

Clint: A Direct Manipulation Tool for Parallelizing Compute-Intensive Program Parts

Oleksandr Zinenko, Stéphane Huot, Cédric Bastoul

▶ **To cite this version:**

Oleksandr Zinenko, Stéphane Huot, Cédric Bastoul. Clint: A Direct Manipulation Tool for Parallelizing Compute-Intensive Program Parts. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Jul 2014, Melbourne, Australia. pp.109-112. hal-01055788

HAL Id: hal-01055788

https://inria.hal.science/hal-01055788

Submitted on 13 Aug 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Clint: A Direct Manipulation Tool for Parallelizing Compute-Intensive Program Parts

Oleksandr Zinenko
Inria, Université Paris-Sud & CNRS (LRI)
Orsay, France
Email: oleksandr.zinenko@inria.fr

Stéphane Huot
Université Paris-Sud & CNRS (LRI), Inria
Orsay, France
Email: stephane.huot@lri.fr

Cédric Bastoul
University of Strasbourg, Inria
Strasbourg, France
Email: cedric.bastoul@unistra.fr

***Abstract*—Parallel systems are now omnipresent and their effective use requires significant effort and expertise from software developers. Multitude of languages and libraries offer convenient ways to express parallelism, but fall short at helping programmers to find parallelism in existing programs. To address this issue, we introduce *Clint*, a direct manipulation tool aimed to ease both the extraction and the expression of parallelism. Clint builds on polyhedral representation of programs to convey dynamic behavior, to perform automatic data dependence analysis and to ensure code correctness. It can be used to rework and improve automatically generated optimizations and to make manual program transformation faster, safer and more efficient.**

Keywords: parallel programming, polyhedral model, software visualization, direct manipulation interface.

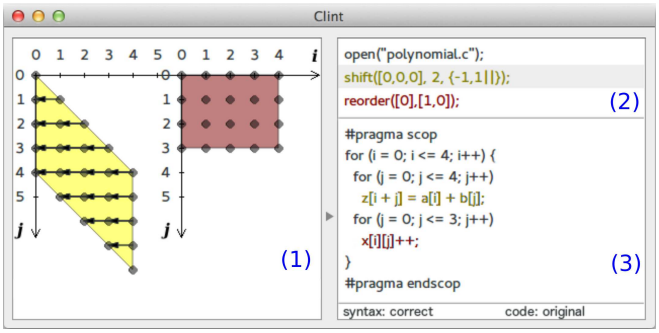


Fig. 1. *Clint* interface with synchronized views: (1) interactive visualization of loop iteration spaces; (2) editable transformation script history (3) code editor switching between transformed and original code to maintain context.

I. Introduction

The large scale adoption of parallel architectures in modern electronic devices urges the software industry to provide developers with tools for building efficient parallel applications. Despite tremendous effort, parallel program development remains challenging, especially when porting legacy sequential applications or targeting new parallel architectures. Several languages and libraries, such as Cilk, X10, Chapel or UPC, offer high-level parallelism expression constructs, but parallelism extraction is left under the responsibility of the developer.

Most existing approaches to automate parallelism extraction focus on compiler techniques for loop vectorization and thread-level parallelization and rely on an algebraic representation of programs called the *polyhedral model* [1]–[3]. It captures the dynamic behavior of a class of compute-intensive loops and allows to perform accurate data dependence analysis to ensure semantics preservation, and to restructure the input program to expose parallelism. However, polyhedral compilers are black-boxes that do not help users if the proposed restructuring does not solve their problem, e.g. has worse dynamic characteristics or does not expose parallelism due to the lack of semantics preservation hints. Semi-automatic tools based on directive scripts were proposed [4], [5], but this method has seen little interest as writing such scripts remains a tedious task, requiring expertise in a complex underlying theory.

