Proceedings of the Federated Conference on Computer Science and Information Systems pp. 705–708
DOI: 10.15439/2016F48
ACSIS, Vol. 8. ISSN 2300-5963

An Iteration Space Visualizer for Polyhedral Loop Transformations in Numerical Programming

Marek Palkowski, Wlodzimierz Bielecki
West Pomeranian University of Technology in Szczecin
ul. Zolnierska 49, 71-210 Szczecin, Poland
Email: mpalkowski@wi.zut.edu.pl, wbielecki@wi.zut.edu.pl

***Abstract*—An iteration space visualizer is presented to analyze parallelism in loop nests including parallelism in tiled code of numerical programs. The tool visualizes exact data dependences available in arbitrarily nested loops as well as tiles generated with TRACO by means of the transitive closure of a loop nest dependence graph. Various graphical operations such as rotation, zooming, coloring and filtering allow for a detailed examination of dependences, iteration space slices, and shapes of generated tiles. The visualizer is a built-in TRACO module which collects results generated with TRACO and it is launched automatically when TRACO finishes code generation. The visualizer helps high-performance application developers discover parallelism available in loop nests and analyze tiled code produced by means of the polyhedral model.**

I. Introduction

AUTOMATIC parallelization in numerical programs has been the topic of research for many decades. In the majority of cases, the techniques focus on two basic steps: dependence analysis and program transformations. Despite the great steps forward in this area, sophisticated dependences, the construction of loop transformations, and statement instance mappings are beyond what the programmer is able to see at first glance [1].

This paper focuses on a graphical support for the automatic parallelizer and optimizer, TRACO, – the source-to-source compiler based on the transitive closure of a dependence graph to transform affine loop nests. A proposed visualizer assists both experts and non-expert programmers to understand the results generated with TRACO [2] and code generated by it.

The TRACO visualizer is based on Python scripts which use wrappers to the Integer Set Library (ISL) [3] and matplotlib [4]. ISL allows users to manipulate on sets and maps while matplotlib is a plotting library to produce 2-D figures and 3-D interactive projections. The tool collects and visualizes data generated by TRACO, for example, dependences, code fragments to be executed in parallel, and shapes of tiles.

The remainder of the paper is organized as follows. The next section discusses the basics of the polyhedral model, iteration space dependence graph and operations to manipulate maps and sets. Section 3 briefs algorithms implemented in TRACO. Section 4 explores visualizing functions and theirs capabilities. Section 5 introduces related work. The last section concludes the paper.

II. Background

In this paper, we deal with affine loop nests where, for given loop indices, lower and upper bounds as well as array subscripts and conditionals are affine functions of surrounding loop indices and possibly of structure parameters (defining loop index bounds), and the loop steps are known constants.

To implement algorithms, we have chosen the dependence analysis proposed by Pugh and Wonnacott [5], where dependences are represented by dependence relations. A dependence relation is a tuple relation of the form [*input list*]→[*output list*]: *formula*, where *input list* and *output list* are the lists of variables and/or expressions used to describe input and output tuples, and *formula* describes the constraints imposed upon input and output lists and it is a Presburger formula built of constraints represented by algebraic expressions and using logical and existential operators [5].

Standard operations on relations and sets are used, such as intersection ($\cap$), union ($\cup$), difference (-), domain (dom *R*), range (ran *R*), relation application (S'= *R*(*S*): $e' \in S'$iff exists *e* s.t. $e \rightarrow e' \in R$, $e \in S$). In detail, the description of these operations is presented in [5], [3].

The positive transitive closure for a given relation *R*, R^+, is defined as follows [6] $R^+ = \{e \rightarrow e' : \ e \rightarrow e' \in R \vee \exists e'' s.t. \ e \rightarrow e'' \in R \wedge e'' \rightarrow e' \in R^+\}$. It describes which vertices e' in a dependence graph (represented by relation *R*) are connected directly or transitively with vertex *e*. Transitive closure, R^*, describes the same connections in a dependence graph (represented by R) that R^+ does plus connections of each vertex with itself.

In sequential loop nests, the iteration i executes before j if i is *lexicographically less* than j, denoted as $i \prec j$, i.e., $i_1 < j_1 \vee \exists k \geq 1 : i_k < j_k \wedge i_t = j_t, for \ t < k$.

