel multicomputers.

Technical Report No. IT-DI-2015-0001

1 Introduction

It is increasingly interesting to generate application programs with the ability of automatically adapt their structure and load to any given target system. Using current main-trend parallel programming technology for this purpose is challenging. The polyhedral model is one of the most promising techniques for frameworks which automatically generate optimized lower-level parallel code from existing sequential programs. It 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. However, it does not support dynamic loops dependent on information not known at compile-time. On the other hand, many successful parallel programming models and tools that explicitly deal with parallelism have been proposed. Message-passing paradigms (e.g. MPI libraries) have been 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. Many parallel programming models like PGAS (Partitioned Global Address Space) languages (Chapel, X10, or UPC), present a middle point approach by explicitly managing local and global memory spaces. The PGAS language more related to our work is Chapel [CDIC10]. It proposes a separation of domain and mapping modules to work with distributed arrays. But, the best aggregated-communication methods presented so far for Chapel abstractions are restricted to specific operations, or aggregated-communication domain mapping properties.

Thus, the parallel applications 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. Moreover, many of these decisions may change for different machine details or structure, or even with data sizes. Productive parallel-software development needs a common approach at the programming level, to simplify the tasks of implementing, testing, and debugging, independently of the machine details.

In this paper we present CMAPS, a new high-level parallel programming language for the Trasgo framework [GEL11]. This language supports a wide range of parallel structures and applications. The programs express coordination at an abstract level. The programmer reasons in terms of logical processes using a global memory space, not facing decisions about granularity, thread management, or interprocess communication. CMAPS approach presents several advantages with regard to: (1) polyhedral frameworks which work with sequential codes as input [Bon13,CG06,KPP$^+$15], as it supports dynamic loops with conditions dependent on expressions involving data-values, or runtime parameters and, (2) explicit parallel languages, because it makes transparent to the programmer the details related to the lower-level programming model. Some of these details are the integration of mapping techniques or, the adaptation of the code to the architecture and execution platform. Moreover, comparing with traditional parallel programming models for distributed-memory machines such as MPI, CMAPS approach reduces the code complexity.

We present the design guidelines of CMAPS in terms of the requirements and capabilities of the Trasgo framework to generate lower-level code that adapts their communication and synchronization structures to the target machine. In particular, these guidelines impose the inclusion of information needed to automatically generate code that computes exact aggregated communications for target machines including distributed-memory architectures.

The rest of the paper is organized as follows: Section 2 presents the Trasgo model and their tools. Section 3 describes the new programming parallel language. Section 4 presents the conclusions and future work.

2 Trasgo framework

The Trasgo model [GEL11] proposes the use of a high-level structured and abstract representation of the parallel algorithms. It uses a restricted synchronization model (nested-parallelism) at the higher level, letting the transformation system to generate more efficient and less synchronized parallel structures at the lower level. The original model is based on the SP (Series-Parallel) process model [Gem97], and data-distribution algebras, providing clear and well-defined semantics [LW98], and allowing hierarchical compositions. The model is free of race conditions, and unexpected stochastic behaviors or dead-locks. The high-level code uses a global view approach with the possibility of decompose it hierarchically. The semantics provide clear synchronization points and hierarchical global states that simplify testing and debugging.

Figure 1 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 to an internal rep-

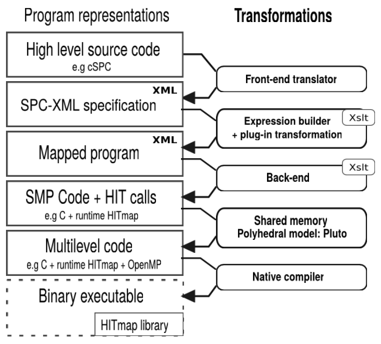

Figure 1. Structure of the Trasgo transformation framework.

resentation in XML. An XML representation has been chosen due to the standard and powerful tools that exist to identify and locate document features (XPath), and to apply document transformations (XSLT). These technologies can be used to write in a compact form code transformation modules. The main part of the transformation layer is oriented to convert the global address space into a partitioned address space. It analyses data dependencies and builds expressions to compute at run-time the communications needed across virtual processes, in terms of the results of mapping and layout functions. The transformed code is rewritten by a back-end that generates C code with calls to the Hitmap run-time library [GETFL13]. The resulting sequential code generated for the local distributed process is finally filtered through polyhedral tools (Pluto compiler [BHRS08]) to generate tiled and optimized parallel code for shared-memory using OpenMP (the methodology used to integrate these techniques at the Hitmap level was described in [MFGEL14]). Finally, the code is compiled with a native C compiler.

