Tulipse: A Visualization Framework for User-Guided Parallelization

Yi Wen Wong[1], Tomasz Dubrownik[2], Wai Teng Tang[3], Wen Jun Tan[3], Rubing Duan[4], Rick Siow Mong Goh[4], Shyh-hao Kuo[4], Stephen John Turner[3], and Weng-Fai Wong[1]

[1] National University of Singapore, Singapore
[2] University of Warsaw, Poland
[3] Nanyang Technological University, Singapore
[4] Institute of High Performance Computing, A*Star, Singapore

Abstract. Parallelization of existing code for modern multicore processors is tedious as the person performing these tasks must understand the algorithms, data structures and data dependencies in order to do a good job. Current options available to the programmer include either automatic parallelization or a complete rewrite in a parallel programming language. However, there are limitations with these options. In this paper, we propose a framework that enables the programmer to visualize information critical for semi-automated parallelization. The framework, called Tulipse, offers a program structure view that is augmented with key performance information, and a loop-nest dependency view that can be used to visualize data dependencies gathered from static or dynamic analyses. Our paper will demonstrate how these two new perspectives aid in the parallelization of code.

1 Introduction

As multicore and multi-node architectures become more prevalent and widely available, programs have to be written using multiple threads to take full advantage of all the cores available to them. Unfortunately, the task of multithreaded programming remains a hard one. Programmers are required to take more factors into account to write code that is both correct and that has good performance at the same time. This places a great burden on application developers, not all of whom may be as proficient in parallel programming as would be required.

Furthermore, it is increasingly difficult for existing legacy programs to take advantage of the multicore capabilities of these chips without resorting to a partial or complete rewrite of the source code. However, the cost of rewriting software is prohibitive. This problem is exacerbated by the fact that many application domain experts who maintain the legacy codes are not parallel programmers. Without the domain knowledge that is required in certain applications, it may also be difficult for programmers outside the domain to convert them from sequential programs into multithreaded ones. This is because apart from understanding the algorithm, the programmer performing the code modification has to understand the data structures, control flow and dependencies within the existing application to be able to do a good job in refactoring the code.

Visualization tools such as SUIF Explorer [8] and ParaGraph [1] have been developed to help facilitate the task of converting sequential code to parallel code. Such tools can reduce the effort needed by the programmer to understand the program and to make the necessary changes to parallelize it. For example, the ParaGraph tool displays a control flow graph of a code fragment to the user, and augments it with data dependency edges in order to help the user understand the data relations between different program statements. From this visual information, the programmer is then able to decide if the code is parallelizable or if synchronization is needed for certain data structures. Many of these tools, such as the ParaScope Editor [6] or ParaGraph, are also front-ends for their corresponding parallelizing compilers. They typically provide a limited form of visualization, usually as text output organized using tables, or in some cases, a two-dimensional graph representation of the data of interest [1]. Furthermore, they do not support the typical workflow a programmer goes through during the code parallelization process. For example, the ParaScope Editor and ParaGraph attempt to parallelize all the loops found within a program regardless of their suitability.

In this paper, we will describe an integrated visualization framework for parallelization that we have developed called Tulipse. The guiding principle behind the design is to simplify the workflow for parallelizing a program through an integrated visualization environment, and to provide visually useful information for the parallelizing process. This can be accomplished by the following capabilities. First, it consists of a graphical view that allows the user to visualize the global structure of an application by displaying procedures and loops hierarchically. Profiler measurements which indicate code sections that take up a large amount of the execution time can be overlaid in the graphical view. Second, a three dimensional view is provided to help the programmer visualize the data dependencies within a code section. It also allows the user to navigate through the view interactively. Through these capabilities, the programmer would then make an informed decision to effect the necessary code changes that will enable the parallelization of the application.

The framework is designed to involve the programmer in the parallelization workflow. This is because the programmer's knowledge about the code will be useful during the parallelization process. In addition, he is ultimately responsible for maintaining the application, and therefore may want to have finer control over code changes. Tulipse provides the following features to support semi-automatic code parallelization: (1) it is a visualization framework written in Java for the Eclipse IDE. In doing so, not only are all the facilities of the Eclipse IDE available to the programmers, the framework can also be extended with new visualization plug-ins; (2) it integrates a profiling and measurement tool that can be used to instrument the application under study. It can, for example, be used to find out which loops dominate execution time or have many cache misses.