Although the polyhedral model has a direct graphical representation featuring its geometric nature, it has not been explored yet as a way to leverage the power of the model through interactive visualizations. We have thus designed *Clint*, a tool to support parallelization through interactive direct manipulation of loop nest visualization (Fig. 1). It presents loop iteration spaces as geometric shapes, dynamic statement executions as solid dots and data dependences as arrows, together with continuous feedback during manipulation. *Clint* maintains correspondence between the valid source code and the visualization using polyhedral compilation techniques. This interactive approach may ease the design and exploration of program transformations comparing to manual or even semi-automatical code restructuring thanks to immediate feedback, editable history and undoability, allowing programmers to focus on parallelism given that code semantic equivalence is guaranteed by the tool.

II. Polyhedral Representation of Programs

The polyhedral model is an algebraic representation of a subset of imperative programs that encodes dynamic executions of statements [1]. It was designed to represent static control parts (SCoPs), which are loops with affine control and memory accesses, i. e., such that conditions and array subscripts are linear forms of outer loop counters and constants. Even with these restrictions, the model captures a broad range of compute-intensive, execution time-consuming loops that can be optimized [6]. Moreover, extensions to the model make it applicable up to entire functions [7].

The model considers dynamic statement *instances*, which are modeled for each statement as integer points inside a polyhedron called the *iteration domain* of the statement, that captures loops and branches surrounding it. For the polynomial multiplication loop nest presented in Fig. 2a, it contains two-dimensional vectors where the first dimension corresponds to the outer loop on `i` and the second to the inner loop on `j`

(Fig. 2b). Each vector refers to a particular execution of the statement S. Several compilers have the ability to raise SCoPs to a polyhedral form such as *GNU GCC*[1] or *LLVM*[2].

```
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
S:    z[i+j] += x[i] * y[j];
```

$$\mathcal{D}_S(N) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \;\middle|\; \begin{array}{l} 0 \leq i < N \\ 0 \leq j < N \end{array} \right\}$$

(a) Polynomial multiply kernel (b) Iteration domain of statement S

Fig. 2. Iteration domain of a statement.

The statement instances can be executed following a specific order defined by associating each point in the iteration domain with a logical execution date by means of a *mapping relation*. In case of parallel systems, multiple points having equal dates can be executed simultaneously given that a location of execution (e. g. processor core) is also associated with each instance. Mapping relations are a convenient and expressive way to define a complex composition of program transformations including, e.g., loop interchange, fusion, fission, skewing, tiling, index-set-splitting [5]. To ensure semantic equivalence, any mapping relation must not *violate* data dependences, i.e. preserve the relative order of instances accessing the same memory location.

A powerful property of the polyhedral approach is to make it possible to compute exact and instance-wise data dependences, and to check the mapping correctness against them [8]. It allows polyhedral compilers to construct correct mapping relations for various objectives automatically, e.g. to efficiently use cache memory or to expose parallelism. For the polynomial multiplication example, a mapping defined in Fig. 3a groups accesses to the same memory address z[i+j] in a single iteration of the outer loop, removing the data dependence between iterations, and thus making this loop parallelizable. The mapping reshapes the iteration domain to the form of Fig. 3b.

$$\theta_S(N) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} t1 \\ t2 \end{pmatrix} \;\middle|\; \begin{array}{l} t1 = i + j \\ t2 = i \end{array} \right\}$$

(a) Example time mapping for the statement S of Figure 2

$$\mathcal{T}_S(N) = \left\{ \begin{pmatrix} t1 \\ t2 \end{pmatrix} \;\middle|\; \begin{array}{l} 0 \leq t1 - t2 < N \\ 0 \leq t2 < N \end{array} \right\}$$

(b) Transformed iteration domain $\mathcal{D}_S(N)$ by time mapping $\theta_S(N)$

Fig. 3. Time mapping example and resulting transformed iteration domain.