3 The new parallel programming language CMAPS

In this section we describe the proposed parallel language that will be used as input in the Trasgo framework. This language will allow the programmer to express parallel algorithms in terms of abstract decomposition and mapping techniques.

3.1 Design principles

We present bellow guidelines for the design of an input language for the Trasgo framework.

1. The input language will be a coordination language. Sequential code will be expressed with a traditional programming language and using runtime library calls (e.g. Hitmap [GETFL13]) to manage the access to the data structures. Extensions to a traditional sequential language (new primitives, programming structures and modifiers) should be used to express the coordination between sequential tasks. Functions containing coordination primitives will be clearly distinguished from the ones containing sequential code. Inside them, it will be only possible to execute data modifications through calls to sequential functions.
2. The input language will use a primitive (with clauses) to annotate each section that we will be executed in parallel. The primitive should allow to indicate an arbitrary number of logical processes, in terms of constants or expressions, in terms of parameters or the number of elements in data structures. This primitive could be nested as many times as needed, even in a recursive way. Inside the scope of the primitive, a mechanism to associate logical processes to calls to other coordination or sequential functions should be provided. This primitive will imply a logical synchronization point of the processes after the computation performed in parallel.

3. The sequential functions will be designed to deal with elements of arbitrary grain. Thus, it avoids the programmer to take decisions about the problem granularity according to the machine capabilities.
4. The language should provide a mechanism to invoke modules that implement partition policies. The inputs will be index spaces. The output will be a map of indexes to processes. Each element of the input domain will be assigned to one and only one virtual process, and each virtual process to one and only one real processing element. The number of virtual processes will be chosen at run-time. This map can be used to: (1) perform a partition, distribution and allocation of data structures or, (2) group and schedule logical processes into virtual processes. The code should be independent of the mapping (partition, distribution, allocation) policies chosen. With this technique it will be possible to change the partition or allocation policy without modifying any other part of the code.
5. The computation will have a single global state and a unique logical process before the first parallel primitive. When the parallel tasks are launched, each one has a local copy of the global state. In the logical synchronization point at the end of the parallel computation the global state is consolidated again. To generate this global state the parallel primitive must provided a way for reducing correctly different values found in the replicated variables of each logical process, into the global state.
6. The language should provide a mechanism to analyse the data dependences between the function calls which are into the scope of different parallel parts. To find data dependencies can be a complex task at compile time. The language should use annotations in the function definition to indicate the input/output role of each parameter in the function interface. This information will be used to obtain the data dependencies in the Trasgo framework.

3.2 Study cases

To show the features of CMAPS in real programs we will use four cases of study whose sequential pseudo code descriptions are presented on Fig.2. Each one presents different features and challenges for a parallel programming model. The first one is a *Jacobi solver*. It is a PDE solver using a Jacobi iterative method to compute the heat transfer equation in a discretized two-dimensional space. It is implemented as a cellular automata or stencil computation. On each iteration, each matrix position or cell is updated with the previous values of the four neighbors. The second one is a *Gauss-Seidel* program that computes the same heat transfer equation described above, but using a Gauss-Seidel iterative method. In this method, the convergence is accelerated using values already computed during the current time-step iteration following sequential semantics. The method simply uses one matrix, with no copy for the old values. Thus, when using the neighbor values of the upper and left matrix positions, values already updated are used. The third one is a *Classical Matrix Multiplication*. It is the typical sequential implementation of the matrix multiplication with three nested loops. The fourth and last one is *Cannon's Algorithm* [Can69] for

```
** Case 1: JACOBI SOLVER
1. While not converge and iterations < limit
     converge = true
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;
1.3. If |M2[i][j] -M[i][j]| > threshold
         converge = false

** Case 2: 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;

** Case 3: MM CLASSICAL ALGORITHM
   For i = 0 .. C.rows
       For j = 0 .. C.columns
            For k = 0 .. A.columns
               C[i][j] = C[i][j] + A[i][k]*B[k][j]

** Case 4: MM CANNON'S ALGORITHM
1. Split A,B,C in k x k blocks
   AA=Blocking(A), BB=Blocking(B), CC=Blocking(C)
2. Initial data alignment:
2.1. For each i in AA.rows
     Circular shift: Move AAi,j to AAi,j-i
2.2. For each j in BB.columns
     Circular shift: Move BBi,j to BBi-j,i
3. For s = 1 .. k
   3.1. CCi,j = CCi,j + AAi,j * BBi,j
   3.2. For each i in AA.rows
        Circular shift: Move AAi,j to AAi,j-1
   3.3. For each j in BB.columns
        Circular shift: Move BBi,j to BBi-1,j
```