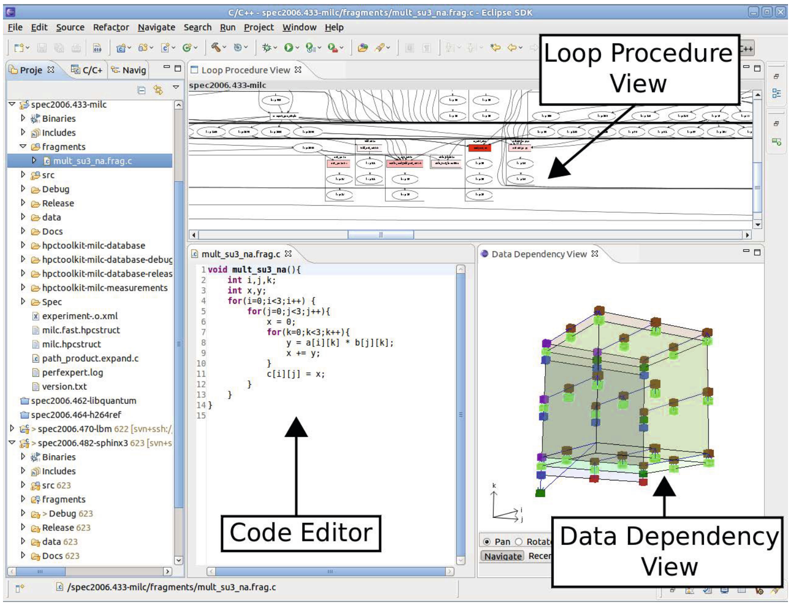
The programmer can then focus his attention on these hotspots, where even small improvements can have a significant impact on the overall execution time; (3) it provides a way to visualize data dependencies in the application through the use of a three-dimensional visualizer. The programmer can also animate, and walk-through the index space to obtain a better understanding of the data dependences. This allows the programmer to experiment with different ways of parallelizing the code; (4) it inserts OpenMP directives [9] into code which is found to be parallelizable using static and dynamic analyses, allowing the user to focus his attention on code sections which cannot be automatically parallelized.

The rest of the paper is organized as follows: Sect. 2 gives an overview of Tulipse and its capabilities. Section 3 presents examples on the usage of the visualization capabilities of Tulipse. Section 4 discusses prior work that is related and Sect. 5 concludes the paper.

2 Overview of Tulipse

The guiding principle behind the design of Tulipse is to offer to the programmer as much help as possible in the parallelization process. It does not attempt to anticipate the programmer's intent, but rather, allows the programmer to make the important decisions with respect to the code changes. It does this by providing sufficient visual feedback for the programmer to identify the best way to proceed with the parallelization. In this way, it gives the programmer control over the way the code is modified. In order to achieve these goals, two visualization capabilities are supported in Tulipse: the *Loop-Procedure View* and the *Data Dependency View*. Figure 1 shows these two views embedded in an instance of the Eclipse editor. The top panel displays the Loop-Procedure View, while the bottom-left panel shows a code editor and the bottom-right panel shows the Data Dependency View.

The workflow supported by Tulipse is shown in Fig. 2. An application is imported and loaded into the Eclipse IDE. Through the Loop-Procedure View, the user gets a hierarchical view of the whole application represented by procedures and loops that are defined in all of the project's source files. Next, the project is compiled and built with instrumentation automatically inserted into the application binary so that run-time statistics can be collected using hardware performance counters. The run-time statistics gathered from the profiling run is then overlaid onto this view. The user can choose to see, for example, which procedures or loops took up a large proportion of the overall execution time, or experienced a significant number of cache misses. The user can then focus his attention on these parts of the code using the Data Dependency View. This is a three-dimensional perspective of the data dependences within a code section. Through interactive visualization, it allows the user to decide whether the code can be effectively parallelized or tuned. The process can be repeated until the user is satisfied with the changes.


