Journal of Visual Languages and Computing (2001) **12,** 163–181
doi:10.1006/jvlc.2000.0191 available online at http://www.idealibrary.com on IDEAL®

Loop Parallelization using the 3D Iteration Space Visualizer

YIJUN YU AND ERIK H. D'HOLLANDER

Department of Electronics and Information Systems, University of Ghent, Belgium, {yijun,edh}@elis.rug.ac.be

Received 14 May 2000; revised 31 August 2000; accepted 18 September 2000

A 3D iteration space visualizer (ISV) is presented to analyze the parallelism in loops and to find loop transformations which enhance the parallelism. Using automatic program instrumentation, the iteration space dependency graph (ISDG) is constructed, which shows the exact data dependencies of arbitrarily nested loops. Various graphical operations such as rotation, zooming, clipping, coloring and filtering, permit a detailed examination of the dependence relations. Furthermore, an animated dataflow execution shows the maximal parallelism and the parallel loops are indicated automatically by an embedded data dependence analysis. In addition, the user may discover and indicate additional parallelism for which a suitable unimodular loop transformation is calculated and verified. The ISV has been applied to parallelize algorithmic kernel programs, a computational fluid dynamics (CFD) simulation code, the detection of statement-level parallelism and loop variable privatization.The applications show that the visualizer is a versatile and easy to use tool for the high-performance application programmer.
© 2001 Academic Press

Keywords: program visualization, dependence analysis, loop transformations, iteration space dependence graph, program instrumentation.

1. Introduction

THE EXTRACTION of parallelism in ordinary programs has been the topic of research for about three decades. In the majority of cases the techniques focus on two basic steps: dependence analysis and program transformations. Most useful parallelism comes from repetitive program tasks which can be assigned to different processors, e.g. the iterations of a loop nest. In this case, the basic task is one iteration of a parallel loop. Depending on the required granularity, the parallel iterations are selected from the outermost parallel loop, e.g. for multiprocessors, or from the innermost parallel loop, e.g. for vectorization and pipelined instruction-level parallelism.

Despite the great steps forward in this area, there still remain many loops with parallelism 'obvious' to the programmer, but which is difficult to detect using algorithmic techniques. The contrary is also true: the sophisticated dependence techniques and the construction of loop transformations and statement mappings are beyond what the programmer is able to see at first glance. Consequently, both approaches are complementary and have their own merits.

This paper focuses on the graphical support for an interactively parallel program development. Basically, it assists the user by showing the exact dependence which prevent parallel loops and it allows the user to perform program transformations which enhance the parallelism. The visualization tool shows a three-dimensional iteration space, which can be freely rotated and zoomed. Dependencies are shown or hidden, for all or a few variables, and the parallel loops can be detected. If the user sees a specific progression in the iteration space which enhances significantly the parallelism, he can mark a progression plane. The corresponding loop transformation is calculated and the dependencies within a plane and between planes can be selectively highlighted. From this information the parallel code is constructed. In order to assist the search for parallelism, the iteration space visualizer (ISV) indicates the dataflow execution which shows the minimal execution time and the maximal obtainable parallelism. The ISV has been used to interactively parallelize both common loops of standard algorithms as well as real-world CFD-code. The visualizer is written in Java, because it makes the tool platform independent, allows a web-based access and good graphics support.

The remainder of the paper is organized as follows. In the next section the definitions of the iteration space dependence graph and its construction are explained. In Section 3, the graphical features of the iteration space visualizer aimed at dependence analysis and parallelism detection are shown. In Section 4, unimodular loop transformations and statement reordering are explored for enhancing parallelism. The results of the ISV for parallelizing a number of applications are given in Section 5 and the related work is discussed in Section 6. Finally, Section 7 concludes the paper.

2. Iteration Space-Dependence Graph

In order to extract parallelism from the loops interactively, the dependencies among the loop iterations must be exposed to the programmer. The object to be visualized is called the Iteration Space-Dependence Graph (ISDG).

Consider an m-fold nested loop, $l = 1, \dots, m$, with index variables $\mathbf{i} = (i_1, \dots, i_m)$, lower and upper bounds L_l and U_l:

$$\begin{array}{l} \text{do } \mathbf{i} \in \mathcal{N} \\ \quad A(f(\mathbf{i})) = \cdots \\ \qquad \cdots \\ \quad \cdots = A(g(\mathbf{i})) \\ \text{enddo} \end{array} \tag{1}$$

The iteration set $\mathcal{N}$ is given by

$$\mathcal{N} = \{\mathbf{i} = (i_1, \dots, i_m) \mid 1 \le l \le m : L_l \le i_l \le U_l\}. \tag{2}$$

In sequential loops, iteration $\mathbf{i}$ executes before iteration $\mathbf{j}$ if $\mathbf{i}$ is lexicographically less than $\mathbf{j}$, denoted as $\mathbf{i} \prec \mathbf{j}$, i.e. there is a $k \in [1, m]$ such that $i_l = j_l$, $l = 1, \dots, k-1$, and $i_k < j_k$.

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 two iterations $\mathbf{i}_1 \prec \mathbf{i}_2$ access the same array element and at least one iteration performs a write, there is a loop-carried dependence between the iterations $\mathbf{i}_1$ and $\mathbf{i}_2$, denoted as $\mathbf{i}_1 \, \delta \, \mathbf{i}_2$.

The dependence set is defined as

$$\mathscr{E} = \{(\mathbf{i}_1, \mathbf{i}_2) \,|\, \mathbf{i}_1, \mathbf{i}_2 \in \mathscr{N} \wedge \mathbf{i}_1 \delta \, \mathbf{i}_2\}. \tag{3}$$

The directed dependence edge is classified as

- flow dependence: a write in $\mathbf{i}_1$ followed by a read in $\mathbf{i}_2$;
- output dependence: a write in $\mathbf{i}_1$ followed by a write in $\mathbf{i}_2$;
- anti-dependence: a read in $\mathbf{i}_1$ followed by a write in $\mathbf{i}_2$;

For example, in Loop (1), there is a flow dependence if $f(\mathbf{i}_1) = g(\mathbf{i}_2)$, an output dependence if $f(\mathbf{i}_1) = f(\mathbf{i}_2)$; and an anti-dependence if $f(\mathbf{i}_2) = g(\mathbf{i}_1)$.