Once the transformed domains have been computed in the model, the resulting code is generated by building a program that enumerates the integer points within the domain with respect to lexicographic ordering of their coordinates. Several efficient algorithms and tools exist for that purpose including *CLooG* [3] and *CodeGen+* [9].

```
  #pragma omp parallel for private(t2)
  for (t1 = 0; t1 <= 2*N-2; t1++)
    for (t2 = max(0, t1-N+1); t2 <= min(t1, N-1); t2++)
S:    z[t1] += x[t2] * y[t1-t2];
```

Fig. 4. Generated code from the transformed domain $\mathcal{T}_S(N)$.

[1]http://gcc.gnu.org/wiki/Graphite

[2]http://polly.llvm.org

Fig. 4 shows the final code generated from the transformed domain in Fig. 3b with a *parallel loop* on t1. The parallelizing optimization corresponds here to a classic loop transformation called *skewing* [6]. Although several automatic mapping algorithms were proposed, they are not necessarily resulting in the best parallelizing solution because of the multiple heuristics they use to handle complex underlying processor architectures [2], [3]. Furthermore, since they heavily transform the original program structure, the result is often unreadable for a programmer and thus uncorrectable. Semi-automatic approaches allow programmers to construct and modify a mapping relation freely, but they do not help in finding an appropriate transformation and still require significant expertise to describe it in a special syntax. Our objective with *Clint* is thus to provide programmers with an interactive tool to ease both parallelism extraction through direct manipulation of visual dependence abstraction and parallelism expression through automatic code generation.

III. The Clint Interactive Code Visualization

Design rationale – *Clint* is a direct manipulation interface designed to (i) help programmers with parallelizing compute-intensive programs parts; (ii) ease the exploration of possible transformations; and (iii) guarantee the correctness of the final code. *Clint* leverages the geometric nature of the polyhedral model by presenting code statements, their instances, and dependences in a scatterplot-like visualization of iteration domains similar to those used in the literature on polyhedral compilation. By making the visualization interactive, it reduces parallelism extraction and code transformation tasks to visual pattern recognition and geometrical manipulations, giving a way to manage the complexity of the underlying model.

During our preliminary design studies, interviewed domain experts expressed the following reasons for showing little interest in semi-automatic approaches: (i) the lack of immediate feedback on the effect of each transformation, (ii) the tight coupling between the code structure and the transformation specification language that complicates composition since each subsequent transformation potentially changes the code structure. *Clint* addresses these concerns by maintaining the consistency of the visualization independently of the underlying program structure, and by postponing the final code generation, making it possible to assess the results after each step of the transformation. For example, in Fig. 6, both blue and green statements are duplicated after transformation. Therefore subsequent transformations of the "original blue" statement require two operations with a semi-automatic tool while it can still be manipulated as one with *Clint*.

Finally, *Clint* introduces a navigable history view of transformations to favor try-and-fail strategy in the search for the best optimization. It can also save and reuse transformation scripts in order to share best practices in parallelizing optimizations. Overall, we believe that this approach may enable finer reasoning in terms of statements and instances rather than in terms of loops.

Visualizing Iteration Domains of Program Statements – *Clint*'s visualization of a program statement is a two-dimensional projection of its iteration domain as a polyhedron. The statement S of the polynomial multiply example of Fig. 2a

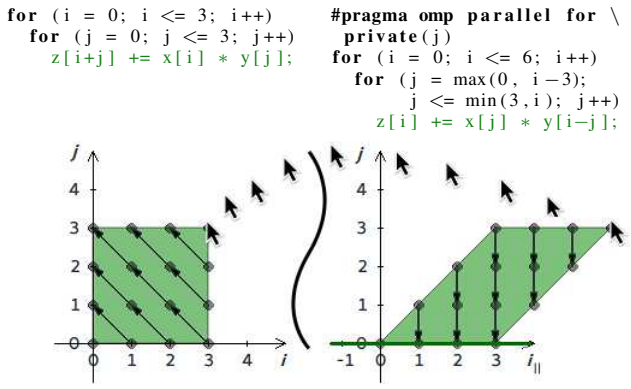


Fig. 5. Performing a *skew* transformation to parallelize polynomial multiplication loop by deforming the polygon. The code is transformed from its original form (left) to the skewed one (right) automatically with OpenMP pragmas generated for parallel loops depicted as thick green axes.

