STUDIA UNIV. BABEŞ-BOLYAI, INFORMATICA, Volume II, Number 1, 1997

# DATA DEPENDENCE TESTING FOR AUTOMATIC PARALLELIZATION

GRIGOR MOLDOVAN ALEXANDRU VANCEA
MONICA VANCEA

**Abstract.** In this paper we prove that automatic parallelization is the moment's most suited approach for large scale integration of parallel program versions resulted from the original sequential versions. This parallelism is of SIMD type, is loop structure oriented and offers the most substantial speedup relative to the sequential versions. Having as the main final purpose the construction of a restructuring compiler, we focus here on the *data dependence analysis* phase of such a compiler. The paper makes a complete and up to date overview together with a critical analysis of the data dependence testing techniques and claims that symbolic analysis is also needed for better results in a more general framework. A substantial list of references is given which can constitute an excellent guidance for the reader interested in the topics.

## 1. Introduction

The trend towards parallel computing has become no more just an option, but a necessity. High performance computing has become vital for scientists. Much technical progress has been made in developing large scale parallel architectures composed of many powerful processors.

The software transition to a novell way of programming and thinking has proved to be not so easy. Research upon the ways in which computer scientists can make this mandatory change emerged a long time ago [Allen69, Karp67, Kuck72, Love77, Lamp74, Mura71]. In spite of its envisioned effectiveness, parallel programming could not impose itself during the years as the general and unanimously accepted method for designing, implementing and/or executing algorithms.

The problems posed by a totally new software strategy proved to be extremely difficult to solve both at the level of the theoretical foundations and especially at the level of programmers' mentalities. Our inherent sequential life and way of thinking together with the already well learned and understood traditional sequential programming made this transition to parallel programming very hard to accomplish.

Logically, there are two practical ways in which we could approach the transition from sequential programming to parallel programming.

---

Received by the editors: January 14, 1998.
*1991 Mathematics Subject Classification.* 68N15.
*1991 CR Categories and Descriptors.* D.1.3 [**Programming Techniques**]: Concurrent Programming - parallel programming; D.2.8 [**Software Engineering**]: Metrics - *performance measures*; D.3.4 [**Programming Languages**]: Processors - *optimization, compilers*

One way is to build from scratch everything again, having in mind the parallel execution model. This way would have the advantage of being the most effective, being oriented towards efficiently building parallel applications. New languages would have to be designed (actually there are already some [Van94]) which had to offer the whole methodology and constructs needed for such an action. Again, we have here two approaches if we want to build imperative language facilities for parallel programming.

The first approach augments an existing language with a set of directives. The programmer then becomes responsible for inserting these directives into strategic parts of the program, instructing thus the compiler on how to best perform the parallelization process.

The second approach incorporates parallel constructs directly into the language definition. Examples of this kind are seen in High Performance Fortran [HPF93] and IBM Parallel Fortran [Sark91], where language constructs such as PARALLEL DO and PARALLEL CASE are introduced.

The other practical way is to pass the whole responsability of paralellization onto an automatic tool which extracts parallelism directly from the sequential program. This approach is extremely important for the actual moment because there are many code libraries in use which could thus maybe execute in parallel. Furthermore, the issues involved in parallelizing a sequential program subsumes many of the problems faced by compiler developers of other languages. The experience gained in developing a parallelizing compiler can therefore be applied to compilers for other languages with more explicit parallel constructs. Also, developing a parallelizing compiler for an imperative language allows a programmer to develop code for a massively parallel programming (MPP) architecture in a familiar language like C or FORTRAN. Such programs would also result in more portable code if effective compilers can be developed for the different classes of MPP architecture.

These are in our opinion the main reasons for which we argue that (even if we think that in the near future the parallel applications will be build from scratch and with the help of new and specialized languages) the actual moment must accept as the main facility for building parallel programs the developing of parallelizing compilers.

## 2. Data dependence

A parallelizing compiler has to detect the possible instructions to execute in parallel. For this, it has to make a *dependence analysis* of the sequential programs to determine the data flow and their interactions.

One of the main actions of a parallelizing compiler is the *restructuring* of the sequential program (that's why they are sometimes called *restructuring compilers*). This restructuring aimes to maintaining where necessary (that means where the data dependencies impose it) the order in which the instructions are executing. That's why data dependence represents the theoretical framework on which the restructuring methodology is based.

It is important to note that this concept does not appear related only with parallel programming. Classic compiler optimizations make also use of it to accomplish their tasks [Aho86].

There are many good introductions to these topics and we refer the reader to [Bane88a] and [Bane88b], because the aim of this paper is not to be a tutorial on data

dependence, but to overview the existing test methodologies and to claim that symbolic analysis is also needed for practical application.

