A 3D-JAVA TOOL TO VISUALIZE LOOP-CARRIED DEPENDENCES

YIJUN YU

Parallel Information Systems, University of Ghent,
St-Pietersnieuwstraat 41, 9000 Ghent, Belgium
E-mail: Yijun.Yu@elis.rug.ac.be

The interactive tool presented allows programmers to visualize and manipulate the three-dimensional iteration space dependence graph (ISDG). Constructed from the runtime analysis, it reveals the potential parallelism and permits the programmer to find suitable loop transformations which maximize the speedup.
The tool manipulates it with a number of graphical operations such as rotations, zooms, cutting planes and projections. Once the runtime trace of the program is generated, the new iteration space of a unimodular or non-singular transformation can be constructed without having to rewrite and execute the transformed program. In addition the temporal behavior of the program is revealed by a step-by-step traversal animating the iterations presently executed as well as the past and the future iterations in either data-flow, loop-wise or plane-wise order. From the ISDG, a dependence distance matrix is derived for both uniform dependence and non-uniform dependence problems.
It has been used to speedup a real-life computational fluid dynamics(CFD) program which is hard to parallelize with traditional compiler.
Keywords iteration space, program visualizing, loop transformation

1 Introduction

In the last decade, many loop transformation techniques have been developed, such as the unimodular [?] and non-unimodular transformations [?] for perfectly nested loops; and transformations for non-perfectly nested loops [?]. These techniques are based on the data dependence analysis to find the parallel loops and optimize their execution. However, the array subscript expressions, the loop bounds, and the conditional branches are often too complicated for a compiler to detect precisely all the loop dependences [?]. Therefore the run-time dependence analysis technique has emerged [?]. Unlike the compile-time dependence analysis, the runtime dependence analysis gathers more accurate information. Although the run-time analysis has to sacrifice the independence from the program input, the accurate runtime information aid to study the difficult loops which are hard for the compiler.

In this paper an interactive tool is presented for three-dimensional(3-D) iteration space dependence graph(ISDG). Presently, the tool has the following functions to construct, visualize and manipulate the ISDG: constructing an

ISDG from run-time analysis of the program; visualizing the dependences of a multi-level nested loop; manipulating the ISDG by a number of graphical operations; detecting parallelism or loop transformation by data-flow, loop-wise or plane-wise traversal of the graph; extracting distance matrix from the loop dependences; constructing a new iteration space without rewriting and execution of the transformed program; testifying and evaluating the loop transformations for maximum speedup.

Section 2 discusses the basics of iteration space and iteration space dependence graph. Section 3 discusses the technique to derive an ISDG from the run-time analysis of the program. Section 4 explores the visualizing and manipulating functions; As a result, section 5 shows the application to the most time-consuming loop nest of a computational fluid dynamics (CFD) program which is hard to parallelize by a traditional compiler.

2 The iteration space dependence graph

The *iteration space dependence graph* is a directed acyclic graph $< \mathcal{N}, \mathcal{E} >$ with nodes $\mathcal{N}$ representing iterations and edges $\mathcal{E}$ representing the dependences among them.

For a m-level normalized nested loop with i_j as index variable, L_j, U_j as lower and upper bounds of loop j and all loop steps are 1, the node set is:

$$\mathcal{N} = \{\mathbf{i} = (i_1, \ldots, i_m) | \forall 1 \leq j \leq m : L_j \leq i_j \leq U_j\} \tag{1}$$

In sequential loops, the iteration $\mathbf{i}$ executes before $\mathbf{j}$ if $\mathbf{i}$ is *lexicographically less than* $\mathbf{j}$, denoted as $\mathbf{i} \prec \mathbf{j}$, i.e., $i_1 < j_1 \vee \exists k \geq 1 : i_k < j_k \wedge i_t = j_t \; for \; 1 \leq t < k$. The lexicographical order of two dependent iterations $\mathbf{i} \prec \mathbf{j}$ also defines a lexicographically positive *distance vector* $\mathbf{d} = \mathbf{j} - \mathbf{i}$.