C/C++ - spec2006.433-milc/fragments/mult_su3_na.frag.c - Eclipse SDK
File Edit Source Refactor Navigate Search Run Project Window Help
Loop Procedure View
Code Editor
Data Dependency View
spec2006.433-milc
Binaries
Includes
fragments
mult_su3_na.frag.c
src
Debug
Release
data
Docs
Pan
Navigate
/spec2006.433-milc/fragments/mult_su3_na.frag.c

```
void mult_su3_na(){
    int i,j,k;
    int x,y;
    for(i=0;i<3;i++) {
        for(j=0;j<3;j++){
            x = 0;
            for(k=0;k<3;k++){
                y = a[i][k] * b[j][k];
                x += y;
            }
            c[i][j] = x;
        }
    }
}
```

Fig. 1. A screenshot of the Tulipse plug-in in the Eclipse development platform

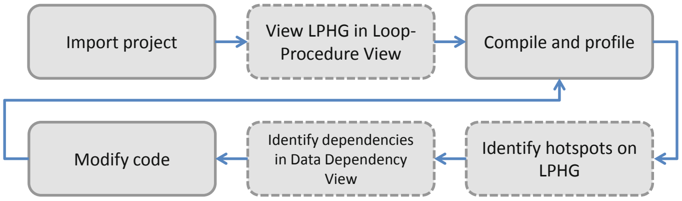

Import project
View LPHG in Loop-Procedure View
Compile and profile
Identify hotspots on LPHG
Identify dependencies in Data Dependency View
Modify code

Fig. 2. Workflow of parallelization process

2.1 Loop-Procedure View

This view presents the user with visual information on the application's overall structure. In this view, a Loop-Procedure Hierarchy Graph (LPHG) is constructed for the entire application project. This is essentially a call-graph embedded with loop nest relations obtained from the application source files. For example, Fig. 3 shows the LPHG for the SPEC2006 `470.lbm` benchmark application [12]. The inset in the figure shows a zoomed-in image of the hotspots. The square-shaped nodes denote procedure definitions, and the ellipse-shaped nodes denote loops. Edges into a procedure node represent calling instances to that procedure. An edge from a procedure to a loop indicates that the loop is defined within the procedure. An edge from a loop to another loop indicates that the latter is nested in the former. There may be multiple incoming edges for a procedure, indicating multiple calling instances of the same procedure. However, there will only be exactly one incoming edge for a loop since there can only be one definition of the loop residing either within a procedure or within another loop. Recursive procedures create cycles in the graph. Nodes which are grouped within a box belong to the same source file.

We integrated HPCToolkit [4] into our visualization framework to simplify run-time performance measurement for the user. HPCToolkit uses hardware counters provided by the underlying microprocessor to measure performance metrics for identifying performance bottlenecks during program execution. Although only HPCToolkit is currently supported by Tulipse, it is relatively easy to add support for other measurement tools such as Tau [13]. Profile measurements of the application taken by the performance measurement component can be overlaid on the LPHG in the Loop-Procedure View. Different measurement metrics can be selected for the overlay. Customized metrics can also be constructed using the base measurement metrics. The metric values are normalized and mapped to a default color gradient from red to white, where red indicates ‘hot’ while white indicates ‘cold’. The respective nodes on the LPHG, including both procedure definition and loop information, are colored according to this mapping.

Profile measurements are loaded into the view by accessing the view menu in the Loop-Procedure View. Different metrics can be overlaid on the LPHG through the `Load Overlay` menu, including base metrics from the profile measurements as well as user-specified derived metrics. A derived metric is essentially a formula constructed by applying operators on metrics and numerical constants. It is also possible to specify a custom color gradient to identify different ranges. For example, the user may want to highlight metric values ranging from 50% to 100% as hotspots, instead of just the top 10%. This can be adjusted using a different color gradient with a larger range for the hotspots. By inspecting the overlaid LPHG, the programmer will be able to identify problematic code regions quickly, and focus his attention on them. By accessing a context menu, we provide the user with the ability to target a code fragment of a problem node through the code editor, or to visualize it through the Data Dependency View.