is represented in Fig. 5-left: axes represent *iterators*; the green shape delimits the *iteration domain* of the statement; the points inside the shape represent the *statement instances*, linked with arrows to denote data *dependences* between them. To convey information about the original or transformed execution order of statement instances, *Clint* visualizes a *mapped iteration domain*: a set of points obtained by applying the mapping relation to the integer points of the iteration domain like in Fig. 5-right for the polynomial multiplication example. Colors and thickness adjustments provide visual feedback on the mapping result: thick red arrows depict violated dependences while thick green axes represent parallel loops.

Multiple statements in the same loop are represented as different colored polygons of the same shape slightly displaced according to the execution order (Fig. 6). They share an axis that corresponds to their common loop. Multi-dimensional iteration domains are split up into two-dimensional projections and displayed as a scatterplot matrix [10]. Multiple statement instances, which differ only in the dimensions that were projected out, are depicted as a single point with a shade of gray representing the relative number of points. Two-dimensional visualization allows direct manipulation with a standard 2D input device (e.g. mouse) [11] and guarantees the consistency of the interface if the number of dimensions outgrows three.

Direct Manipulation of Statements and Instances – This geometrical representation of program statements affords direct manipulation of either the entire iteration domain (polygons) or particular statement instances (points), making parallelization just a matter of aligning dependence lines. *Clint*'s direct manipulation builds on the geometry-related vocabulary used in the polyhedral compilation community: points or polygons can be dragged and thus *shifted* (Fig. 6); polygons can be *fused* or *split* apart (Fig. 7), or *skewed* in space (Fig. 5) to reshape the respective iteration domains. Each graphical action is mapped to a sequence of program transformations that, if applied to the code, would change the program structure so that its visualization corresponds to the displayed one.

Clint Interface – As shown in Fig. 1, *Clint* combines three components: (i) the interactive visual representation of the program statements; (ii) an editable and navigable transformation history view; and (iii) the source code editor, in which the

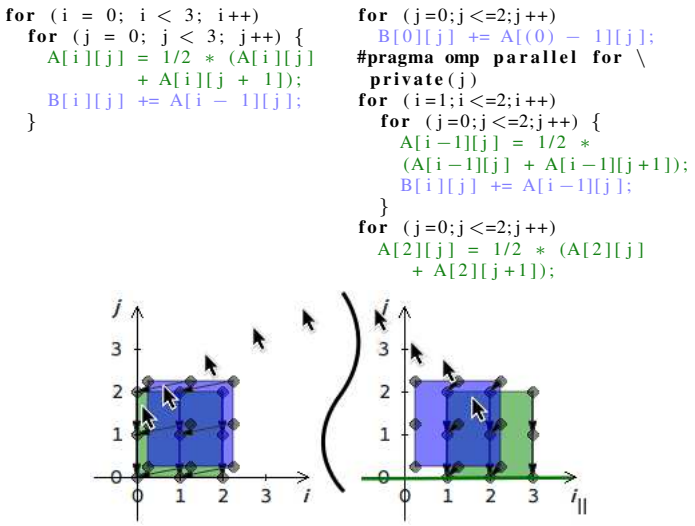


Fig. 6. Interactive *shift* transformation: the lighter polygon is dragged right to make dependence lines vertical so that they do not span between different iterations on `i`. Thanks to consistent visualization, statements can be manipulated as if they were not split between two loops.