If both iterations $\mathbf{i_1} \prec \mathbf{i_2}$ read or write to the same array element $A(f(\mathbf{i_1})) = A(g(\mathbf{i_2}))$ and at least one of the iteration write, no intermediate iteration $\mathbf{j} \mid \mathbf{i_1} \prec \mathbf{j} \prec \mathbf{i_2}$ overwrite the same array element by any reference $A(h(\mathbf{j}))$, there is a direct *loop carried dependence* between the iterations $\mathbf{i_1}$ and $\mathbf{i_2}$, denoted as $\mathbf{i_1} \; \delta \; \mathbf{i_2}$. Therefore, the edge set of is defined as:

$$\mathcal{E} = \{(\mathbf{i}_1, \mathbf{i}_2) | \mathbf{i}_1, \mathbf{i}_2 \in \mathcal{N} \wedge \mathbf{i}_1 \; \delta \; \mathbf{i}_2\} \tag{2}$$

3 Constructing an ISDG from run-time analysis

To construct an ISDG from the program, the tool extends the run-time method described in [?] with treatment of non-perfectly nested loops.

First, a statement is inserted for each array reference $A(f(\mathbf{i}))$ in the loop to output in the execution order the sequential counter for iteration $\mathbf{i}$, the array name A, the subscript expression $f(\mathbf{i})$ and the type of reference (Read/Write).

Then, for all references outside the innermost loop body of an m-level non-perfectly nested loop: $A(f(i_1, \dots, i_k))$ where $k < m$, it records the next iteration count so that these references are treated in the same manner with the references within the innermost loop body. The converted perfectly nested loop, whose ISDG is constructed in this way, subjects to the unimodular transformation and code generation as described in ?. For example, the ISDG of the non-perfectly nested loop in Figure 1(a) has nodes $(i_1, i_2, i_1 + 1)$ which summarize all the references in both instances of the statements $S_1(i_1, i_2)$ and $S_2(i_1, i_2, i_1 + 1)$.

The graph is constructed from the records: (1) the tool keeps a record in a stack, for each array element, the last iteration that writes the element; (2) it marks the loop-carried dependence edges from the records to construct the graph: (2.1) Before pushing each new write reference in an iteration $\mathbf{j}$ to the stack, it pops up each the references to the same array element in iteration $\mathbf{i}$ and marking an output- or an anti- dependences edge $\mathbf{i}\ \delta\ \mathbf{j}$ if the reference in iteration $\mathbf{i}$ is a write or read reference. Then the iteration $\mathbf{j}$ is remembered as the new last write iteration; (2.2) Before pushing each new read reference in an iteration $\mathbf{j}$ to the stack, if iteration $\mathbf{i}$ is the last iteration that writes to the same array element, it marks a flow-dependence edge from $\mathbf{i}$ to $\mathbf{j}$.

For the programmer, the pragma *C$doisv* before the innermost loop of the selected iteration space is the only required modification to the program.

4 Exploring the ISDG

There are two ways to expose loop parallelism. One way is looking into the graph to see if there are any parallel partitions of the iteration set $\mathcal{N}$. Rotating the graph to a certain angle may expose the partitions of independent iterations to eyes for simple programs like matrix multiplication.

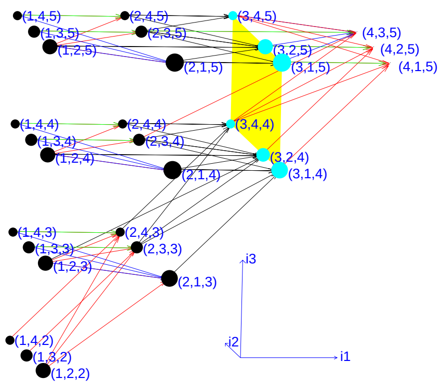
Another way to detect loop parallelism is traversing the iteration space in the *data-flow*, *loop-wise* or *plane-wise* order. The execution of the loop is animated through step-by-step highlighting the traversal iterations presently executed as well as the past and the future iterations.