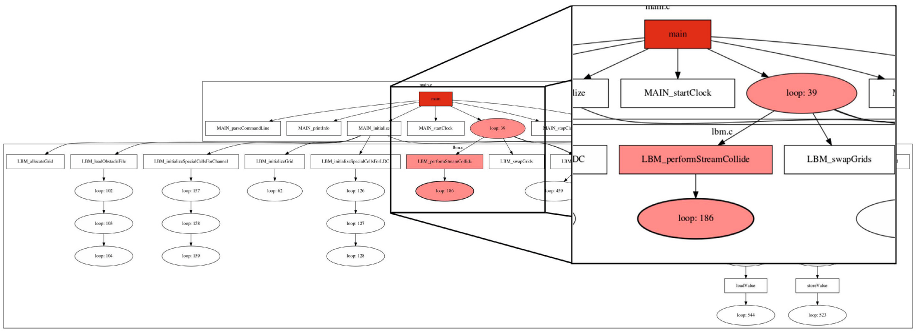
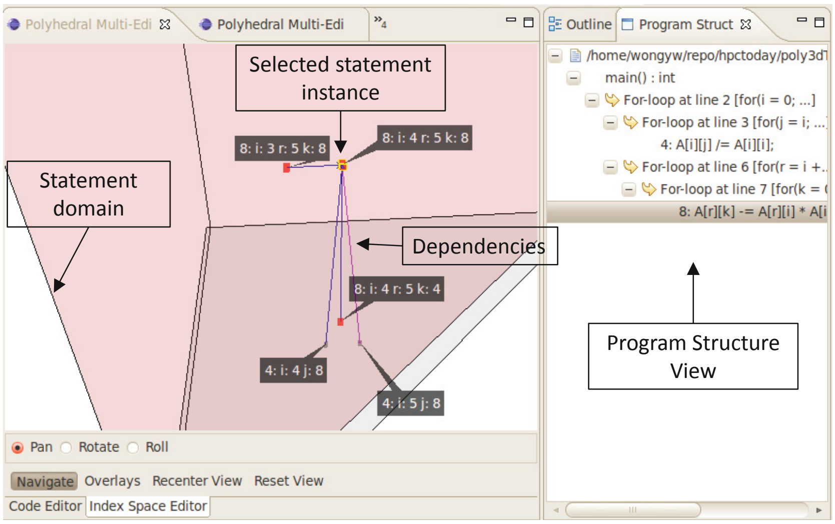


Fig. 3. Loop-Procedure View overlaid with performance measurements

2.2 Three-Dimensional Data Dependency View

The 3D Data Dependency View provides a way for programmers to interactively explore the code's data layout as well as the parallelization options for the loop nests. This view can be launched directly from the Loop-Procedure View by selecting a hotspot. In this view, the iteration spaces of loops within a procedure are visualized. Statements in the loops are mapped to a higher dimensional space based on the polyhedral model [3]. Each statement has a domain determined from its enclosing control statement. Its upper and lower bounds are extracted from the enclosing loops, and intersected with the domains of conditional statements to obtain a system of linear constraints that defines a polyhedron. The polyhedron in this space is projected into the 3D world space for visualization.

Loop nests do not necessarily have to be tightly nested and dependencies can cross loop boundaries. There is no limit on the level of nesting. By selecting a loop in the code panel, the visualizer will highlight the associated domains of the all enclosed statements within the loop. This reduces clutter in the visualization. The data dependencies are currently obtained by static analysis and dynamic analysis built into the framework. The analyses yield *flow dependence* (read after write), *output dependence* (write after write), and *anti-dependences* (write after read). Each dependency is represented by an arrow drawn between the projected coordinates of the source and target statement instance in the 3D world space. Static analysis of dependencies is conservative in order to guarantee correctness, thus dependencies may be over-reported. On the other hand, dependencies obtained by dynamic analysis are dependent on the execution instance and may



```
/home/wongyw/repo/hpctoday/poly3d
  main() : int
    For-loop at line 2 [for(i = 0; ...]
      For-loop at line 3 [for(j = i; ...
        4: A[i][j] /= A[i][i];
      For-loop at line 6 [for(r = i +..
        For-loop at line 7 [for(k =
          8: A[r][k] -= A[r][i] * A[i
```

Fig. 4. Selection of a statement instance (line 8, iteration $(i = 4, r = 5, k = 8)$) in the Data Dependency View. Blue arrows indicate the dependences of this instance. Pink arrows show which statements are dependent on the selected instance.

be input-dependent. However, it can obtain dependencies in cases where static analysis is difficult or impossible, such as code involving pointer arithmetic.