An ultimate dependence source is a source that is not the destination of another dependence. Set including all dependence sources,S_{UDS} is calculated as follows: $S_{UDS} = domain(R) - range(R)$, where a dependence relation *R* describes all the dependences in a loop nest.

An (iteration-space) slice is defined as follows. Given a dependence graph defined by a set of dependence relations, a slice *S* is a weakly connected component of this graph, i.e., a maximal sub-graph such that for each pair of vertices in the sub-graph there exists a forward or backward path.

Let IS denotes the loop nest iteration space. A function is called a schedule if it maps each iteration of IS onto another

space so that all data dependences available in the loop nest are preserved. The schedule that maps every $x \in IS$ to the first possible time step allowed by the dependences is called the free schedule.

III. The TRACO Compiler

TRACO allows us to generate parallel code. Parallelization is based on extracting synchronization-free slices or producing and applying the free-schedule. An approach to extract synchronization-free slices takes two steps [2]. First, for each slice, a representative statement instance is defined (it is the lexicographically minimal statement instance from all the ultimate sources of a slice). Next, slices are reconstructed from their representatives and code scanning these slices is generated.

In order to find representatives of slices, we build a relation, R_{USC} that describes all pairs of the ultimate dependence sources being transitively connected in a slice. The relation is constrained with the intersection of the sets $R*(e)$ and $R*(e') : (R*(e) \cap R*(e'))$ which guarantees that vertices e and e' are transitively connected, i.e., they are the sources of the same slice.

Next, set, S_{repr}, containing representatives of each slice is found as S_{repr} = S_{UDS} - range(R_{USC}). Then the remaining sources of this slice can be found by applying the relation (R_{USC})* to set S_{repr}. A set, representing slice elements, is formed by applying R* to the sources of a slice. To generate code, we apply the CLooG library [7] or ISL [3] to the set comprising statement instances of independent slices.

To parallelize loop nests which expose a single synchronization-free slice, time partitioning is applied. The algorithm, presented in paper [8], allows us to generate the free schedule and next fine-grained parallel code; all statement instances of a time partition can be executed in parallel, while partitions are enumerated sequentially.

Given relations R, representing all dependences in a loop nest, we calculate R^k, where $R^k = \underbrace{R \circ R \circ ... R}_{k}$, "$\circ$" is the composition operation. Given set *UDS* comprising all loop nest statement instances that are ready to execution at time *k*=0, each vertex, belonging the set S_k = $R^k(UDS)$ - $R^+ \circ R^k(UDS)$, is connected in the dependence graph, defined by relation *R*, with some vertex(ices) represented by set *UDS* with a path of length *k*. Hence at time *k*, all the statement instances belonging to the set S_k can be scheduled for execution and it is guaranteed that *k* is as few as possible.

Tiling is a very important iteration reordering transformation for both improving data locality and extracting loop nest parallelism. TRACO allows users generate parallel tiled code by means of algorithms based on the transitive closure of a dependence graph [2]. First, we form rectangular set *TILE*(***II***,***B***) including iterations belonging to a parametric tile as follows *TILE*(***II***, ***B***) = $\{[\boldsymbol{I}] \mid \boldsymbol{B}*\boldsymbol{II} + \boldsymbol{LB} \leq \boldsymbol{I} \leq \min(\boldsymbol{B}*(\boldsymbol{II}+\mathbf{1}) + \boldsymbol{LB} - \mathbf{1}, \boldsymbol{UB})$ AND $\boldsymbol{II} \geq 0\}$, where vectors ***LB*** and ***UB*** include the lower and upper loop index bounds of an original loop nest, respectively; diagonal matrix ***B*** defines the size of a rectangular original tile; elements of vectors ***I*** and ***II*** represent the original loop nest indices and the identifiers of tiles, respectively; **1** is the vector whose all elements are equal to 1.

TRACO, instead of program transformations represented by a set of affine functions, one for each statement, uses the transitive closure of a loop nest dependence graph to carry out corrections of original rectangular tiles so that all dependences of the original loop nest are preserved under the lexicographic order of target tiles. This may lead to changing shapes of original rectangular tiles; target tiles can be of an arbitrary shape which is affected with dependences available in a loop nest. Recognizing shapes from a mathematical representation of target tiles can be difficult. The visualizer helps recognize what are tile shapes. After code generation, the visualizer forms a graphical representation of the iteration space, dependences, and target tiles.