4.1 Traversing ISDG in data-flow order.

The data-flow orders of iterations are calculated as the following. Since the ISDG is a directed acyclic graph, that is, each iteration $\mathbf{i}$ can not start execu-

```
  do i1=1,n
    do i2=1,n
      if(i1.ne.i2) then
S1      f=a(i2,i1)/a(i1,i1)
C$doisv
        do i3=i1+1,n+1
S2        a(i2,i3)=a(i2,i3)-f*a(i1,i3)
        enddo
      endif
    enddo
  enddo
```

(a) Gauss-Jordan elimination



(b) Its ISDG(n=4)

Figure 1. (a) the loop to be visualized; (b) The graph showing all types of dependence, the data-flow execution at the third sequential step highlights a plane $i_1 = 3$.

tion until all its precedent iterations $prec(\mathbf{i}) = \{\mathbf{j} \mid (\mathbf{j}, \mathbf{i}) \in \mathcal{E}\}$ in the dependence edges $\mathcal{E}$ have been executed. Therefore, the optimal schedule time $T(\mathbf{i})$ for each iteration $\mathbf{i}$ is recursively defined as the topology order of the graph:

$$T(\mathbf{i}) = \begin{cases} 1 + max\{T(\mathbf{j}) \mid \mathbf{j} \in prec(\mathbf{i})\}, & for \;\; prec(\mathbf{i}) \neq \{\} \\ 1, & for \;\; prec(\mathbf{i}) = \{\} \end{cases} \tag{3}$$

In principle, if the loop is executed by a data-flow machine, the data-flow parallelism is defined as the total number of iterations divided by the number of data-flow steps. Therefore, the number of data-flow steps tells the programmer how much potential parallelism lies in the loop.

For example, the Gauss-Jordan elimination has a 3-level loop as shown in Figure 1(a). There is a conditional statement in the non-perfectly nested loop. A non-perfect-to-perfect conversion is automatically applied to visualize an equivalent perfectly nested loop, as shown in Figure 1(b).

After detecting the data-flow parallelism, the tool will find a proper parallel execution by traversing the parallel loop iterations or traversing the parallel planes. The traversal of parallel iterations detects if there are any dependences carried by a certain loop; the traversal of parallel planes finds a unimodular transformation which makes the loop parallel while keeping the lexicographical ordering of dependent iterations.

4.2 Traversing parallel loop iterations

If all the dependences are not carried by a loop as testified, the loop can run in parallel, otherwise the iterations of the loop must traverse sequentially.

The tool not only testifies the loop parallelization specified by the programmer, but also judges the parallelizability of each loop automatically. Each loop can be run as a parallel DOALL loop or a sequential DO loop. Therefore for the 3-level nested loops, there are 8 possible DOALL/DO combinations. It testifies all the combinations in the outer-first parallelizing order to detect as much coarse grain parallelism as possible.

The amount of loop parallelism revealed by the loop parallelization is reported once it's testified. If it is equal to the data-flow parallelism of the data-flow execution, the detected loop parallelization has realized the data-flow parallelism under given loop boundaries.

For example, the ISDG of Gauss-Jordan elimination loop in Figure 1(b) shows that the two innermost loops can be parallelized, which has also revealed as much parallelism as the data-flow execution.

4.3 Defining and traversing planes

A *plane* in a 3D iteration space is defined by the equation:

$$ai_1 + bi_2 + ci_3 = d \tag{4}$$

where a, b, c, d are any integers. The iteration space is divided into three subspaces by the plane: $\{(i_1, i_2, i_3) \mid ai_1 + bi_2 + ci_3 < d\}$, $\{(i_1, i_2, i_3) \mid ai_1 + bi_2 + ci_3 = d\}$ and $\{(i_1, i_2, i_3) \mid ai_1 + bi_2 + ci_3 > d\}$.