Initially, the convex hulls corresponding to each statement domain are shown to the user, with each statement using a different color. As the user zooms in, statement instances and data dependencies (represented by cubes and arrows) become visible to the user. This constrains the amount of information that is shown to the user at any one time. Furthermore, users can use the mouse to obtain more information about each point of interest, e.g. the dependencies of a particular statement instance in the iteration space, or which specific statement instances are responsible for a particular dependency (see Fig. 4), by clicking on a node or an arrow in the view. This view also provides additional features that allow users to further understand the code, and reason about the parallelization options. For example, users can step through the iteration space either manually or via animation to visualize the execution order of the statement instances. This allows users to see the dependencies as they are generated during execution, and to visually inspect if parallelization of certain loops are safe.

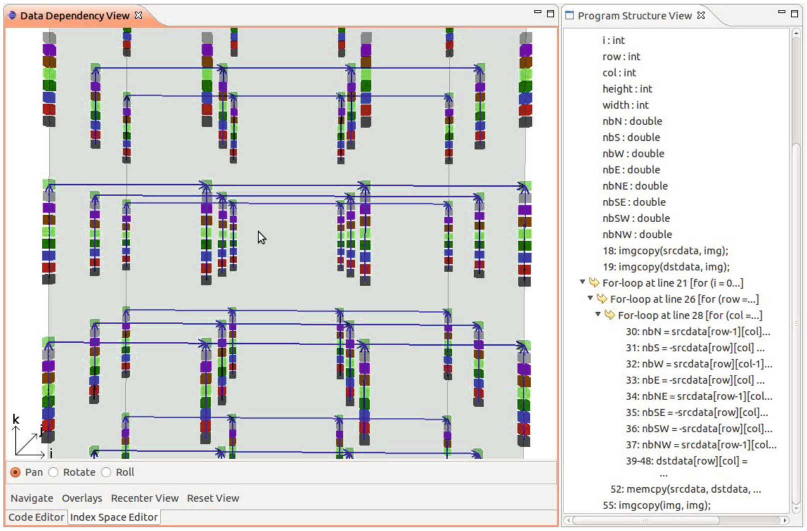
As a convenience to users, hints that depict parallel loops are provided by the visualization engine by drawing a set of hyperplanes on top of the view. In general, the equations for each hyperplane are specified as a set of constraints such that statement instances that fall within the same plane can be executed by the same thread. These planes are projected down to 3D space, and drawn as an overlay on top of the polyhedral visualization. The framework also includes an OpenMP parallelization component that assists the user in parallelizing code. This component helps to insert OpenMP directives into the original source code by presenting a list of parallelizable loops for the user to choose from. After selecting the loop to parallelize, it then allows the user to modify a list of shared and private variables that have been automatically detected. Reduction patterns can also be detected and highlighted to the user. The modified code can be previewed by the user before committing the changes.

3 Examples

In this section, we shall demonstrate the use of our visualization framework on two examples. The first example is an image processing example found in many applications. The second example is from *482.sphinx3*, a speech recognition application taken from the SPEC CPU2006 benchmark suite [12].

