Dependences

January 16, 2010

Byline

Paul Feautrier, Emeritus Professor of Computer Science
Ecole Normale Supérieure de Lyon
46, Allée d'Italie
69364 LYON CEDEX, France
Paul.Feautrier@ens-lyon.fr

Synonyms

Hazard (in hardware)
Race (a dependence which has been ignored, and hence may cause errors)

Related Entries

Definition

One needs a paradigm shift when reasoning about parallel programs. The usual approach, as for instance in denotational semantics, is to consider each statement as a function from memory state to memory state, and to consider two program as equivalent if they implement the same function. For instance, `x = 0; x = x+1;` and `x = 1;` are deemed equivalent. However, if these two programs are run in parallel with `x = 7;` they may give different results. Similarly, asking whether a statement can be run in parallel with itself does not make sense. However, the statement may be enclosed in a context – for instance, a loop – which cause it to be executed repeatedly. Each repetition is an instance or *operation*, and it makes sense to ask whether all or some of these operations can be executed in parallel.

These considerations motivate the following definitions. A program is a specification for a set of operations. Each operation is executed only once, and must have a unique name. The program must also specify the order in which operations are executed. It is easy to see that a sequential program has a total execution order, and a parallel program has a partial execution order. In fact, an embarassingly parallel program, in which everything can be done in parallel, has the empty execution order.

Methods for specifying operations and their execution order differ from one program style to another: loop programs, recursive programs, functional programs all have different conventions.

Optimizing compilers, and in particular parallelizing compilers, try to transform the source program into an equivalent program which is better adapted to the target architecture, or runs faster, or has more parallelism. The problem is that in general, program equivalence is undecidable. A possible solution is to restrict the class of admissible transformations to a decidable subset. One of the classes of choice consists of *reordering* transformations. The operations of the transformed program are the same as in the source, but they may be executed in a different order (including parallel execution). The necessary information for validating such a transformation is the dependence relation:

Two operations u and v are independent if the programs $u; v$ and $v; u$ give the same results.

One can prove that a sufficient condition for the equivalence of two terminating programs which execute the same operations is that dependent operations are executed in the same order in both programs.

According to this definition, the dependence relation is symmetric. If one of the programs above is the reference program (e.g. if it is the sequential program to be parallelized), one usually orients the dependence relation according to the execution order of the reference program. With this convention, a transformed program is equivalent to the reference program if its execution order is an extension of the dependence relation.

Bernstein's Conditions

Deciding if the execution order of two operations can be reversed can be arbitrary difficult. Bernstein [3] has devised a simple test which gives a sufficient condition for independence. This test depends only on the memory cells which are accessed by the operations to be tested.

Let $R(u)$ be the set of memory cells which are read by u. Similarly, let $W(u)$ be the set of memory cells which are written by u. Then u and v are

independent if the three sets: $R(u) \cap W(v), R(v) \cap W(u)$ and $W(u) \cap W(v)$ are all empty.

Note that in most languages, each operation writes at most one memory cell: $W(u)$ is a singleton. However, there are exceptions: multiple and parallel assignments, vector assignments among others.

Bernstein's conditions reflect a kind of worst-case reasoning. Consider:

```
y = e;  y = f;
```

The final value of `y` is given by the operation which executed last; in this case, this value is the value of `f`. If the operations are reversed, `y` will get the value of `e`, and *in the absence of any information* on the respective values of `e` and `f`, one must conclude that the two operations are dependent.

If the dependence relation is computed according to Bernstein's conditions, the above property can be extendend in two direction. Firstly, it applies now also to non terminating programs: one can show that the *history* of each variable (the succession of values it holds during execution) is the same for both programs. Secondly, it also holds if one of the program has real parallelism. The reason is that two operations which write to the same location are always in dependence (this is the third Bernstein condition) and hence can never be executed in parallel. Hence, two writes to the same memory cell never occur at the same time.