The following sections review the most important data dependence analysis methods proposed in the literature and makes a critical analysis of their strengths and weaknesses.

## 3. Data dependence tests and their effectiveness

The main aim for an automatic parallelizer are loops and inside them the analysis is mostly concerned with array elements. We can classify data dependence tests based on the appeareance of the array subscripts.

**Definition**. An array variable has *coupled* subscripts if there exists some index variable which appears in two or more subscript expression fields.

For example, the array variable *A[x+y,x]* is said to have coupled subscripts, but *A[y,x]* no (so we term them *uncoupled subscripts*).

### 3.1. Uncoupled index subscript analysis methods

We present further a class of data flow analysis strategies which attempt to analyse the array subscript expressions directly. For example, in the code fragment

**for** *i:=1* **to** *n* **step** *2* **do**

*d1* *a[2∗i] := ...*

*d2* *... := a[2∗i + 1]*

**end for**

many schemes will attempt to solve the dependence equation

(1) $$2i_1 = 2i_2 + 1 \quad \text{or} \quad 2i_1 - 2i_2 = 1$$

with $1 \le i_1, i_2 \le n$.

If a solution exist for equation (1), we deduce a dependence when a value defined in variable *a*, in statement *d1*, is referenced in statement *d2*. This information is important because statements in a loop kernel which are deduced to be data independent can then be executed in parallel.

#### 3.1.1. The GCD-Banerjee test

The expression in equation (1) is known as a *linear diophantine equation* in two variables. From the GCD theorem in Number Theory, equation (1) has an integer solution if and only if the greatest common divisor (GCD) of the coefficients in the left hand side (LHS) exactly divides the constant term in the right hand side (RHS). Thus, for dependence equation (1), the GCD of the coefficients on the LHS, GCD(2,2) = 2, does not exactly divide the constant term on the RHS. Hence we can deduce no data dependence. This test is known as the GCD test and it was first introduced by Utpal Banerjee [Bane88a].

The major disadvantage of the GCD test is that it only *predicts* the existence of an *unconstrained integer solution*. Where an integer solution exists which does not satisfy the bound inequalities constraint, i.e. $1 \le i_1, i_2 \le n$, the GCD test will predict a data dependence when none exist.

Another test which takes the bound constraints into consideration uses the intermediate value theorem. A *real* solution exists if the RHS is found to lie within the minimum and maximum values which the LHS can take. Therefore for dependence equation (1), $\min(2i_1^{\min} - 2i_2^{\max}) = 2 - 2n$ and $\max(2i_1^{\max} - 2i_2^{\min}) = 2n - 2$. If $2 - 2n \leq l \leq 2n - 2$, equation (1) has a *real* solution within the bound constraints, otherwise a real solution does not exist. The bounds test is commonly referred to as *the Banerjee test* as it was also first introduced by Utpal Banerjee [Bane88a]. Note that the Banerjee test only decides if a constrained real solution exists.

An array subscript analysis scheme can be generalized for a pair of def-ref $m$-dimensional array variables illustrated by the code fragment shown below:

**for** $x_1 := L_1$ **to** $U_1$ **do**

...

**for** $x_n := L_n$ **to** $U_n$ **do**

d1: $A[f_1^{gen}(X), \ldots, f_m^{gen}(X)] := \ldots$

d2: $\ldots := A[f_1^{use}(X), \ldots, f_m^{use}(X)]$

**end for**

...

**end for**

where $X$ is the index set $\{x_1, \ldots, x_n\}$ and $f_i^{gen}(X)$ and $f_i^{use}(X)$ are linear functionals in terms of $X$. A data dependence exists between *d1* and *d2* if the system of diophantine equations,

$$f_1^{gen}(X) - f_1^{use}(X') \equiv a_{1,1}x_1 - b_{1,1}x'_1 + \ldots + a_{1,n}x_n - b_{1,n}x'_n = c_1$$

$$\ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots$$

$$f_m^{gen}(X) - f_m^{use}(X') \equiv a_{m,1}x_1 - b_{m,1}x'_1 + \ldots + a_{m,n}x_n - b_{m,n}x'_n = c_m$$

or described more concisely

$$\begin{aligned} S_1 &\equiv a_{1,1}v_1 + \ldots + a_{1,2n}v_{2n} = c_1 \\ &\ldots \quad \ldots \quad \ldots \\ S_m &\equiv a_{m,1}v_1 + \ldots + a_{m,2n}v_{2n} = c_m \end{aligned} \tag{2}$$

has an integer solution subject to the following subscript constraint inequalities

$$\begin{aligned} L_1 &\leq v_1, v_2 \leq U_1 \\ &\ldots \quad \ldots \quad \ldots \\ L_n &\leq v_{2n-1}, v_{2n} \leq U_n \end{aligned} \tag{3}$$