Given a, b, c, there are a number of parallel planes with d ranging from $min(ai_1 + bi_2 + ci_3)$ to $max(ai_1 + bi_2 + ci_3)$. Traversing these planes, one may find parallel *partitions* without inter-plane dependences or sequential *wavefronts* without intra-plane dependences. In other words, the partitions corresponds to the parallel outermost loop, the wavefronts corresponds to the sequential outermost loop with the two inner loops parallel. If neither partitions nor wavefronts exist, the loop can not parallelize without an index reordering transformation.

The tool distinguishes the three cases by filtering dependence according to the specified plane: hiding dependence edges between different planes may reveal partitions if the edge set of the graph becomes empty; hiding dependence edges within all the parallel planes may reveal wavefronts if the edges set of the graph becomes empty.

Traversing the planes gives one more insights into inter-partition or intra-wavefront parallelism. For Gauss-Jordan elimination in Figure 1(a), the planes $1 \leq i_1 \leq 4$ are wavefronts, meaning that the outermost loop is sequential, the two inner loops are parallel.

4.4 Extracting distance matrix and evaluating loop transformations

The tool is able to extract a distance matrix from the smallest basis cone of all the distance vectors in the iteration space. It is not only useful for judging the parallelizability of the outermost or the innermost loops, but also for finding a proper unimodular transformation.

A unimodular or non-singular loop transformations correspond to coordinate transformations of the iteration space. As long as the transforming matrix T is specified from either the plane-wise traversal or the distance matrix, the new iteration coordinates are recalculated as $\mathcal{N}' = \{\mathbf{j} | \mathbf{j} = \mathbf{i}T\}$.

However the edges set $\mathcal{E}'$ of the transformed iteration space is not easy to obtain without executing the transformed program. This difficulty is solved by reusing the trace records generated from the program and construct the graph according to the new coordinate system. If the new dependence graph is inconsistent to the dependence constraints, the illegal transformation is rejected and the dependence graph is restored. Rapid prototyping of the loop transformation is useful to testify and evaluate the parallelism in the transformed program without rewriting and execution of the transformed program.

5 Application

The tool has been used to detect parallelism in a CFD program that is hard to parallelize by a traditional compiler. This program has a 3-level loop nest whose computation consumes most execution time. The loop body contains 176.5 array references in average for each iteration that has to be analyzed at run-time. To avoid the graph being over-crowded, the run-time generated ISDG was zoomed to show the first $N = 4$ part of the whole iteration space so that there are $N^3 = 64$ visible iterations, see Figure 2(a).

To detect the parallelism, the iteration space was traversed in data-flow order and there are 19 sequential steps for the 64 iterations. Testing the data-flow execution for dimensions $N = 5, 6, \ldots$, the sequential steps were found to be $25, 31, \ldots$, as shown in Figure 2(c), or $6N - 5$ in general. Therefore, there is $N^3/(6N - 5) \approx N^2/6$ data-flow parallelism, as shown in Figure 2(d).

Looking for intra-plane wavefront parallelism, it found that the iterations in planes $6 \leq 3i_1 + 2i_2 + i_3 \leq 6N$ are parallel intra-plane, since there is no more dependence edge after hiding inter-plane dependence edges. According to these planes, and the extracted distance matrix whose rows are the bases of all the distance vectors of the dependences: $\begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix}$, it found a unimodular transformation $\begin{pmatrix} 3 & 0 & 1 \\ 2 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$.

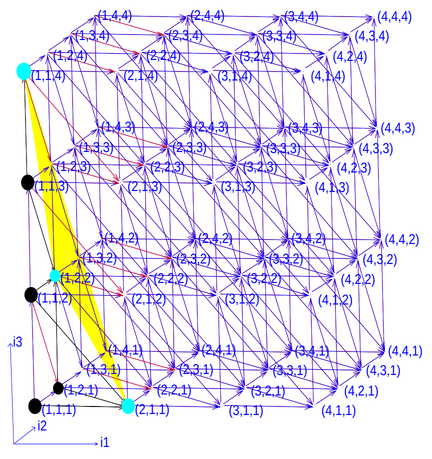
Finally, it constructed the ISDG after such a unimodular transformation without rewriting and execution of the transformed program. The reshaped iteration space after this unimodular transformation is shown in Figure 2(b). The figure shows that the reordered iterations are parallelizable for the inner two transformed loops because no intra-plane dependence exists for each plane i'_1. There are also $6N - 5$ planes as testified by the tool, thus all the data-flow parallelism has been revealed by this transformation.