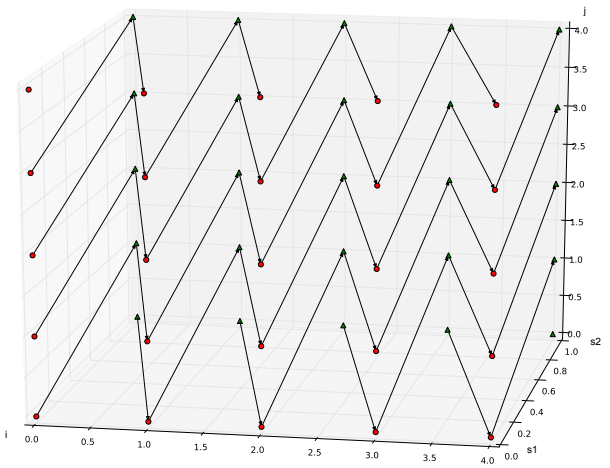

Fig. 1. Iteration space and dependences of Example 1 (n=4); statement instances are marked with the red circles and green triangles.

IV. Applying the TRACO Visualizer

Let us consider the example from paper [9] presented below:

```
// Example 1.
for(i=0; i<=n; i++)
 for(j=0; j<=n; j++){
  a[i][j] = a[i][j] + b[i-1][j];//s1
  b[i][j] = a[i][j-1 * b[i][j]; //s2
}
```

Figure 1 shows the iteration space and dependences for Example 1 generated with the visualizer. Analyzing this figure, we can discover that coarse-grained parallelism represented with synchronization-free slices is available in the considered loop nest (9 threads when n=4), but slices are load imbalanced. We also can see that there exist fine-grained parallelism, i.e., exist time partitions: for each value of index j, we can execute all instances of statement s2 in parallel for all values of index i, then all instances of statement s1. Visualization helps to

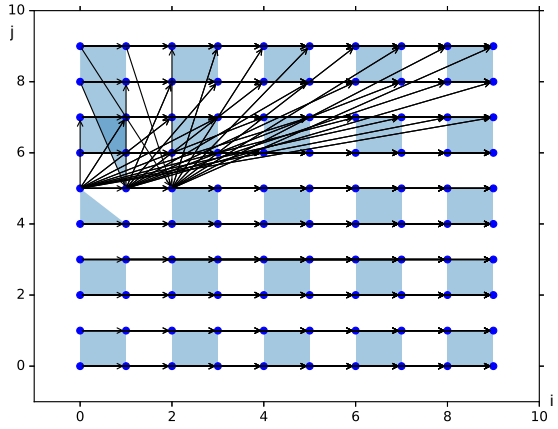

Fig. 2. Dependences, tiles of the size 2x2, and slices for Example 2.

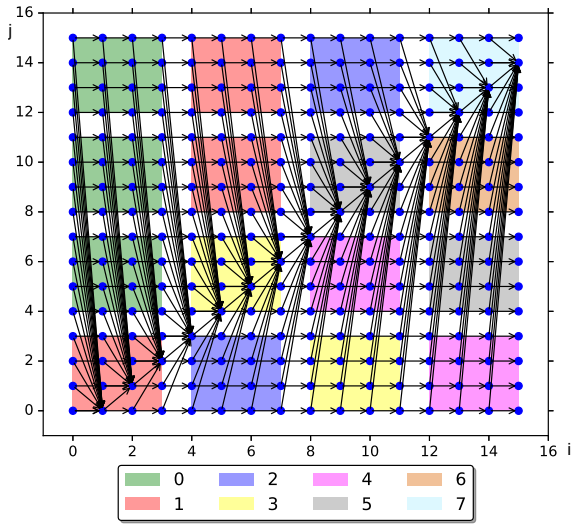

Fig. 3. Dependences, tiles of the size 4x4 and the schedule for Example 3.

discover parallelism available in Example 1 and what is the type of parallelism.

For Example 2 below

```
// Example 2.
for(i=0; i<=9; i++)
 for(j=0; j<=9; j++)
  a[j+4][j+1]=a[i+2*j+1][i+j+3];
```

Figure 2 depicts the iteration space with dependences. The loop tiling algorithm, implemented in TRACO, moves iteration [1,5] from tile [0,2] to tile [0,4] because it is the destination of the dependence whose source belongs to iteration [0,8]. For this example, TRACO is able to find three independent slices whose elements are tiles.