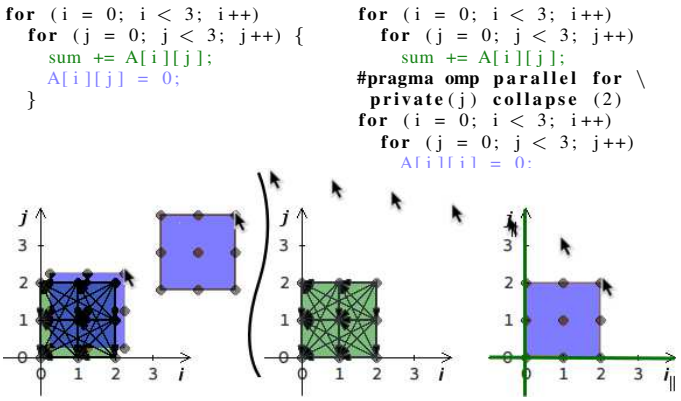


Fig. 7. Interactive complementary transformations to *split* statements between loops (left to right) or *fuse* them in a single loop (right to left). Each loop nest is represented as a separate coordinate system.

program structure can be changed manually. The three views are synchronized and updated automatically to reflect actual changes. The source code view can however be "locked" to the original version as the transformed code tends to quickly become complex and unreadable. The history is represented as a transformation script (using *Clay* [3] language) with coloring and grouping to reflect visual manipulations that resulted in multiple transformations.

Architecture and Implementation – *Clint*'s implementation relies on a collection of research tools and libraries that are the building blocks for polyhedral compilers and that usually work together as a single black-box [3]. *Clint* is a standalone application that consolidates editing facilities, feedback from the tools and output into a single consistent interface. It takes the original source code and uses *Clan* to raise it to the polyhedral model, it then maps direct manipulations to *Clay* transformation scripts and represents dependence violation feedback from *Candl*. Finally, it relies on *CLooG* to generate the C code with OpenMP pragmas for parallelism (Fig. 8).

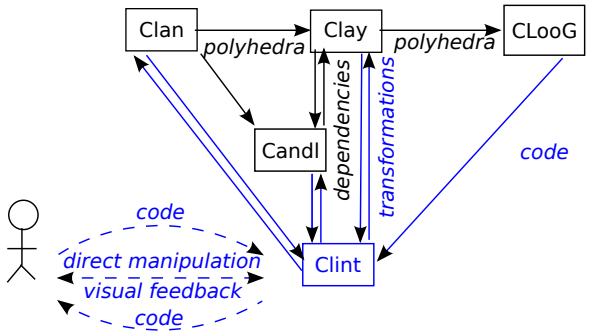


Fig. 8. *Clint* Software Architecture and Interaction Loop. Previously the user interacted with all tools (in black); *Clint* provides a single interface (in blue).

IV. Related Work

Visual Representations for Polyhedral Model – Spatial projections of the polyhedra are widely used in the domain literature. Several polyhedral libraries and automatic tools include visualization functionality for iteration domains and dependences, such as VisualPolyLib [12] or LooPo [13]. 3D iteration space visualizer [14] relies on an automatic search for parallelizing transformation. Tulipse [15] integrates dependence visualization into the Eclipse IDE. Our work extends these static visualizations by providing a mapping between direct manipulation of the visualization and code transformation.

Semi-Automatic Program Transformations – Several frameworks expose a high-level interface on top of polyhedral libraries through scripting languages, for example *UTF* [4] or *URUK* [5], more recent ones focusing on transformation sequencing like *Clay* [3]. *Clint* relies on it to convert visual actions to mapping relations and to provide programmers with a graphical interface that facilitates analysis tasks.

V. Conclusion

This paper introduced *Clint*, a direct manipulation visualization tool for parallelism extraction and expression. It builds on polyhedral compilation techniques to feature automatic data dependence analysis with real-time feedback, and to ensure correctness of the generated code. *Clint* leverages the geometric nature of the polyhedral model to complement conventional code editing with interactive visualization.