Figure 2. Algorithm expressions of four study cases.

matrix multiplication. It works with a partition of the matrices in $k \times k$ pieces, requiring no more than one local piece of the same matrix at the same time, and using a simple circular block shift pattern to move data across processes. Each matrix-block product is computed using the classical sequential algorithm.

```
/* Jacobi Solver (Poisson equation): Function to update one cell element */
void updateCell( in double up, in double down, in double left, in double right,
        inout double result, out double diff ) {
 double old = *result;
 *result = ( up + down + left + right ) / 4 ;
 *diff = fabs( *result - old );
}

/* Jacobi Solver (Poisson equation): Parallel solver */
coordination void jacobiSolver( inout tile double M[][],
                                in int limit, in double threshold) {
 double inside[][] = M[1:$-1][1:$-1];
 Map distribution = Map( inside.shape, blocks, rectangular2D ) );
 ArrayMap( inside, distribution );

 double diff, maxDiff;
 loop( i in [1:limit] and maxDiff > threshold ) {
  resetDiff( maxDiff );
  parallel ( distribution ) {
   do: updateCell( M[ pidx(0)-1 ][ pidx(1) ],
     M[ pidx(0)+1 ][ pidx(1) ], M[ pidx(0) ][ pidx(1)-1 ],
     M[ pidx(0) ][ pidx(1)+1 ], M[ pidx(0) ][ pidx(1) ],
     diff );
   reduce: MAX( diff, maxDiff );
} } }
```

```
/* Poisson equation: Function to update one cell element */
void updateCell( in double up, in double down, in double left, in double right,
               out double result ) {
      *result = ( up + down + left + right ) / 4 ;
         }

/* Gauss-Seidel Parallel Solver */
coordination void gaussSolver( inout tile double M[][], in int limit ) {
      double inside[][] = M[1:$-1][1:$-1];
      Map distribution = Map( inside.shape, blocks, topRectangular2D ) );
      ArrayMap( inside, distribution );

     loop( i in [1:limit] ) {
        parallel ( distribution ) {
           do :  waitflow( M[ pidx(0)-1 ][ pidx(1) ], M[ pidx(0) ][ pidx(1)-1 ] )
                 updateCell( M[ pidx(0)-1 ][ pidx(1) ], M[ pidx(0)+1 ][ pidx(1) ],
                           M[ pidx(0) ][ pidx(1)-1 ], M[ pidx(0) ][ pidx(1)+1 ],
                           M[ pidx(0) ][ pidx(1) ] );


} } }
```