3.1 Anisotropic Diffusion

Anisotropic diffusion is an image noise reduction technique that is commonly used in applications such as in ultrasound imaging or magnetic resonance imaging [10]. It simulates an iterative diffusion process which is non-linear and space-variant, and is aimed at removing image noise while at the same time preserving important image details, especially the edges in an image. The algorithm takes a noisy image and calculates for each pixel a set of eight values based on predefined kernels, and then accumulates the sum of their weighted differences.



```
    i : int
    row : int
    col : int
    height : int
    width : int
    nbN : double
    nbS : double
    nbW : double
    nbE : double
    nbNE : double
    nbSE : double
    nbSW : double
    nbNW : double
    18: imgcopy(srcdata, img);
    19: imgcopy(dstdata, img);
  ▾ For-loop at line 21 [for (i = 0...]
    ▾ For-loop at line 26 [for (row =...]
      ▾ For-loop at line 28 [for (col =...]
          30: nbN = srcdata[row-1][col]...
          31: nbS = -srcdata[row][col] ...
          32: nbW = srcdata[row][col-1]...
          33: nbE = -srcdata[row][col] ...
          34: nbNE = srcdata[row-1][col...
          35: nbSE = -srcdata[row][col]...
          36: nbSW = -srcdata[row][col]...
          37: nbNW = srcdata[row-1][col...
          39-48: dstdata[row][col] =
                 ...
      52: memcpy(srcdata, dstdata, ...
    55: imgcopy(img, img);
```

Fig. 5. Visualization of the code section in the `diffuse` procedure. The user can select statements using the Program Structure View on the right to highlight the enclosing domains. No arrows cross the planes normal to j and k axes, indicating that the `j` and `k` loops are parallelizable.

From the Loop-Procedure View, we determined that the code section in the `diffuse` procedure took up about 98% of the total execution time while the rest is mainly due to I/O operations. Figure 5 shows the corresponding polyhedral model of the code. Since there are three `for`-loops present, the 3D iteration space resembles a rectangular cuboid. Each colored cube represents a statement instance within the iteration space, and each arrow represents a producer-consumer relationship between statement instances. The i, j and k axes denote the direction of their respective loop iterations. As the implementation contains pointer arithmetic, the dependencies were obtained using dynamic analysis. The user can highlight a statement instance in the Data Dependency View by selecting the corresponding statement in the code panel on the right. By inspecting the polyhedral model in Fig. 5, the user is able to determine that only planes normal to the j and k axes do not have any arrows crossing them. Therefore, the corresponding `j` and `k` loops do not have any loop-carried dependencies and are fully parallelizable. The programmer may then select the outer loop and invoke the OpenMP parallelization component to insert OpenMP directives to the loop.

On a Intel Core 2 Extreme Q6850 processor running at 3.00GHz and with a 512 by 512-pixel image, the OpenMP version of the code yielded close to a 4 times speedup compared to the original single-threaded version. This is a significant improvement because the sequential version of the code is not suitable for many practical applications that demand real-time processing of images acquired from sensors. The sequential form of the code could only manage about 4.2 frames per second (fps), whereas the OpenMP version obtained a respectable 16.6 fps. This improvement is significant as it will allow the algorithm to be used in many real-time applications.

3.2 Speech Recognition System (`482.sphinx3`)

Sphinx-3 is a speech recognition system based on the Viterbi search algorithm using beam search heuristics. The inputs are read initially, and the application then processes the speech to calculate the probabilities at each recognition step in order to prune the set of active hypothesis. We overlaid the runtime statistics on the Loop-Procedure View and identified two hotspots. Essentially, the application is dominated by two procedures that together account for over half of the total runtime. The two hotspot procedures identified are `mgau_eval`, which accounted for 30.1% of the total cycles, and `vector_gautbl_eval_logs3`, which accounted for 24.5% of the total cycles. These two procedures also accounted for 92.9% of the total floating point instructions issued. Analysis of the source code revealed that the two procedures are executing similar loops. As such, we shall only present one of the procedures, `mgau_eval`, in this paper. The procedure consists of a two-dimensional loop with the inner loop accumulating a score for the search probabilities.

The polyhedral model generated by the Data Dependency View is shown in the Fig. 6. The polyhedra are two-dimensional planes, which correspond to the two-dimensional loop. The statement represented by the gray nodes has a flow dependence to the statement represented by the blue nodes. Within the `j` iteration, the red and blues nodes have a flow dependence, as well as a loop-carried data dependence. By inspecting Fig. 6, the user can see that it is possible