The iteration space with dependences for Example 3 is illustrated in Figure 3. Original rectangular tiling is preserved for this example. The figure shows tiles which are executed according to the free schedule. Time partitions are marked by different colors starting with the three independent green tiles.

```
// Example 3.
for(i=0; i<=15; i++)
 for(j=0; j<=15; j++)
    a[i][j] = a[i+1][i]+a[i+1][j];
```

Figure 4 illustrates dependences and tiles for the loop nest below in the 3D space.

```
// Example 4
for(k=0; k<=15; k++)
 for(i=0; i<=15; i++)
  for(j=0; j<=15; j++)
   a[i][j][k] =  a[i+1][j-1][k];
```

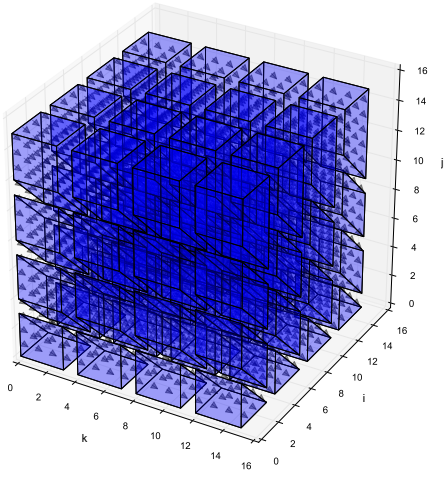

Fig. 4. Dependences and tiles of the size 2x2 for Example 4.

We use also a 3D projection to visualize 2D tiles for the arbitrary nested loops of Example 5, see Figure 5. The third axis indicates numbers of statements in the loop nest.

```
// Example 5
for(i=0; i<=15; i++){
 for(j=0; j<=15; j++)
  a[i][j] = a[i+1][j-1];
 for(j=0; j<=i; j++)
  b[i][j] = b[i][j+1]+a[i][0]; }
```

Figure 6 presents four independent slices whose elements are tiles for Example 6, the Planckian distribution. A 3D-projection reveals synchronization-free parallelism after appropriate rotating.

```
// Example 6 - Planckian distribution, loop=n=7
for ( l=1 ; l<=loop ; l++ )
  for ( k=0 ; k<n ; k++ ){
    y[k] = u[k] / v[k];
    w[k] = x[k] / ( exp( y[k] ) -1.0 );
  }
```

Summing up, we can conclude that visualization allows the programmer to find suitable loop transformations and discover available parallelism or code optimization.

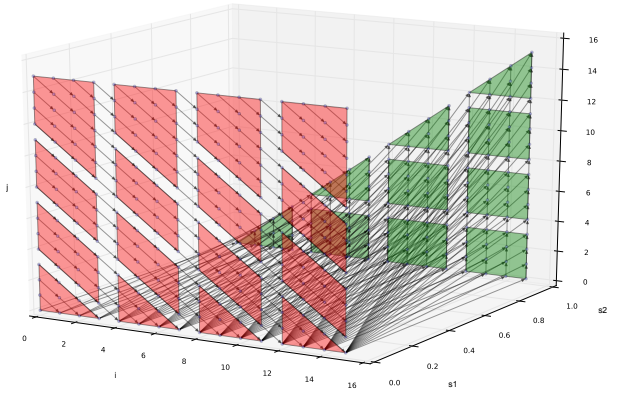

Fig. 5. Dependences and tiles of the size 4x4 for Example 5.

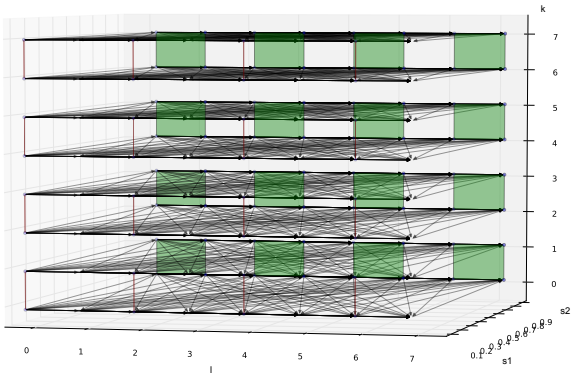

Fig. 6. Dependences and tiles of the size 2x2 for Example 6.