6 Conclusion

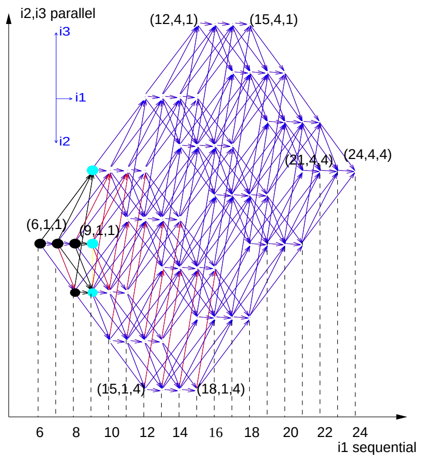
An interactive tool for exploring 3-D iteration space dependence graph of nested loops is presented. Using runtime analysis method, it helps programmers to accurately study the loop dependence and to detect parallelism of the loop. It has been applied to the difficult but significant loop nest of a CFD program, as result, a unimodular transformation is found to generate the parallelism that the compiler could not find. This web accessible tool [?] has been added into the parallel programming environment of FPT [?].

References

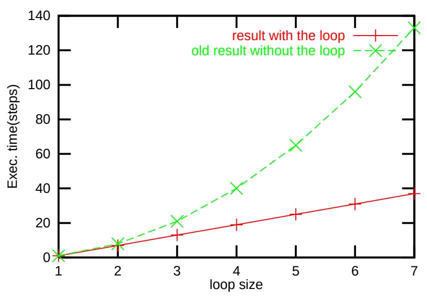
1. Utpal Banerjee. *Loop Parallelization*. Kluwer Academic Publishers, 1994.
2. J. Ramanujam. Non-unimodular transformations of nested loops. In *Proceedings, Supercomputing '92*, pages 214–223, Nov 1992.
3. Jingling Xue. Unimodular transformations of non-perfectly nested loops. *Parallel Computing*, 22(12):1621–1645, February 1997.
4. William Pugh and David Wonnacott. Constraint-based array dependence analysis. *ACM Trans. on Prog. Lang. and Sys.*, 20(3):635–678, May 1998.
5. L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans. on Parallel and Distributed Systems*, 10(2):160–180, February 1999.
6. Yijun Yu. The iteration space visualizer. Technical report, ELIS, University of Ghent, Belgium, http://sunmp.elis.rug.ac.be/ppt/isv/, 1999.
7. E. D'Hollander, F. Zhang, and Q. Wang. The fortran parallel transformer and its programming environment. *J. Information Sciences*, 106:293–317, 1998.



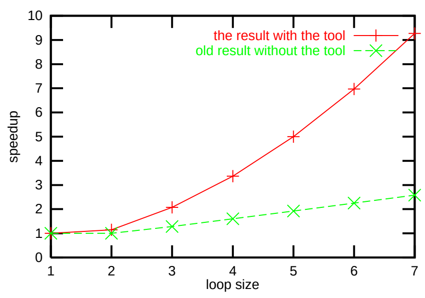
(a) the major loop



(b) the unimodular transformed loop



(c) sequential steps



(d) speedup

Figure 2. The CFD application:ISDG (a) shows the iteration space of CFD major loop with dimension $N = 4$, the current wavefront plane is $3i_1 + 2i_2 + i_3 = 9$, (b) shows the iteration space of the transformed loop, where the iterations in 19 planes form parallel loops in the new iteration space. The new plane $i_1' = 9$ is highlighted with 3 parallel iteration nodes. Comparison between 2-D and 3-D transformation (c) The number of sequential steps is the minimum execution time of the parallel program. (d) The speedups are the sequential execution time divided by the minimum parallel execution time.