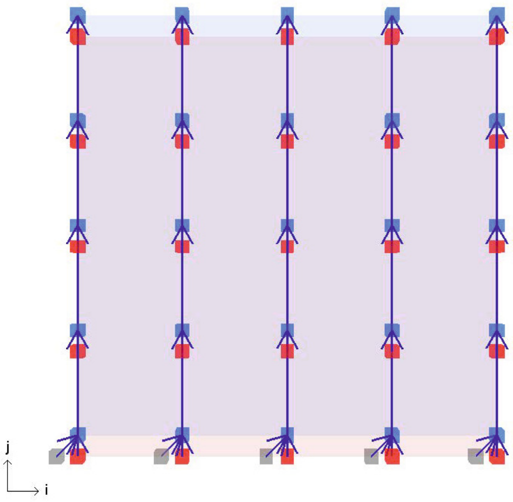


Fig. 6. Visualization of the `mgau_eval` procedure. Loop iterations along the i axis can be partitioned since there are no dependencies in the horizontal direction.

to partition the polyhedra in the i domain by cutting across the i axis. In other words, one can parallelize the `i` loop without violating the correctness of the code, since there are no arrows in the horizontal direction, and therefore, no dependencies in the i direction.

4 Related Work

A number of visualization tools that target the Fortran language have been developed over the past two decades. The ParaScope Editor [6], developed at Rice University, allows the user to step through each loop in the program and displays the relevant information in a two-panel window. The bottom panel displays a list of the detected data-dependences along with the dependence vectors, as well as other relevant details in a tabular format. The top panel displays the source code of the current loop under consideration. It then allows the user to select a transformation that is deemed safe to be applied to the program. NaraView introduced an interactive 3D visualization system for aiding the parallelization of sequential programs [11]. Of notable interest is the 3D Data Dependence View, which displays data accesses as colored cubes and dependences as poles connecting the cubes. However, NaraView does not perform program profiling to determine the important loops, and does not include the ability to animate a walk-through of the iteration space. Instead, it uses the z-axis to represent the iteration access time and the x-y plane to denote the data location. Therefore, visualization is limited to loops with a depth of at most two.

SUIF Explorer [8] is another interactive parallelization tool which targets both the Fortran and C languages. It includes a Loop Profile Analyzer that identifies the important loops that dominate the execution time. A useful feature of the tool is that it applies inter-procedural slicing to the program to display only the relevant lines of code to the programmer so that he can make the appropriate decisions. Annotations are added to the code, which are then checked for their validity by the built-in checkers. The annotations help to enable the parallelizing compiler to parallelize the loop. However, it does not make use of OpenMP directives. Other tools such as Merlin [7] provide a textual representation of the program analysis to the user. It compiles the code using the Polaris parallelizer, gathers static and dynamic execution data, then performs analysis using "performance maps", and finally presents diagnostics about the program as well as suggestions on how to improve the code to the user in a textual format. Currently, the only tool that supports 3D visualization for Fortran code is the Iteration Space Visualizer [15].

In comparison, there are fewer visualization tools that target the C language. The SUIF Explorer is able to perform inter-procedural analysis on C programs by building upon the SUIF parallelizing compiler, and presents the results to the user using program slicing. Another tool that supports C code parallelization is ParaGraph [1]. It makes use of the Cetus compiler [2] to automatically parallelize loops using OpenMP directives [9]. Alternatively, it also allows the user to specify the directives, which it validates before attempting parallelization.

To our knowledge, ParaGraph is currently the only visualization tool for C that runs as a Eclipse plug-in. However, apart from the source code outline and a properties tab, the visualization support provided to the user consists of only a control flow graph augmented with the dependency information in the form of directed and dashed arrows between control blocks. Another related work is VisualPolylib [14], which draws the polyhedral model supplied by the user. However, the user has to manually extract the model from the code and supply it to the tool. Apart from ParaGraph, most of the tools are stand-alone front-ends, and are not integrated with any IDEs. On the other hand, Tulipse is wholly integrated in the Eclipse software development environment, which makes it highly extensible and allows it to leverage many existing software engineering tools and plug-ins available in the Eclipse developer framework. In addition, in using our framework, parallelization opportunities can be identified even if the code cannot be analyzed using static approaches, such as code involving pointer arithmetic.