Scalars, Arrays, and beyond

Computing dependences for scalar variables is easy. The sets R and W above are finite, and computing intersections is trivial. The only real difficulty is caused by *aliasing*, when two distinct identifiers refer to the same memory cell. The detection of aliasing is a difficult problem. However, it is easy to detect cases in which there is no aliasing, e.g. among the local variables of a procedure, and to handle all other cases conservatively. Orienting each dependence is easy, at least within linear code (*basic blocks*).

The case of arrays is more problematic. Arrays usually occur in loops, and their subscripts usually differ from one iteration to the next in complex ways. This has lead to the invention of the *polyhedral model*[9, 6], in which one consider only regular DO loops, and subscripts which are affine functions of the surrounding loop counters and of constant parameters. A function is affine if it is the sum of a linear part and of a constant. One may construct a dependence relation among operations (i.e. loop iterations), in the following way. To name an iteration, one may use its *iteration vector*, whose coordinates are the surrounding loop counters, ordered from outside inward.

Iteration are ordered according to the lexicographic order of iteration vectors:

$$\vec{\imath} \prec \vec{\jmath} \equiv i_1 < j_1 \vee (i_1 = j_1 \wedge i_2 < j_2) \vee \ldots \quad (1)$$

where $\prec$ (read "before") is the execution order.

Consider two iterations $\vec{\imath}$ and $\vec{\jmath}$ of some loop nest, and two accesses to the same array, $A[f(\vec{\imath})]$ and $A[g(\vec{\jmath})]$, one of them at least being a write. f and g are subscript functions, which relate subscripts to the surrounding iteration vectors. To be in dependence, the two iterations must satisfy the following conditions:

- They must access the same array cell: $f(\vec{\imath}) = g(\vec{\jmath})$.
- They must be legal iterations, i.e. each loop counter must be within the corresponding loop bounds, which define the *iteration domains* of the operations.
- The direction of the dependence is given by the execution order $\vec{\imath} \prec \vec{\jmath}$.

For programs which fit in the polyhedral model, the subscript equations and the iteration domain constraints are are conjunctions of affine equalities or inequalities. According to (1), the ordering constraint can be split in several conjunctions. Hence, the set of iterations in dependence is the union of several disjoint *dependence polyhedra*. Each polyhedron is characterized by the number of equations at the begining of the execution order, the so-called *dependence depth*, which goes from 0 to the number of loops which enclose both array references. For some authors, the dependence depth (or level) starts at 1 and goes up to infinity. The notation $\vec{\imath}\,\delta_p\,\vec{\jmath}$ is commonly used to state that there is a dependence from iteration $\vec{\imath}$ to $\vec{\jmath}$ at depth p, i.e. that the first p coordinates of $\vec{\imath}$ and $\vec{\jmath}$ are equal.

For more general programs, one has to resort to approximations: a constraint is ommitted or approximated if not affine. This has the effect of enlarging the dependence polyhedra, and hence reducing the amount of parallelism. However, the correctness of the generated program is still guaranteed.

When a program uses pointers, the computation of dependences becomes much more difficult. The reason is that, depending on the source language, a pointer can point anywhere in memory, and that it is very difficult to decide whether two pointers points to the same memory cell. The best that can be done is to associate to each pointer (conservatively) a region in memory, and to decide a dependence when two regions intersect.

Classification

While in Berstein's conditions the two operations play the same role, the symmetry is broken as soon as the direction of the dependence is taken into account. One usually distinguish:

- Flow dependences (or true dependences, or Producer/Consumer dependences (PC), or Read after Write hazards (RAW)), in which the write to a memory cell occurs before the read.
- Output dependences (or PP dependences, or WAW hazards) in which both accesses are writes.
- Anti-dependences (or CP dependences, or RAW hazards) in which the read occurs before the write
- In some contexts (e.g. when discussing program locality), one may consider Consumer/Consumer dependences, which do not constrain parallelization.

If the objective is just to decide which operations can be executed in parallel, all three kinds are equivalent. Differences appear as soon as one consider modifying the source code for increased parallelism. It is easy to see that in a flow dependence, a value which is computed by the first operation is reused later by the second operation. Hence, a flow dependence corresponds to the naming of an intermediate result, is intrinsic to the code algorithm, and cannot be removed except by modifying it. In constrast, an output dependence simply indicates that the a memory cell is reused when the value it holds becomes useless. Such a dependence can be removed simply by using two distinct cells for the two values. Lastly, in a correct program, where all memory cells are defined before being used, removing output dependences has the side effect of removing anti-dependences.