Figure 3. CMAPS code for two Stencils algorithms: Jacobi solver and Gauss-Seidel

```
/* Parallel Block Matrix Multiplication: Classical algorithm */
coordination void mmProductClassical( in tile double A[][], in tile double B[][],
              out tile double C[][] ) {
 Map mC = Map( C.shape, blocks, topRectangular2D ) );
 ArrayMap( C, mC );
 ArrayMap( A, Map( A.shape, blocks, topRectangular2D ) ) );
 ArrayMap( B, Map( B.shape, blocks, topRectangular2D ) ) );

 parallel ( mC ) {
  do: mmProductSeq_1( A[pidx(0)][0:$], B[0:$][pidx(1)], C[pidx(0)][pidx(1)] );
} }

/* Parallel Block Matrix Multiplication: Cannon's algorithm */
coordination void mmProductCannons( in tile double A[][], in tile double B[][],
              out tile double C[][] ) {
        Map mC = Map( C.shape, blocks, topSquare ) );
        ArrayMap( C, mC );
        ArrayMap( A, Map( A.shape, blocks, topSquare ) ) );
        ArrayMap( B, Map( B.shape, blocks, topSquare ) ) );

        loop( i, [ 0 : max( mC.size(0), mC.size(1) ) ] ) {
             parallel ( mC ) {
                  do: mmProductSeq_2(
                          dmap(A, mapidx(0),cyc(( mapidx(1) - mapidx(0) - i)) ),
                          dmap(B, cyc(( mapidx(0) - mapidx(1) - i )),mapidx(1) ),
                          dmap(C, mapidx(0), mapidx(1)) );
} } }

/* Parallel Block Matrix Multiplication: Hierarchical composition */
coordination void mmProductCannons( in tile double A[][], in tile double B[][],
              out tile double C[][] ) {
        Map mC = Map( C.shape, blocks, topSquare ) );
        ArrayMap( C, mC );
        ArrayMap( A, Map( A.shape, blocks, topSquare ) ) );
        ArrayMap( B, Map( B.shape, blocks, topSquare ) ) );

        loop( i, [ 0 : max( mC.size(0), mC.size(1) ) ] ) {
             parallel ( mC ) {
                  do: mmProductClassical(
                         dmap(A, mapidx(0),cyc(( mapidx(1) - mapidx(0) - i)) ),
                         dmap(B, cyc(( mapidx(0) - mapidx(1) - i )),mapidx(1) ),
                         dmap(C, mapidx(0), mapidx(1)) );
} } }
```

Figure 4. CMAPS code for a multilevel Matrix Multiplication.

3.3 Notations and definitions

A Trasgo input programming language can be designed in several ways, as far as it complies with the Trasgo model semantic (which derives in the guidelines presented on section 3.1). We have designed a coordination language extension of classical C language named CMAPS. CMAPS has been created to express parallel algorithms in a simple, explicit and intuitive way for C programmers.

Domains, mapping policies and data structures The CMAPS language provides the native data types of the original sequential language (C), and *tile* types for more complex data structures, such as arrays. Manipulation of native data types is direct. Data in tiles will be managed in the sequential functions using the Hitmap library tile access functionalities [GETFL13]. Domain declarations and subselection of tile domains, are expressed with an extended C array notation similar to the Fortran90 colon notation inside square brackets *[begin:end:stride]*.

The language provides a *Map* constructor (see line 13 of Jacobi solver code in Fig. 3). It receives three parameters: An index domain to be mapped, the name of a partition and layout technique, and the name of a virtual topology building policy. Layout and topology policies are plug-in modules in the runtime system [GETFL13]. *Map* objects transparently map an index domain to the devices of a target system (guideline 4). The *ArrayMap* function is used to map a tile to virtual processes according to the results in the *Map* object. The language supports a symbol to represent the last index of the global index space in a dimensional domain (*$*). For example, in the Jacobi and Gauss examples in Fig. 3, the inner part of a matrix (without the border rows and columns) is selected.

Functions CMAPS supports two different kinds of functions, sequential and coordinated. Both will use a compulsory modifier for the formal parameters that makes explicit their input/output behaviour (see line 2 of Jacobi solver code of Fig. 3). These modifiers will be used to derive the dependencies between parallel tasks (guideline 6). All data modification statements should be encapsulated into classical sequential C void functions, that are called from the coordination code. Each call to a sequential function is done in a logical process. Coordination functions are the functions where parallelism and synchronization can be expressed. They are specified adding the *coordination* modifier in its definition (see line 10 of Jacobi solver code of Fig. 3). Nested parallelism is exploited by encapsulating each parallelism level in a coordination function (guideline 1).

Trasgo allows to generate codes with several levels of parallelism. Figure 4 shows the implementation of the two matrix-multiplication algorithms. Cannon's algorithm can be used at the upper level to reduce the memory usage in a distributed memory platform, and the classical one at the shared memory level for reducing synchronization time. Making a hierarchical composition of the two algorithms in CMAPS is trivial. Simply substituting the name of the sequential function in the Cannon's algorithm (see Fig.4 line 39) by the name of the Classical's parallel algorithm also written in CMAPS.