The iteration space-dependence graph is now defined as the directed acyclic graph $\langle \mathscr{N}, \mathscr{E} \rangle$ with nodes $\mathscr{N}$ representing iterations and edges $\mathscr{E}$ representing the dependencies.

As an example, consider the following program:

```
parameter(n = 4)
real a(0:n + 1, 0:n + 1, 2)
do i = 1, n
  do j = 1, n
    do k = 1, 2
       if (k.eq.1) then
        a (i, j, k) = (a(i – 1, j, k) + a(i + 1, j, k))/2
       else
        a (i, j, k) = (a(i, j – 1, k) + a(i, j + 1, k))/2
       endif
    enddo
  enddo
enddo
end
```

The iteration space-dependence graph (Figure 1) is extracted from the program in three steps,

(1) instrumenting the program;
(2) executing the instrumented program;
(3) constructing the ISDG from the trace of the execution.

The program is instrumented to generate the following output:

- at the start of an iteration: the iteration counter, id, and the loop indices, indices;
- at a read or write access: the iteration counter, id, the type of reference, ref = R or W, the variable name: variable, and the subscript values: subscripts.

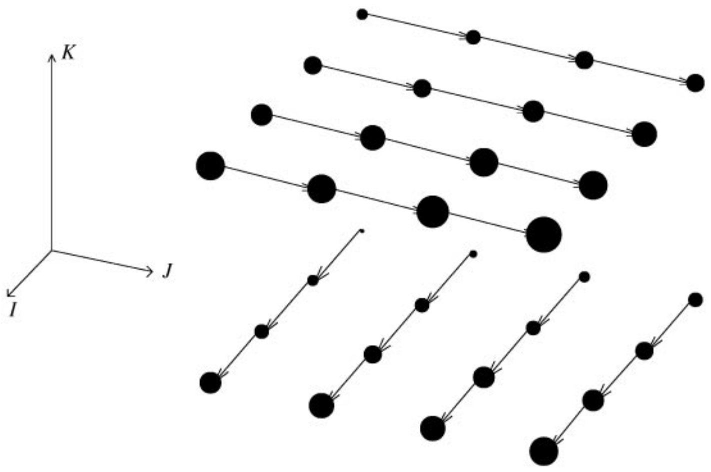

Figure 1. The ISDG of the example program, from which one can easily recognize that the size of the iteration space is $4 \times 4 \times 2$. The 32 iterations belong to eight independent partitions

Scalar variables are treated as one-dimensional arrays with a single element. Non-perfectly nested loops are converted to perfectly nested loops similar to the approach in [1].

After executing the instrumented program, the ISDG graph is constructed. First, an empty list of read or write references is created for each memory location. Then the trace records are processed as follows.

(1) Every read or write reference is appended to the reference list of the memory location addressed by the subscripts.
(2) Dependence edges are constructed according to the following rules:

- a read reference creates a flow dependence with the preceding write into the same location;
- a write reference creates an output dependence with the preceding write into the same location;
- a write reference creates an anti-dependence with all the reads since the preceding write into the same location.

3. Dependence Analysis

Having constructed the iteration space-dependence graph, this section first explains the graphical features of the ISDG and then shows how to use them effectively to analyze data dependencies.

3.1. Loop Visualization

Consider an m-level deep nested loop.

- If $m = 3$, the iteration space-dependence graph is displayed in 3D corresponding to the iteration indices of the three loops.

- If $m < 3$, a 2D view is available.
- If $m > 3$, three loop indices must be selected from the hyper-dimensional iteration space and the ISDG is projected onto a 3D space.

The size of the iteration spheres are proportional to the distance from the viewer so that the programmer can recognize the spatial relationship between the adjacent iterations.

Furthermore, the programmer can arbitrarily choose the size of the spheres either to clearly indicate the iterations or to emphasize the dependence edges.

The graph can be zoomed in or out easily by resizing the window. It can be clipped by changing the visible index range. This helps the programmer to examine the regularity of the dependence patterns.

Optionally, the iteration indices can be displayed next to the iteration nodes. Grid lines are available to show the shape and structure of the iteration space.

The graph can be rotated freely in three directions by changing the viewpoint angle. The rotation can be done by dragging the mouse, by selecting an animated rotation, or by directly specifying the *X–Y–Z* angles. The index-axes show the direction of the three loops. The axes can also be dragged anywhere in the canvas.

Each directed edge represents the dependence between the connected iterations. Three colors (red, green and blue) classify the edges into flow-, output- and anti-dependencies, respectively. The programmer can click on any visible edge to find out the source and target loop indices of the selected edges.

Dependencies can be selectively hidden by the dependence type and/or loop variable names. The filter feature is useful to focus on the individual variables, to study the algorithmic data dependencies, i.e., the flow dependencies; or the shared-memory-originated dependencies, i.e., the anti- and output-dependencies. Memory-originated dependencies can be eliminated by variable privatization or scalar expansion [2]. Similarly, filtering variables from the ISDG can clarify the cause of the loop dependencies.

To allow the high-resolution print of the graphics implemented in the visualizer, a color Postscript interface is defined.

3.2. Detecting and Enhancing Program Parallelism

The runtime behavior of the loops is shown by simulating the program execution in different kinds of iteration order. The traversal of the iteration space can be driven by sequential loop execution, dataflow execution, parallel-loop execution and plane execution. During the simulated execution, the color of the nodes distinguishes the past, present and future iterations. The following subsection explains the difference between these execution orders and discusses the use of these features.

3.2.1. Sequential Execution: the Lexicographical Order

The trace from the program execution is ordered lexicographically. In the ISDG, the iteration nodes are highlighted one-by-one by clicking the mouse and the total number of the iterations is reported.

3.2.2. Dataflow Execution: the Maximal Parallelism

In a dataflow execution, each iteration is executed as soon as its data are ready, i.e. after the dependent iterations are all carried out. By clicking at an empty area of the canvas, the highlighted nodes show the parallel executable iterations in each time step when every iteration is assigned to different processor. This corresponds to a minimal execution time with the maximum parallelism exploited. Although the dataflow execution normally does not follow the iteration order expressed by parallel DOALL loops, it reflects the maximum speedup obtainable within this loop nest. This maximum speedup is shown to the programmer.