It can even be shown that some of the flow dependences are spurious. Let us consider a memory cell and an operation v that reads it. There may be many writes to this cell. Intuitively, the only one on which v really depends is the last one u which is executed before v. The dependence from u to v is a *direct dependence*. If the source program is sufficiently regular, direct dependences can be computed using linear programming tools [5]. It is, however, difficult to find conservative approximations for more general programs.

Discussion

Dependence Tests

Early automatic parallelizers were concerned only with the existence or non existence of dependences. For instance, to decide that a loop is parallel, one has only to show that there are no dependences between its iterations, i.e. that all related dependence polyhedra are empty.

Since dependences occur between a pair of statements, it follows that the number of dependences increases more or less as the square of the size of the program. Hence, the scientists of the 1970's initiated a quest for fast but approximate *dependence tests*. One may for instance observe that some of the dependence constraints are linear equations whose unknowns, the loop counters, are integers. Such an equation can have solutions only if the gcd of the unknown's coefficients divides the constant term. This observation is the basis of the very fast *gcd test*.

The Banerjee test [2] is based on the observation that in many cases, when solving an equation $f(x) = 0$, one knows a lower bound a and an upper bound b for x, which come, most of the time, from loop bounds. Now, the equation has solutions only if $f(a)$ and $f(b)$ are of opposite signs. The approximation comes from the observation that the eventual solution is not necessarily integral, hence the idea of coupling the gcd and Banerjee test, for increased precision. This idea can be extended in the following way: the inequalities of the problem dictate that x belong to some polyhedron, and the condition for the existence of a solution is that the maximum of f over this polyhedron be positive, and the minimum be negative. Now the extrema of an affine function over a polyhedron are located at some of its vertices. Hence, one has only to test the sign of f at a finite set of points. This is especially efficient if the vertices are in evidence.

Many other tests were invented with the aim of handling systems of equations instead of a single equation. For instance, in the Lambda test [10] the Banerjee test is applied to well chosen linear combinations of equations. If the answer is negative, then the original system has no solution.

However, it was realized around 1990 that the question of the emptyness of the dependence polyhedron can be solved using classical linear programming algorithms. This approach was originally deemed too costly, but with improved algorithms and a thousandfold increase in processing power (Moore's law), the argument has lost its strength.

One possibility is to use the Fourier-Motzkin algorithm [12], in which the unknowns of the problem are eliminated by combining the inequalities which define the dependence polyhedron, until one obtains numerical inequalities,

which can be decided by inspection. Programming the Fourier-Motzkin algorithm is quite simple, but its complexity is doubly exponential, which is not critical since dependence testing problems are usually small. The standard Fourier-Motzkin algorithm finds rational solutions or proves than none exists. An extension, the Omega test [11], considers only integral solutions. Other extensions are the I test [8] and the Power test [13].

Another possibility is to use the Simplex algorithm, which is more complex, but which runs most often in time proportional to the third power of the number of inequalities, and hence scales better for large problems. The algorithm can also be extended to the integer case using Gomory cuts, and even solve parametric problems [4].

Dependence Approximations

Early parallelization algorithms (e.g. deciding if a loop has parallelism) did not need a precise knowledge of dependence polyhedra. Just testing them for emptyness was enough. As the sophistication of parallelization algorithms increased, more precise descriptions of dependences were needed. All such approximations can be described as supersets of dependence polyhedra. On the relations between dependence approximations and program transformations, see [14].

The simplest approximation consists in ignoring the fact that iteration vectors have integer components. When testing a transformed program for legality, this may generate false negatives. However, this occurs sufficiently seldom that it is usually ignored.