Coordination primitives CMAPS supports into the coordination functions several kind of primitives. An unified *parallel* primitive, iterative primitives (*loop*, *while*), and conditional primitives (*if*, *else*). Statements such as assignments are not allowed in the predicates or bodies of the coordination primitives. The *loop* primitive substitutes the functionality of the classical *for* primitive that are controlled by counter, and added generic conditions with a restricted syntax (see line 17 of Jacobi solver in Fig 3).

The *parallel* primitive performs computations in parallel (guideline 2). The parallel primitive receives a *Map* object as parameter. It spawns as many logical processes as indicated in the domain used to build the Map object and they are assigned to virtual processes according to the information contained in the *Map* object. Virtual processes are automatically scheduled to real processors following the policies used to build the *Map* object. The primitive contains *do* clauses. In these clauses scope we can write coordination code with function calls to be executed by the logical processes (see line 20 of Jacobi solver in Fig.3). They can be followed by optional *reduce* clauses. Each logical process works in a virtual copy of the tiles. The transformation system is responsible of generating copies of data parts, and temporal buffers, if needed to preserve the parallel semantics; even when several logical processes are mapped to the same real process, and consequently should be executed sequentially in the same real process scope (guideline 5). The *do* clauses may be followed by *waitflow(...)* clauses to express data-flow restrictions during the parallel execution. For example, in the Gauss-Seidel study case each logical process needs to wait for the data produced by its upper and left logical processes (see line 15 of Fig. 3).

The calls in the *do* clauses of a *parallel* primitive can use two types of indexes to build subdomains expressions and select the subtiles used as real parameters in the function calls. They both allow to deal with locality in different ways. First, *pidx(<dim>)*, which is the dimensional index of the logical process. Logical processes are grouped automatically in virtual processes by Trasgo framework, avoiding the programmer to take decisions about the problem granularity (guideline 3). The second is *midx(<dim>)*. The expression *midx(<dim>)* represents the index of the virtual process in the active processes topology in a chosen dimension. It is possible to use the function *cyc()* in expressions built with *midx(<dim>)*. It returns the virtual process dimensional index in a periodic way in the active processes topology. The function *dmap(<tile>, <processes>)* returns the subdomain of the tile specified as the first parameter, which is mapped to the virtual process indicated as the second parameter (see line 23 of Cannon's matrix multiplication CMAPS code). Loop indexes and invariant parameters can also be used in expressions involving the *pidx(<dim>)* and *midx(<dim>)* functions.

3.4 Example

This section describes the way to use CMAPS to derive a parallel code from a given algorithm already expressed in sequential C code. To accomplish this task we use an illustrating example, the Jacobi 2-D solver. We show a simple

```
 1  void updateCell( double up, double down, double left, double right, double *result,
 2                    double *diff){
 3       double old= *result;
 4       *result = (up + left + right + down)/4;
 5       *diff = fabs( *result - old );
 6  }
 7  /** Sequential Jacobi solver */
 8  void jacobiSolver( double a[][], double b[][], int limit){
 9    int maxDiff, t, i, j;
10    for (t=0; t<limit && maxDiff < threshold; t++) {
11       maxDiff=0;
12       for (i=1; i<N-1; i++)
13          for (j=1; j<N-1; j++)
14                  a[i][j]=b[i][j];
15       for (i=1; i<N-1; i++)
16          for (j=1; j<N-1; j++) {
17              updateCell( a[i+1][j], a[i-1][j], a[i][j+1], a[i][j-1], &(b[i][j]),
18                                         &diff);
19              if (diff > maxDiff) maxDiff=diff;
20       }
21
22  } }
```

Figure 5. Sequential code of Jacobi solver

sequential C implementation of the Jacobi solver in Fig. 5. The first step in the main *jacobiSolver* function is to declare the necessary variables. Thus, in CMAPS (code of Jacobi solver of Fig. 3) we do the same. In CMAPS, we have to distribute the arrays to perform a parallel computation. To distribute the domain of the array, we use the *Map* constructor. We have to choose the index domain that we want distribute. In this case, it is the domain of the whole array except its borders $(1 : \$ - 1)$. We choose a plug-in layout that performs a classical partition by balanced two-dimensional blocks, and a plug-in to generate a rectangular 2D topology. We use the *ArrayMap* function to allocate the array across the processor topology using the domain partition resulting of the Map. The sequential program executes some loop iterations until it achieves a limit or until it reaches the convergence criteria (line 10 of Fig. 5). In CMAPS we use the *loop* primitive.