3.2.3. Parallel Loop Execution: the Automatic Parallelization

When one or several loops are executed in parallel, the iterations in the parallelized loops can run in one step and the iterations in the sequential loops must run one-by-one.

According to the selected dependencies in the ISDG, the visualizer checks all the combinations of loops to find the coarsest grain of DOALL loop parallelism automatically. When the DOALL loops are found, the speedup is reported by calculating the ratio between the sequential time and the parallel execution time.

The automatic loop parallelization feature relieves the programmer of further analysis when enough parallelism is obtained, e.g. compared with the dataflow execution.

The programmer may also interactively specify which loops are to be checked for parallel execution. In that case, blinking edges warn for critical dependencies that prevent the attempted loop parallelization.

After being verified by the parallel check, the DOALL loops will be enabled for parallel traversal of the ISDG. By clicking at the empty area of the canvas, the programmer can see what happens after the parallelization: how much parallelism or speedup can be obtained by the automatic parallelization.

When the automatic parallelization shows less parallelism than the dataflow execution, some transformations of the loop should be considered to enhance the parallelism. Therefore, the plane traversal is provided to find such a suitable loop transformation.

3.2.4. Plane Execution: Finding More Loop Parallelism

It is possible to specify any cutting plane by clicking on three nodes that are not on one line. The cutting plane $Ax + By + Cz = D$ is calculated and highlighted in the ISDG as a polygon, bounded by the iteration space.

Alternatively, an experienced user can specify the plane by giving the four integer parameters A, B, C and D.

When the cutting plane is defined, a mouse click starts the execution of the loop such that all iteration nodes in the plane are executed in parallel. At each click, the cutting plane progresses sequentially through the iteration space in a number of steps corresponding to the parallel execution time.

Plane parallelization requires that there are no dependencies between the iterations in the plane. This can be checked

- by hiding the dependencies between the planes, or
- by projecting the 3D iteration space onto a 2D executing plane.

In summary, the programmer may apply the following procedure to interactively find and enhance the parallelism of a program:

(1) detect the maximal parallelism possible, by watching a dataflow execution;
(2) apply automatic parallelization to parallelize as much loops as possible;
(3) hide the false dependencies and the dependencies caused by private variables such that the pruned ISDG allows for more loops parallelization;
(4) do a plane execution if the loop parallelism is still less than the dataflow parallelism; if a suitable plane traversal is found, calculate the corresponding loop transformation.

4. Program Transformations

In this section the unimodular loop transformations and statement reordering to amplify the parallelism are discussed.

4.1. Unimodular Loop Transformations

A unimodular matrix $\mathbf{T}$ specifies a one-to-one mapping between two loop iteration spaces. Consequently, a unimodular transformation can be applied to re-orient the ISDG in such a way that more parallelism can be extracted.

A unimodular matrix $\mathbf{T}$ has $|\,det(\mathbf{T})| = 1$ and the mapping between the loop indices $\mathbf{i}$ and $\mathbf{i}'$ is described by

$$\mathbf{i}' = \mathbf{i}\mathbf{T}. \tag{4}$$

Generally, the loop boundaries are changed after a unimodular transformation, and need to be recalculated. Furthermore, the transformation may change the lexicographical ordering of the dependent iterations. For example, if $\mathbf{i} \leq \mathbf{j}$ and $\mathbf{i}' > \mathbf{j}'$ then the dataflow dependence becomes an anti-dependence, and therefore the loop transformation is invalid. However, the correctness of a proposed loop transformation is checked.

To find the unimodular loop transformation which engenders a plane execution in the outermost loop, the normal vector (A, B, C) of the plane is placed into the first column of the 3D unimodular transformation matrix $\mathbf{T}$. The other two columns need to be chosen such that (1) the matrix is unimodular and (2) the inner loops of the transformed loop nest execute the dependent iterations in lexicographical order. Different unimodular solutions are possible, and the viewer will indicate the valid loop transformations. After the unimodular transformation, the new independent loop (either outermost or innermost) can be parallelized. The corresponding loop boundaries can be calculated using integer programming tools like the Omega calculator [3].

In case of linear array subscripts, a suitable loop transformation can be found based on the pseudo-distance vectors as described in [4]. This method generates a parallelizing unimodular transformation automatically and is also available in the viewer.

4.2. Loop Projections

Regarding the dimensionality of the visual space, there are three kinds of index mappings: from a 3D loop space to a 3D visual space is a 1–1 mapping, used for unimodular and non-singular loop transformations; from $>3D$ to 3D is a projection useful to analyze higher-dimension loops; from a $<3D$ to 3D is a dimension expansion, useful for treating the parallel execution by statement reordering transformations.

The schemes discussed so far can be extended to non-perfectly nested loops after suitable transformations such as statement reordering by affine mappings proposed by Kelly and Pugh [5], including loop fusion, loop fission, etc.

In order to handle statement-level parallelism, the statements are given a dummy index and the corresponding loop iterates through all the statements. This allows to integrate a statement-level program-dependence graph (PDG) into the framework of the ISDG.

Extending the ISDG with statements dependencies, a suitable affine mapping like unimodular transformation on non-perfectly nested loops can be found [1]. In the next section it is shown that for two examples in the recent paper of Lim and Lam [6], the extended loop iteration space allows to use unimodular transformations to find statement-level parallelism.

5. Applications and Results

To apply the visualizer, the instrumentation can be done by adapting front-end compilers, such as FPT [7] for Fortran programs and in SUIF [8] for C programs. The ISV instrumentation has been carried out for both compilers. A pragma C$doisv in Fortran or #pragma doisv in C before the selected innermost loop is the only required modification to the source program to obtain the trace-generating code.

The visualization itself is written in Java so that it is portable and web-ready. All the above instrumentation and visualization tools have been integrated into a web-based environment that takes the source program as input and yields an applet, visualizing the iteration space dependence graph [9].

The applet has been applied to several application programs and kernel loops. The parallelism has been detected visually and the suitable program transformations were found interactively. Note that the applet applies to the submitted program; it is the programmer's responsibility to verify the extensibility of the results found by the applet in particular to a different size of the loop region.