The underlying model limits the applicability of *Clint* to programs with static control flow, but still covers a broad range of compute-intensive loops, in which most of execution time is spent. Extending the visualization to capture entire programs would however lead to visual cluttering issues, especially in the case of numerous data dependences. Zoomable interfaces and on-demand animated transitions [16] between any combination of code and visualization may help to reduce cluttering as well as to provide a smoother link between the original and transformed code. Another future work direction is to investigate the use of interactive visualizations for learning parallelization by featuring dynamic feedback and interactive guidance through manipulation restrictions/enhancements (e.g. pseudo-haptic feedback [17] or semantic pointing [18]).

Finally, despite being widely used in the literature, the visualization of programs used in *Clint* was never evaluated. We conducted a preliminary study whose results suggest that users are able to reliably reconstruct code structure from the visualization and vice versa, which in turn suggests that the visualization is a complete representation of a program. We also asked some potential users of *Clint* to parallelize codes of different complexities, which they did faster and with greater accuracy than with usual methods (code editing and semi-automatic tools). While these results should be considered as preliminary, our interactive visual approach is a promising solution to ease the construction and adjustment of complex parallel optimizations of imperative programs.

References

[1] P. Feautrier and C. Lengauer, "Polyhedron model," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer US, 2011, pp. 1581–1592.

[2] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proc. of PLDI '08*, vol. 43, no. 6. ACM, 2008, pp. 101–113.

[3] C. Bastoul, "*Contributions to High-Level Program Optimization*. Habilitation Thesis. Paris-Sud University, France," 2012.

[4] W. Kelly and W. Pugh, "A framework for unifying reordering transformations," University of Maryland Institute for Advanced Computer Studies, Tech. Rep. UMIACS-TR-92-126.1, 1993.

[5] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," *International Journal of Parallel Programming*, vol. 34, no. 3, pp. 261–317, 2006.

[6] M. J. Wolfe, *High Performance Compilers for Parallel Computing*, C. Shanklin and L. Ortega, Eds. Addison-Wesley Longman Publishing Co., Inc., 1995.

[7] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *Compiler Construction*. Springer, 2010, pp. 283–303.

[8] P. Feautrier, "Dataflow analysis of array and scalar references," *Int. Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[9] C. Chen, "Polyhedra scanning revisited," in *Proc. of PLDI '12*. ACM, 2012, pp. 499–508.

[10] J. Heer, M. Bostock, and V. Ogievetsky, "A tour through the visualization zoo." *Commun. ACM*, vol. 53, no. 6, pp. 59–67, 2010.

[11] M. Beaudouin-Lafon, "Designing interaction, not interfaces," in *Proc. of AVI '04*. ACM, 2004, pp. 15–22.

[12] V. Loechner, "Polylib: A library for manipulating parameterized polyhedra," 1999. [Online]. Available: http://icps.u-strasbg.fr/polylib/

[13] M. Griebl, *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. University of Passau, 2004, habilitation thesis. [Online]. Available: http://www.uni-passau.de/ griebl/habilitation.html

[14] Y. Yu and E. D'Hollander, "Loop parallelization using the 3d iteration space visualizer," *Journal of Visual Languages & Computing*, vol. 12, no. 2, pp. 163–181, 2001.

[15] Y. W. Wong, T. Dubrownik, W. T. Tang, W. J. Tan, R. Duan, R. S. M. Goh, S.-h. Kuo, S. J. Turner, and W.-F. Wong, "Tulipse: a visualization framework for user-guided parallelization," in *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 4–15.

[16] P. Dragicevic, S. Huot, and F. Chevalier, "Gliimpse: Animating from markup code to rendered documents and vice versa," in *Proc. of UIST 11*. ACM, 2011, pp. 257–262.

[17] A. Lécuyer, J.-M. Burkhardt, and L. Etienne, "Feeling bumps and holes without a haptic interface: The perception of pseudo-haptic textures," in *CHI '04*. ACM, 2004, pp. 239–246.

[18] R. Blanch, Y. Guiard, and M. Beaudouin-Lafon, "Semantic pointing: Improving target acquisition with control-display ratio adaptation," in *CHI '04*. ACM, 2004, pp. 519–526.