On each iteration, each position of the array will be updated with the previous values of its neighbors. The sequential code first updates a copy of the original array values. Then, the inner array elements are traversed. The sequential function *updateCell* is executed for each element using the old values contained in the copy of the array and checking if the element has converged.

On the other hand, CMAPS does not need a copy of the old values. In CMAPS logical processes work with a virtual copy of the global state. So, we only have to perform the computation (using the *updateCell* function) on one element of the array in each logical process. Each logical process can access to the old values of its neighbors in his local state independently of the real execution order. We accomplish this using the *parallel* primitive. We launch as many logical process as indicated by the shape or domain used to build the *Map* object. In this case, we launch a logical process for each element in the array except for the global borders (see line 12 of code CMAPS of Jacobi solver). Each logical process executes the *updateCell* function updating an element, using its four neighbours (see line 20-22 of code CMAPS of Jacobi solver). The convergence check is performed using the *reduce* clause. This reduce chooses the maximum value of the evaluated convergence check of each element. An internal copy of the old values will be managed automatically by Trasgo if needed to preserve the CMAPS parallel semantics.

4 Conclusions

This paper describes a new high level parallel programming language, CMAPS. It is designed to be a front-end language for a parallel code generation framework named Trasgo. It allows to express coordination at a highly abstract level, supporting a wide range of parallel structures and applications. We describe the design principles needed in a Trasgo input language and we present our approach to create CMAPS, an extension of the classical C language that complies with the required principles. We review the syntax of CMAPS showing examples of real study cases that show the features of the language, how it manage domains, mappings, and data structures in parallel, in an abstract way.

Acknowledgements

This research has been partially supported by MICINN (Spain) and ERDF program of the European Union: HomProg-HetSys project (TIN2014-58876-P), CAPAP-H5 network (TIN2014-53522-REDT), and COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS).

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.

[BHRS08] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

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

[Can69] L.E. Cannon. *A cellular computer to implement the kalman filter algorithm.* Doctoral dissertation, Montana State University Bozeman, 1969.

[CDIC10] B.L. Chamberlain, S.J. Deitz, D. Iten, and S-E. Choi. User-defined distributions and layouts in Chapel: Philosophy and framework. In *2nd USENIX Workshop on Hot Topics in Parallelism*, June 2010.

[CG06] Michael Claßen and Martin Griebl. Automatic code generation for distributed memory architectures in the polytope model. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 7–pp. IEEE, 2006.

[GEL11] A. Gonzalez-Escribano and D.R. Llanos. Trasgo: A nested-parallel programming system. *The Journal of Supercomputing*, 58(2):226–234, 2011.

[Gem97] A.J.C. van Gemund. The importance of synchronization structure in parallel program optimization. In *Proc.* 11th *ACM ICS*, pages 164–171, Vienna, Jul 1997.

[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).

[KPP$^+$15] Martin Kong, Antoniu Pop, Louis-Noël Pouchet, R Govindarajan, Albert Cohen, and P Sadayappan. Compiler/runtime framework for dynamic dataflow parallelization of tiled programs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):61, 2015.

[LW98] K. Lodaya and P. Weil. Series-parallel posets: Algebra, automata, and languages. In *Proc. STACS'98*, volume 1373 of *LNCS*, pages 555–565, Paris, France, 1998. Springer-Verlag.

[MFGEL14] A. Moreton-Fernandez, A. Gonzalez-Escribano, and D.R. Llanos. Exploiting distributed and shared memory hierarchies with Hitmap. In *Proc. HPCS'2014*, pages 278–286, Bologna (Italy), 2014.