5.1. Non-perfectly Nested Loop

The following program shows the well-known Gauss Jordan (GJ) elimination to explain the approach to find parallelism in programs. GJ is an example of a 3D non-perfectly nested loop, since there is an assignment statement out of the *k* loop body.

```
do i = 1, n
  do j = 1, n
    if(i.ne.j) then
      f = a(j, i)/a(i, i)
```

```
C$doisv
        do k = i + 1, n + 1
          a (j, k) = a (j, k) − f*a (i, k)
        enddo
      endif
    enddo
  enddo
```

Pragma C$doisv is inserted before the *k* loop to indicate which iteration space should be instrumented. The program instrumented by FPT writes trace records into an ASCII file serving as the input for the ISDG construction.

The ISDG (Figure 2) displays all types of dependence. By running the viewer, the user can verify that the highlighted plane along the *i* axis cuts through exactly the same iterations as the dataflow execution. This confirms that both *j* and *k* loops are maximally parallelizable.

5.2. Statement Reordering

In Lim and Lam [6], an example of a doubly nested loop is illustrated for statement reordering.

```
     do I1 = 1, n
       do I2 = 1, n
S0:     a(I1, I2) = a (I1, I2) + b(I1 − 1, I2)
S1:     b(I1, I2) = a (I1, I2 − 1)*b(I1, I2)
       enddo
     enddo
```

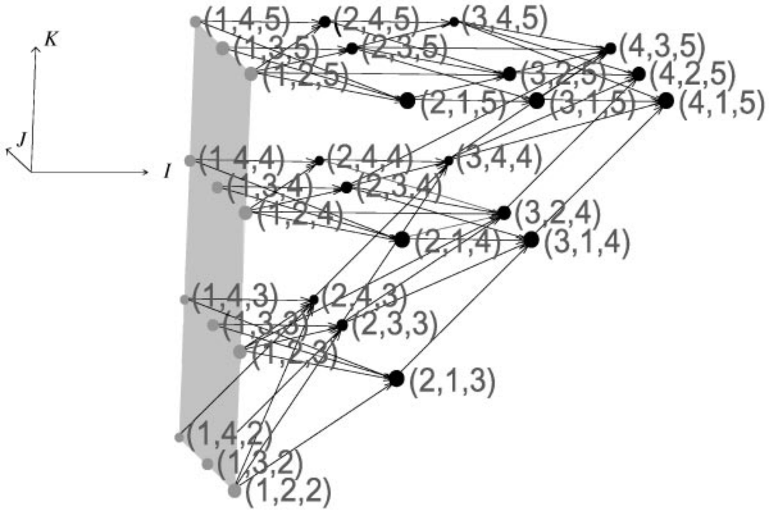

Figure 2. The ISDG of the Gauss Jordan elimination indicating the dependencies and the highlighted plane $I = 1$ with parallel iterations. The sequential time shows 30 sequential iterations while the data-flow time shows four dataflow steps. Therefore, the potential speedup is 7.5. Since executing the loops J, K in parallel is valid, the DOALL execution yields the same speedup as the dataflow execution

The following adapted program uses the statement number as an additional loop index l_3.

```
      do l1 = 1, n
        do l2 = 1, n
  c$doisv
          do l3 = 0, 1
            if (l3.eq.0) a (l1, l2) = a (l1, l2) + b (l1 − 1, l2)
            if (l3.eq.1) b (l1, l2) = a (l1, l2 − 1)*b (l1, l2)
          enddo
        enddo
      enddo
```

With the extra dimension, a 3D iteration space is obtained in Figure 3.

The planes $l_1 - l_2 + l_3 = D$ traverse the iteration space in the same way as the dataflow execution. Using a unimodular matrix

$$\begin{pmatrix} 1 & 1 & 0 \\ -1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

the same plane traversal can be obtained, leading to a parallel i_1 loop (see the transformed ISDG in Figure 4).

The Fourier–Motzkin method is used to calculate the new loop bounds, giving the following program:

```
      DOALL i1 = 1 − n, n
        do i2 = max(i1, 1), min(n, i1 + n)
  C$doisv
          do i3 = max( − i1 + i2, 1), min( − i1 + i2 + 1, n)
            l1 = i2
            l2 = i3
            l3 = i1 − i2 + i3
            if(l3.eq.1) a(l1, l2) = a(l1, l2) + b(l1 − 1, l2)
            if(l3.eq.2) b(l1, l2) = a(l1, l2 − 1)*b(l1, l2)
          enddo
        enddo
      enddo
```

In order to optimize the code generation, the constraint of Lim's mapping and the unimodular mapping are given to the Omega calculator [3, 10] as the following:

```
# statement ordering mapping
symbolic n;
IS1:= {[i, j]: 1 <= i, j <= n};
IS2:= {[i, j]: 1 <= i, j <= n};
T1:= {[i, j] → [i − j, i, j, 1]};
```

```
# unimodular mapping
symbolic n;
IS1:= {[i, j, k]: 1 <= i, j <= n && k = 0};
IS2:= {[i, j, k]: 1 <= i, j <= n && k = 1};
T1:= {[i, j, k] → [i − j + k, i, j]};
```

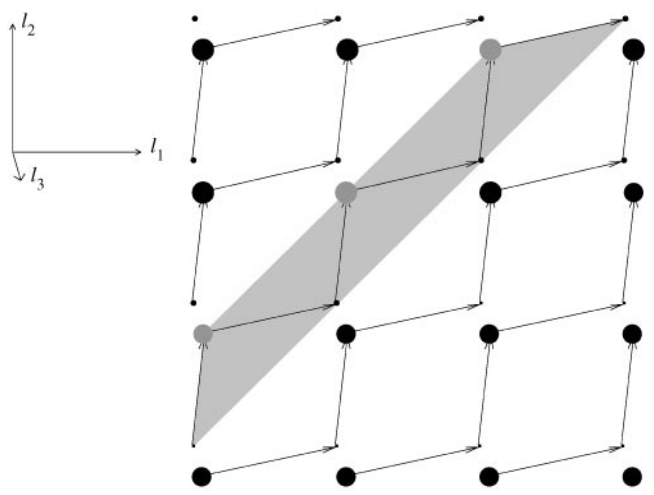

Figure 3. The ISDG of the expanded Lim's loop is visualized in 3D space. Sequential execution requires 32 steps, while dataflow execution needs seven steps. Therefore, the maximum speedup is 32/7 = 4.57. The i_3 loop is automatically verified as a DOALL where its parallel execution requires 16 steps, yielding speedup 2.0. The highlighted plane $l_1 - l_2 + l_3 = 0$ is selected by clicking at three iteration points (1, 1, 0), (1, 2, 1) and (2, 2, 0)