Dependence Depth

The dependence depth abstraction is a natural byproduct of the decomposition of the lexical order into disjoint polyhedra as in (1). One simply has to record at which depth a dependence occurs, instead of oring the test results for several depths. Knowledge of the dependence depth allows one to decide, in a loop nest, which loop must be kept sequential and which one can be executed in parallel. This information is necessary for the Allen and Kennedy algorithm, which uses loop splitting to find more parallel loops.

Dependence Distances

The dependence distance is the difference between the iteration vectors of two dependent operations. One can define a *distance polyhedron* as:

$$D = \{\vec{d} | \exists \vec{i}, \vec{j} : \vec{i}\, \delta_p\, \vec{j}, \vec{d} = \vec{j} - \vec{i}\}.$$

A projection algorithm is needed to eliminate the existential quantifiers: one may use the Omega test or parametric integer programming. Observe also that the dependence distance is always lexicopositive.

Computing the distance polyhedron is especially interesting when it reduces to a single constant vector. One says in that case that the dependence is uniform. Many parallelization algorithms are especially simple when the source program has only uniform dependences.

Dependence Direction Vectors

In the presence of non uniform dependences, one may futher abstract the distance polyhedron by considering only the signs of the components of the distance vectors. The components of a Dependence Direction Vector (DDV) are either integers (meaning that the component is constant) or one of the symbols $<, >, \geq, \leq$ (meaning that the component has the corresponding sign) or $*$, meaning that the component may have an arbitrary sign. DDVs are usually computed by adding the corresponding constraint to the definition of the dependence polyhedron and testing for feasibility. While simpler to compute than the distance polyhedron, DDVs give enough information to check the legality of some transformations, like loop permutation.

Dependence Cones

The dependence cone is simply the cone generated by the dependance distance vectors. The simplest way of computing the dependence cone is to compute first the vertices of the distance polyhedron, $d_1, \ldots, d_n$. The dependence cone is then:

$$C = \{\sum_{k=1}^{n} \lambda_k d_k | \lambda_k \geq 0\}.$$

Knowledge of the dependence cone is especially useful when tiling a loop nest.

Bibliographic Notes and Further Readings

There is a large litterature on pointer analysis which may be useful for program parallelization, albeit its orientation is more toward program verification and safety. A good starting point is [7] (110 references!).

For an attempt at parallelization of recursive programs, see [1].

References

[1] P. Amiranoff, A. Cohen, and P. Feautrier. Beyond iteration vectors: Instancewise relational abstract domains. In *Static Analysis Symposium (SAS'06)*, Seoul, Corea, August 2006.

[2] Utpal Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston / Dordrecht / London, 1988.

[3] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. on El. Computers*, EC-15, 1966.

[4] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[5] Paul Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, February 1991.

[6] Paul Feautrier. Automatic parallelization in the polytope model. In Guy-René Perrin and Alain Darte, editors, *The Data-Parallel Programming Model*, volume LNCS 1132, pages 79–103. Springer, 1996.

[7] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE'01*. ACM, 2001.

[8] X. Kong, D. Klappholz, and K. Psarris. An efficient data dependence analysis for parallelizing compilers. *IEEE Trans. on Parallel and Distributed Systems*, 1(3):342–359, July 1991.

[9] Christian Lengauer. Loop parallelization in the polytope model. In Eike Best, editor, *CONCUR'93*, volume LNCS 715, pages 398–416. Springer, 1993.

[10] Zhiyuan Li, Pen-Chung Yew, and Chuan-Qi Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Trans. on Parallel and Distributed Systems*, 1:26–34, January 1990.

[11] William Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, 1991.

[12] Rémi Triolet, François Irigoin, and Paul Feautrier. Automatic parallelization of FORTRAN programs in the presence of procedure calls. In Bernard Robinet and R. Wilhelm, editors, *ESOP 1986, LNCS 213*. Springer-Verlag, 1986.

[13] Michael Wolfe and Chau-Wen Tseng. The power test for data dependence. *IEEE Transactions on Parallel and Distributed Systems*, September 1992.

[14] Yi-Qing Yang, Corinne Ancourt, and François Irigoin. Minimal data dependence abstractions for loop transformations. In *LCPC '94: Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 201–216, London, UK, 1995. Springer-Verlag.