The Intel Parallel Advisor [5] is closest to what our framework offers in terms of capabilities and workflow. However there are a few major differences. First, Intel Parallel Advisor is mostly textual. Unlike our framework, it does not provide visualization capabilities. Secondly, the workflow is different. The Advisor requires the programmers to take a trial and error approach by guessing and annotating parts of the code which they believe can be parallelized. The Advisor then performs a trial run to detect data races. If races are detected, the user is notified and may again attempt to identify other parallelizable sections of the code. On the other hand, in our workflow, the data dependencies are first obtained and then projected to 3D space to allow the user to directly identify the parallelization opportunities.

5 Conclusions and Future Work

In this paper, we introduced Tulipse, a visualization framework for parallelization built on top of Eclipse, with the goal of enhancing program understanding and reducing the cognitive load of the developers in parallelizing applications.

In Tulipse, the programmer starts with a loop-procedural hierarchy graph of the entire application using the Loop-Procedure View. This gives the user a bird's eye view of the application. Color coding allows for the display of selectable performance data in this view, allowing the programmer to quickly zoom in to the hotspots or the problem areas. Zooming in to the loop level exposes the polyhedral view of the loop. In the Data Dependency View, the programmer can easily correlate data dependencies found in the code by static or dynamic analysis. This visualization also allows for the programmer to estimate the effort involved in attempting to parallelize the loop along a particular dimension of the iteration space. The user may invoke the OpenMP parallelization component to aid in adding OpenMP directives to the selected loop.

Performance is but one aspect of software development. By being integrated in a rich software development environment such as Eclipse, it gives developers access to a wide range of tools that support their various workflows. As future work, we shall be extending Tulipse with other parallelization and performance tuning visualizers that will grow its functionality, for example cache usage visualization that will help in identifying optimal data layout. We will also be investigating how the user can carry out code transformations such as loop skewing and tiling with the help of interactive visualization.

Acknowledgments. This work was supported by the Agency for Science, Technology and Research PSF Grant No. 102-101-0028. We are also grateful to the anonymous reviewers for their suggestions.

References

1. Bluemke, I., Fugas, J.: A tool supporting C code parallelization. Innovations Comput. Sci. Soft. Eng., 259–264 (2010)
2. Dave, C., Bae, H., Min, S.J., Lee, S., Eigenmann, R., Midkiff, S.: Cetus: A source-to-source compiler infrastructure for multicores. Computer 42, 36–42 (2009)
3. Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M., Temam, O.: Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. Int. J. Parallel Program. 34, 261–317 (2006)
4. HPCToolkit, `http://www.hpctoolkit.org`
5. Intel Parallel Advisor, `http://software.intel.com/en-us/`
6. Kennedy, K., McKinley, K.S., Tseng, C.W.: Interactive parallel programming using the ParaScope editor. IEEE Trans. Parallel Distrib. Syst. 2, 329–341 (1991)
7. Kim, S.W., Park, I., Eigenmann, R.: A Performance Advisor Tool for Shared-Memory Parallel Programming. In: Midkiff, S.P., Moreira, J.E., Gupta, M., Chatterjee, S., Ferrante, J., Prins, J.F., Pugh, B., Tseng, C.-W. (eds.) LCPC 2000. LNCS, vol. 2017, pp. 274–288. Springer, Heidelberg (2001)
8. Liao, S.W., Diwan, A., Bosch Jr., R.P., Ghuloum, A., Lam, M.S.: SUIF Explorer: an interactive and interprocedural parallelizer. In: PPoPP, pp. 37–48 (1999)
9. OpenMP, `http://www.openmp.org`
10. Perona, P., Malik, J.: Scale-space and edge detection using anisotropic diffusion. IEEE Trans. Pattern Anal. Mach. Intell. 12, 629–639 (1990)
11. Sasakura, M., Joe, K., Kunieda, Y., Araki, K.: Naraview: An interactive 3D visualization system for parallelization of programs. Int. J. Parallel Program. 27, 111–129 (1999)
12. SPEC CPU 2006 v1.1, `http://www.spec.org`
13. Tau: `http://www.cs.uoregon.edu/research/tau/`
14. VisualPolylib, `http://icps.u-strasbg.fr/polylib/`
15. Yu, Y., D'Hollander, E.H.: Loop parallelization using the 3D iteration space visualizer. J. Visual Lang. Comput. 12, 163–181 (2001)