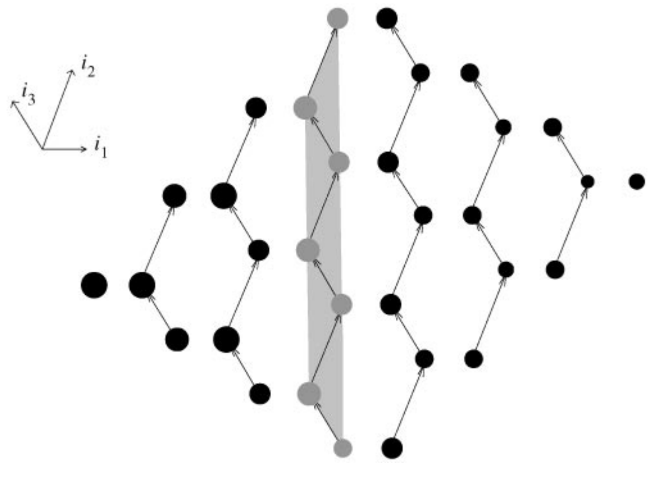

Figure 4. The ISDG after the unimodular transformation. The transformed outermost loop i_1 is a DOALL. The highlighted plane $i_1 = 1$ shows the largest partition which contains 7 sequential steps, whereas the sequential execution takes 32 steps. Therefore, the parallel plane execution has a speedup of 4.57

```
T2:= {[i, j] → [i − j + 1, i, j, 2]};        T2:= {[i, j, k] → [i − j + k, i, j]};
codegen 0 T1 : IS1, T2 : IS2;                codegen 0 T1 : IS1, T2 : IS2;
```

The affine functions T1, T2 map two statements $S1$, $S2$ to their processor id, where IS1, IS2 are the iteration space constraints for $S1$, $S2$, respectively. The statement reordering mappings found by Lim [6] is on a two-dimensional iteration space (i, j), while the unimodular mappings found by the ISV is on a three-dimensional iteration space (*i*, *j*, *k*) which has a dimension *k* for the statements. The optimized code obtained from unimodular transformation is the same as in [6].

```
DOALL p = 1 − n, n
  if (p.ge.1)b(p, 1) = a(p, 0)*b(p, 1)
  do l1 = max(p + 1, 1), min(p + n − 1, n)
    a(l1, l1 − p) = a(l1, l1 − p) + b(l1 − 1, l1 − p)
    a(l1, l1 − p + 1) = a(l1, l1 − p)*b(l1, l1 − p + 1)
  enddo
  if (p.le.0)a(p + n, n) = a(p + n, n) + b(p + n − 1, n)
enddo
```

Having the branch statements removed, the optimized code has parallel *p* loop.

5.3. High-level Nested Loop

Cholesky is one of the seven kernel subroutines in the NASA7 program of the SPECfp92 benchmarks. It contains two 4-level nested non-perfectly nested loops. The 4-level perfectly nested loop converted from the standard Cholesky program is shown here.

```
      do i = 0, nrhs
        do k = 0, 2*n + 1
          if (k.le.n) then
           i0 = min(m, n − k)
          else
           i0 = min(m, 2*n − k + 1)
          endif
          do j = 0, i0
C$doisv
           do l = 0, nmat
             if (k.le.n) then
               if (j.eq.0) then
          8    b(i, l, k) = b(i, l, k)*a(l, 0, k)
               else
          7    b(i, l, k + j) = b(i, l, k + j) − a(l, − j, k + j)*b(i, l, k)
               endif
             else
               if (j.eq.0) then
          9     b(i, l, k) = b(i, l, k)*a(l, 0, k)
               else
          6     b(i, l, k − j) = b(i, l, k − j) − a(l, − j, k)*b(i, l, k)
               endif
             endif
           enddo
          enddo
        enddo
      enddo
```

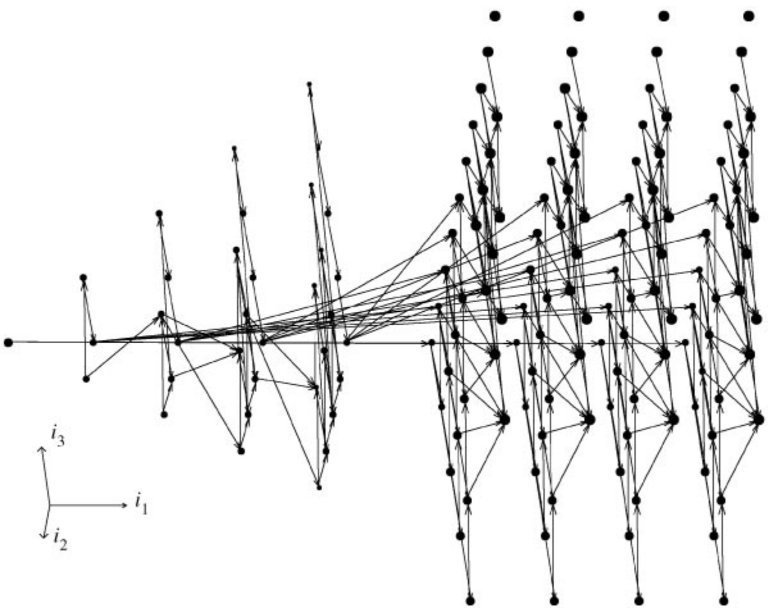

Figure 5. The 4D ISDG of the Cholesky loop shows the projected 3D view of the (i_1, i_2, i_3) loops. The dependencies between the left and right parts of the combined iteration space along i_1, i_2, i_3 directions prevent the parallelization of these three loops

The ISDG obtained from the instrumented program trace contains four loop indices, which can be projected to four 3D views: (i_1, i_2, i_3), (i_1, i_2, i_4), (i_1, i_3, i_4) and (i_2, i_3, i_4). The projection (i_1, i_2, i_3) of the ISDG is shown in Figure 5.