V. Related Work

Popular polyhedral libraries and compilers provide interface to visualization. The PolyLib tool [10] projects loop iteration domains by means of the VisualPolylib. LooPo [11] visualizes loop dependences before and after automatic parallelization. The 3D iteration space visualizer [12], [1] allows programmers to visualize and manipulate 3D dependence graphs, and to select a desired iteration space to start automatic parallelization search based mainly on unimodular transformations.

Clint [13] translates manipulations back to the polyhedral representation and ultimately transforms code to match visualization. This tool seems to be the most advanced visualizer and corresponds to extended versions of classical loop transformations (reordering, shifting, interchange, fusion, splitting, index set splitting, grain, reversal, skewing, tiling etc.).

The islplot[1] and Linpy tools [14] are based on islpy[2], a Python wrapper around Sven Verdoolaeges isl [3], the library for manipulating sets and relations of integer points bounded by linear constraints. Both the libraries are based on the matplotlib framework. Given the fact that TRACO is implemented also by means of the islpy interface, we considered these libraries for the presented approach. Islplot allows us to draw 2D maps and sets directly from islpy classes. However, domains of unions in isplot are drawn as separated figures. Therefore, we draw 2D polygons of tile points building convex hulls by means of Jarvis' March. Moreover, islplot is not integrated with matplotlib in 3D and provides only one function to draw sets in WebGL. Hence, for solids we used the interface provided by LinPy, which allows us to build the Polyhedron object from its constraints.

[1]http://tobig.github.io/islplot/

[2]https://documen.tician.de/islpy/

VI. Conclusion

Visualization of results of a dependence analysis and TRACO transformations carried out assists the development of parallel numerical programs. The approach complements analytical methods used in traditional automatic parallelizing compilers. Furthermore, the tool is helpful to investigate the use of interactive visualization for learning parallelization and the polyhedral model. In future, we plan to provide more interactive functions to the visualizer using event handling and object picking provided by matplotlib. We are going also to use the approach in order to design algorithms of arbitrary shaped tiles.

References

[1] Y. Yu and E. H. D'Hollander, "Loop parallelization using the 3d iteration space visualizer," *J. Vis. Lang. Comput.*, vol. 12, no. 2, pp. 163–181, Apr. 2001.

[2] M. Palkowski, T. Klimek, and W. Bielecki, "Traco: An automatic loop nest parallelizer for numerical applications," in *Computer Science and Information Systems (FedCSIS)*, Sept 2015, pp. 681–686.

[3] S. Verdoolaege, "Integer set library - manual," Tech. Rep., 2011. [Online]. Available: www.kotnet.org/~skimo//isl/manual.pdf,

[4] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[5] W. Pugh and D. Wonnacott, "An exact method for analysis of value-based array data dependences," in *Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*. Springer-Verlag, 1993.

[6] Wayne Kelly et al., "The omega library interface guide," College Park, MD, USA, Tech. Rep., 1995.

[7] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *PACT'13 IEEE Intern. Conf. on Parallel Architecture and Compilation Techniques*, Juan-les-Pins, 2004, pp. 7–16.

[8] W. Bielecki, M. Palkowski, and T. Klimek, "Free scheduling for statement instances of parameterized arbitrarily nested affine loops," *Parallel Computing*, vol. 38, no. 9, pp. 518–532, Sep. 2012.

[9] A. Lim, G. I. Cheong, and M. S. Lam, "An affine partitioning algorithm to maximize parallelism and minimize communication," in *In Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing*. ACM Press, 1999, pp. 228–237.

[10] V. Loechner, "PolyLib: A library for manipulating parameterized polyhedra," 1999. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.9.8197

[11] M. Griebl, "Automatic parallelization of loop programs for distributed memory architectures," 2004.

[12] Y. Yu, "A 3d-java tool to visualize loop-carried dependences," in *Applications, Proceedings of the International Conference ParCo'99*. College Press, 1999, pp. 17–20.

[13] O. Zinenko, C. Bastoul, and S. Huot, "Manipulating visualization, not codes," in *International Workshop on Polyhedral Compilation Techniques 2015 (IMPACT)*, 2015, p. 8.

[14] Mines Paristech, "Linpy documentation release 1.0," 2014.