In this projection, no parallel loops can be detected. However, the (i_1, i_2, i_4) view (Figure 6) shows that the i_4 loop always iterates through parallel partitions and therefore can be permuted to the outermost loop.

This is true also for other 3D projections. Thus, a parallel program like the one in Lim *et al.* [11] is obtained.

5.4. A CFD Application

In the 3D mould-filling simulation code developed by the WTCM research center [12], the majority of the computations is spent on an iterative solver of Navier–Stokes equations. Each iterative step is a 3-level kernel loop, which performs Successive Over Relaxation to solve a system of linear equations. The complexity of the iteration reference patterns (an average of 172 indirect array references per iteration, spread over 33 if-branches) makes it hard if not impossible for automatic parallelizing compilers to find a parallel loop. The ISDG of the kernel loop is shown in Figure 7.

A parallel plane is obtained by shift-clicking on three nodes in one of the dataflow execution steps. This plane cuts through the iteration space with exactly the same iterations as the dataflow execution, yielding the maximal iteration-level parallelism. The plane execution shows that there are 19 parallel planes going through the 19 dataflow steps. Projecting the ISDG to 2D, a cutting plane $3i_1 + 2i_2 + i_3 = 15$ is shown in Figure 8.

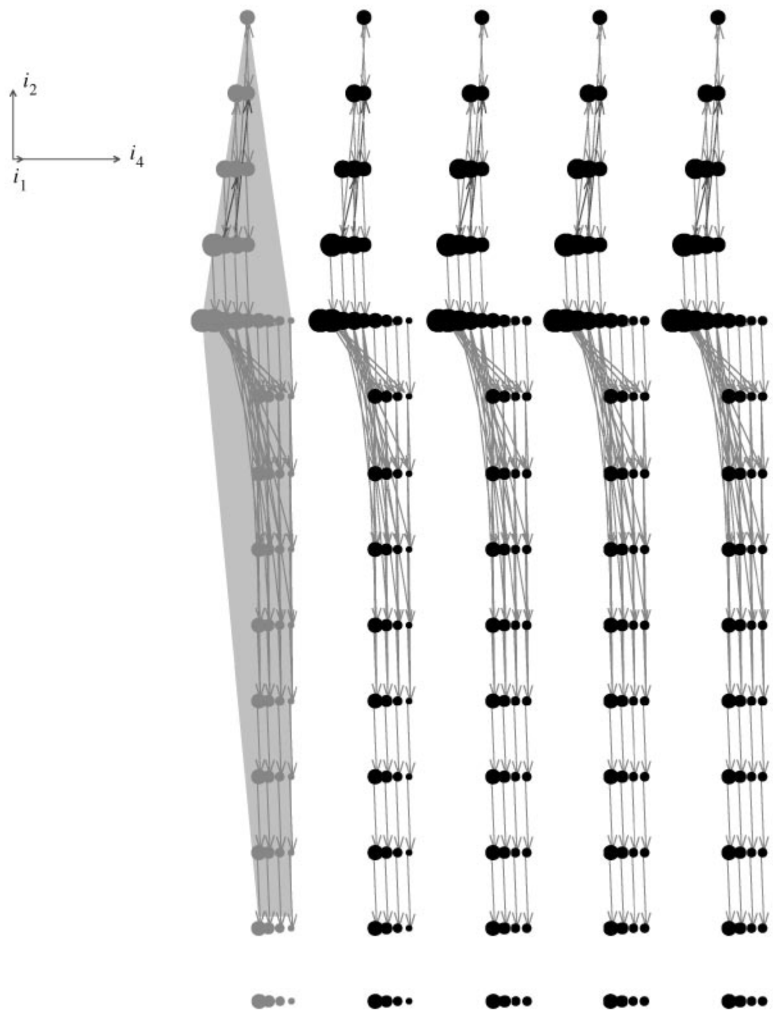

Figure 6. The same 4D Cholesky ISDG is projected to 3D space (i_1, i_2, i_4) where dimension i_1 is vertical onto the (i_2, i_4) plane. Here loop i_4 iterates through independent partitions

Independence between the iterations within each of the 19 planes allows two parallel innermost loops. However, the dependencies between the iterations of different planes requires the outermost loop to be sequential. Therefore, a unimodular transformation

$$\begin{pmatrix} 3 & 0 & 1 \\ 2 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

is obtained from the plane direction vector (3, 2, 1). The transformed ISDG is shown in Figure 9.

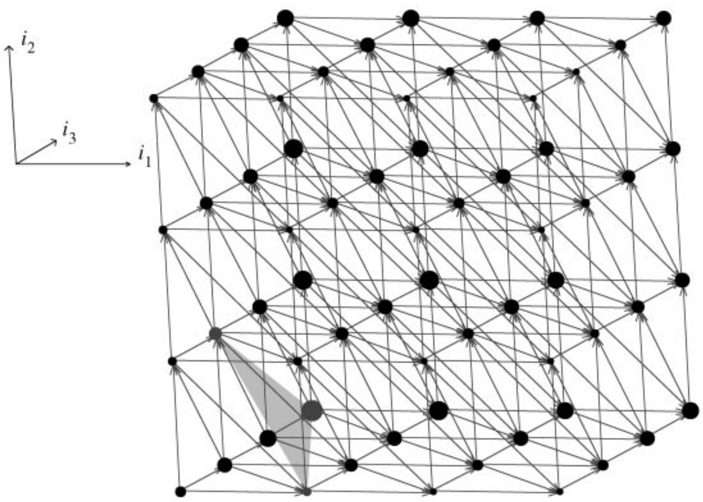

Figure 7. The ISDG of the original CFD loop with $n = 4$ is shown. The sequential execution has 64 steps while the dataflow execution has 19 steps, thus a speedup of $64/19 = 3.37$ is the maximal. This picture shows that the fourth step has three parallel iterations. Shift-clicking at the three dataflow parallel iterations, namely (1, 2, 2), (1, 1, 4) and (2, 1, 1), a cutting plane is found as $3i_1 + 2i_2 + i_3 = 9$, as highlighted

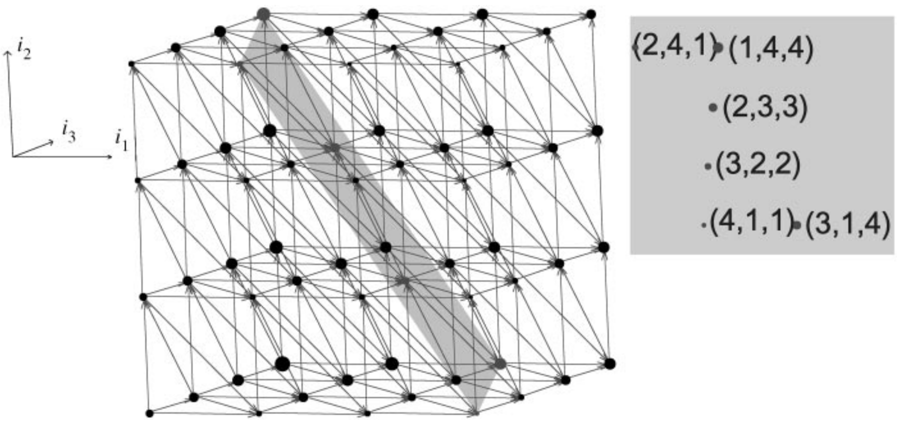

Figure 8. The ‘largest’ cutting plane as highlighted intersects the iteration space with six iterations, i.e. (1, 4, 4), (2, 3, 3), (2, 4, 1), (3, 1, 4), (3, 2, 2), (4, 1, 1). The 2D projection of this plane shows that the iterations in the plane are independent of each other

By the regularity of the calculations, we can draw the conclusion that the inner two loops are parallelized while the outermost loop has $6n - 5$ steps. Therefore, a $O(n^2/6)$ speedup is found when executing the n^3 iterations.

5.5. Using the ISV as an Education Tool

The ISV has been used during the laboratory exercises of the course ‘Parallel and Distributed Systems’, taught at the Gent University. The students had to analyze

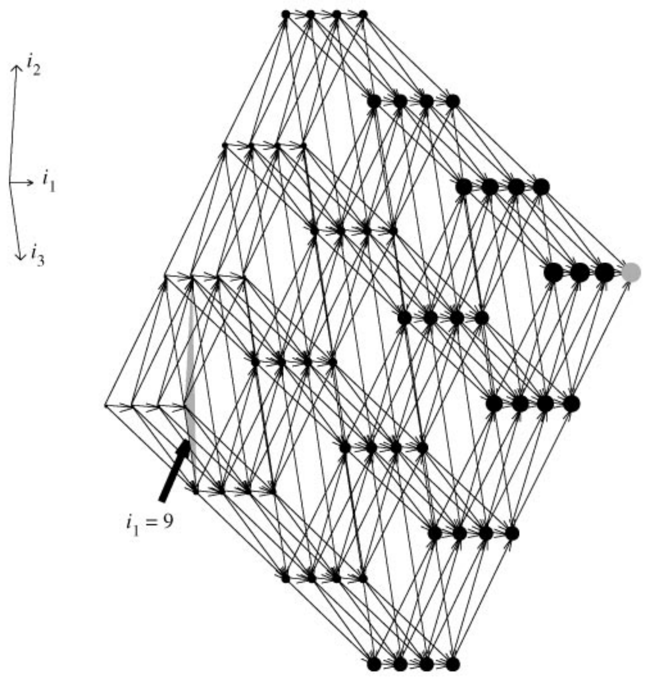

Figure 9. Performing the unimodular transformation, the new ISDG is calculated without regenerating the trace. It shows that the i_2, i_3 loops can run in parallel while the sequential i_1 loop goes through the planes $i_1 = 6$–24. The highlighted plane is corresponding to the fourth step as in Figure 7, where the three iterations there are transformed as (9, 2, 1), (9, 1, 1) and (9, 1, 2), respectively

a doubly nested loop for the distance vectors, to find a suitable unimodular loop transformation, to write out the transformed code and to test it for performance gain. The program is given below.

```
PARAMETER (m = 10)
DIMENSION a(0: m + 1, 0:m + 1)
do i = 1, m
do j = 1, m
a(i, j) = 0.2*(a(i, j) + a(i, j − 1) + a(i − 1, j) + a(i + 1, j) + a(i, j + 1))
enddo
enddo
end
```

There are two distance vectors (1, 0) and (0, 1). We knew that a unimodular (column) transformation $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ yields two new distance vectors (1, 1) and (1, 0), which allow to parallelize the inner loop. We intended to guide the students to find out this 'standard' answer, given below.

```
do i1 = 2, 2*m
DOALL i2 = max(1, i1 − m), min(m, i1 − 1)
```

```
    i = i2
    j = i1 − i2
    a(i, j) = 0.2*(a(i, j) + a(i, j − 1) + a(i − 1, j) + a(i + 1, j) + a(i, j + 1))
    enddo
    enddo
```

After 1 h of paper and pencil work, 10 out of 26 students had found the answer. Then, they could verify their findings using the ISV. After using the visualizing tool for about 10 min, almost everyone obtained the transformed code.

Interestingly to note, three students unexpectedly presented us with two other transformations $\begin{pmatrix} 1 & -1 \\ 1 & 0 \end{pmatrix}$ and $\begin{pmatrix} 2 & 1 \\ 1 & 0 \end{pmatrix}$. These solutions are also correct, showing that the ISV allows to find several correct transformations for some programs. In addition, the ISV indicates that the first transformation yields a speedup of 5.26 for m = 10, while the second one sacrifices the performance with a speedup of 3.57 for m = 10.

This example shows that it is easier to convey the concept of parallel programming to the students with the help of a visualizing tool.

6. Related work

Experience of using a parallel programming environment shows that scientific programmers prefer an interactive tool to study data dependencies and program transformations [13, 14].

During the past decades, many techniques in the area of data-dependence tests [15–17] and program transformations have provided the programmer with much useful material, e.g. the Banerjee, Range [18, 19] and Omega [3, 17] tests, the unimodular [1, 20, 21] and non-singular [22] loop transformations and recently statement reordering transformations [5, 6, 11] for non-perfectly nested loops. Most techniques are illustrated by dependence graphs, such as the program-dependence graph (PDG) and the iteration space dependence graph (ISDG). The difference between the PDG and the ISDG is that the PDG emphasizes the statement-level dependencies and ISDG emphasizes the iteration-level dependencies. The ISDG makes it easier to see the effects of unimodular and non-singular-loop transformations.

Most examples in the published papers use two-dimensional graphs in order to explain techniques which can be extended to multiple dimensions. However, 2D graphs cannot easily reveal the details of real programs with more than doubly nested loops. Therefore, 3D assisting tools have entered the parallel programming scene.

For instance, in the recent paper of Sasakura *et al.* [23], a 3D visualization tool 'NaraView' is presented for studying data dependence. The visualizing approach of the authors is to linearize the iterations into a single time dimension and to layout data arrays on the other two dimensions. Their objectives are closely related to this paper. The choice of using two dimensions for array and one dimension for the linearized iteration index is useful for lifetime and variable privatization analysis. However, for analyzing iteration-level parallelism, the explicit visualization of the loop variables comes at the expense of two visualization dimensions. Therefore, the approach presented in this paper is better geared towards the visualization of the dependence distance patterns in a multi-dimensional iteration space, which is very important for loop transformations.

7. Conclusion

A 3D iteration space visualizer (ISV) is presented, which shows the exact loop dependencies and allows programmers to discover parallelism in an interactive way. The approach complements the analytical methods in the traditional automatic parallelizing compilers. The dependence analysis and program transformation tools integrated in the visualizer assist the development of parallel programs when the dependencies are too complex for the compiler to analyze or the dependence patterns show more parallelism than the compiler has exploited.

References

1. Jingling Xue (1997) Unimodular transformations of non-perfectly nested loops. *Parallel Computing* **22**(12), 1621–1645.
2. Z. Li (1992) Array privatization for parallel execution of loops, *Conference proceedings/1992 International Conference on Supercomputing*, Washington, DC, July 19–23. New York, NY, USA ACM Press, New York, pp. 313–322.
3. W. Pugh, D. Wonnacott (1998) Constraint-based array dependence analysis. *ACM Transactions on Programmer Language and Systems* **20**(3), 635–678.
4. Yijun Yu & Erik H. D'Hollander (2000) Partitioning loops with variable dependence distances, *Proceedings of the 2000 International Conference on Parallel Processing*, Toronto, Canada, August 21–24, 2000, (D.J. Lilja, ed.). IEEE Computer Society, Silver Spring, MD, pp. 209–218.
5. W. Kelly & W. Pugh (1996) Minimizing communication while preserving parallelism. *FCRC '96: Conference Proceedings of the International Conference on Supercomputing*: Philadelphia, PA, USA, May 25–28, ACM Press, New York, pp. 52–60.
6. A. W. Lim & M. S. Lam (1998) Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing* **24**(3-4), 445–471.
7. E. H. D'Hollander, F. Zhang & Qi Wang (1998) The Fortran Parallel Transformer and its programming environment. *Journal of Information Sciences* **106,** 293–317.
8. Zhi An Xu (1993) Fluid flow and thermal analysis during mould filling and solidification of castings. Ph.D. dissertation, Ghent University, Faculty of Applied Science.
9. Y. Yu (2000) A 3D-java tool to visualize loop-carried dependences. In: *Parallel Computing: Fundamentals & Applications*, *Proceedings of the International Conference ParCo'99*, Delft, The Netherlands 17-20 August 1999, (E. H. D'Hollander, J. R. Joubert, F. J. Peters & H. Sips, eds), Imperial College Press, April, pp. 730–737.
10. W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman & D. Wonnacott (1996) The Omega library interface guide. Technical Report, Department of Computer Science, University of Maryland, College Park, April.
11. A. W. Lim, G. I. Cheong & M. S. Lam (1999) An affine partitioning algorithm to maximize parallelism and minimize communication. *Proceedings of the Conference on Supercomputing* (*N.Y.*), *ACM SIGARCH*, June 20–25 ACM Press, New York, pp. 228–237.
12. Zhi An Xu (1993) Fluid flow and thermal analysis during mould filling and solidification of castings. Ph.D. Dissertation, Ghent University, Faculty of Applied Science.
13. M. W. Hall, T. J. Harvey, K. Kennedy, N. McIntosh, K. S. McKinley, J. D. Oldham, M. H. Paleczny & G. Roth (1993) Experiences using the ParaScope Editor: an interactive parallel programming tool. *ACM SIG-PLAN Notices* **28**(7), 33–43.
14. K. Zhang, T. Hintz & X. Ma (1999) The role of graphics in parallel program development. *Journal of Visual Languages and Computing* **10**, 215–243, doi: 10.1006/jvlc.1998.0109.
15. U. Banerjee (1994) *Loop Parallelization*. Kluwer Academic Publishers, Dordrecht.
16. K. Psarris (1996) The Banerjee-Wolfe and GCD tests on exact data dependence information. *Journal of Parallel and Distributed Computing* **32,** 119–138.

17. P. M. Petersen & D. A. Padua (1996) Static and dynamic evaluation of data dependence analysis techniques. *IEEE Transactions on Parallel and Distributed Systems* **7**(11), 1121–1132.
18. W. Blume & R. Eigenmann (1994) The range test: a dependence test for symbolic, non-linear expressions. *Proceedings*, *Supercomputing '94*, Washington, DC, November 14–18, IEEE Computer Society Press, Silver Spring, MD, pp. 528–537.
19. W. Blume & R. Eigenmann (1998) Nonlinear and symbolic data dependence testing. *IEEE Transactions on Parallel and Distributed Systems* **9**(12), 1180–1194.
20. U. Banerjee (1990) Unimodular transformations of double loops. *Advances in Languages and Compilers for Parallel Computing*, *1990 Workshop*, Irvine, CA, *Research Monographs in Parallel and Distributed Computing*, August MIT Press, Cambridge, MA, pp. 192–219.
21. E. H. D'Hollander (1992) Partitioning and labeling of loops by unimodular transformations. *IEEE Transactions on Parallel and Distributed Systems* **3**(4), 465–476.
22. J. Ramanujam (1992) Non-unimodular transformations of nested loops. *Proceedings*, *Supercomputing '92*, November, pp. 214–223.
23. M. Sasakura, K. Joe, Y. Kunieda & K. Araki (1999) NaraView: an interactive 3D visualization system for parallelization of programs. *International Journal of Parallel Programming* **27**(2), 111–125.