This problem is similar to integer programming where given the subscript equalities represented by system (2) and the inequalities represented by system (3), an integer solution represented by the vector $\bar{v} = (v_1, v_2, \ldots, v_{2n})$ is required which will maximize some cost function $cost(\bar{v})$. Our data flow analysis problem is simpler in that we only determine if an integer solution exists, or in the case of a geometric interpretation, we aim to determine if the hyperplanes described by the subscript equalities intersect at $S$, within a region $V$ bounded by the subscript constraint inequalities, with $S$ containing some integer point.

### 3.1.2. The I-test

The GCD-Banerjee test does not determine exactly if a def-ref array variable pair is data independent. The reason for this is that it does not distinguish the case where the *real* solution satisfies the bounds constraints (3) but the integer solution does not; both tests will determine a dependence. The I-test proposed by Kong, Klappholz and Psarris [Kong91], integrates the two tests and determines exactly whether an integer solution exists for each equation in the dependence system (2) restricted by the bounds constraints (3).

The I-test poses the subscript equality equations as *interval equations* of the form

$$\begin{aligned} & a_{1,1}x_1 - b_{1,1}x'_1 + \ldots + a_{1,n}x_n - b_{1,n}x'_n = [L_1, U_1] \\ & \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \\ & a_{m,1}x_1 - b_{m,1}x'_1 + \ldots + a_{m,n}x_n - b_{m,n}x'_n = [L_m, U_m] \end{aligned} \tag{4}$$

where for the $i^{th}$ subscript equality, the interval equation

$$a_{i,1}v_1 + \ldots + a_{i,2n}v_{2n} = [L_i, U_i] \tag{5}$$

is checked for integer-solvability. If $g$ is the GCD of the coefficients $\{a_{i,1}, \ldots, a_{i,2n}\}$, an integer solution exists if $L_i \le g\lceil L_i/g \rceil \le U_i$. Each interval equation is then successively transformed through a series of variable elimination steps for which the step to eliminate $v_{2n}$ is shown below:

$$a_{i,1}v_1 + \ldots + a_{i,2n-1}v_{2n-1} = [L_i - a^+_{i,2n}U_{2n} + a^-_{2n}L_{2n},\ U_i - a^+_{2n}L_{2n} + a^-_{2n}U_{2n}]$$

assuming $|a_{i,2n}| \le U_{2n} - L_{2n} + 1$ and

$a^+ = a$ if $a \ge 0$, $0$ otherwise

$a^- = a$ if $a \le 0$, $0$ otherwise

With each new interval equation generated the bounds test is performed. If the procedure encounters an equation which is inconsistent, the I-test concludes that the array variables concerned are independent. The I-test is an exact test for def-ref array variable pairs possessing uncoupled subscripts. Array variables with coupled subscripts require more sophisticated techniques to disambiguate the array variable access patterns.

## 3.2. Coupled index subscripts analysis methods

We have until now avoided the complications involved in the analysis of coupled subscript expressions. For all the tests described so far multi-dimensional array variables are tested *subscript-by-subscript*. That is, the systems (2) and (3) are formulated and tested one array dimension at a time.

As we mentioned before, an array variable has coupled subscripts if there exists some index variable $x_i \in X$ which appears in two or more subscript expression fields (the array variable *A[x+y,x]* is said to have *coupled subscripts*). For a def-ref array variable pair with coupled subscripts a subscript-by-subscript test will yield an over conservative estimate. For example, in the code fragment below a subscript-by-subscript application of the GCD-Banerjee test will determine a data dependence when none exists.

```
      for i := 1 to n do
d1:     A[i+1,i+2] := A[i,i] + 3;
      end for
```

### 3.2.1. The Delta test

The Delta test proposed by Goff, Kennedy and Tseng [Goff91] accounts for coupled subscripts by using a *constraint propagating technique*. Their scheme first involves partitioning the subscript equality expressions into coupled and uncoupled groups, i.e. checking for *separability*. Then they attempt to solve the subscript equalities in an order where an uncoupled subscript equality is favoured over a coupled one. A data independence test (i.e. the GCD-Banerjee test as suggested by the authors) is then performed and constraints are generated in the form of either a dependence hyperplane, a dependence distance or a dependence point.

The generated constraints are propagated into the next subscript test where independence is concluded when either the application of the data independence test determines so, or the set of constraints do not intersect. A simple example of the application of Delta test can be seen in conjunction with the code fragment above. The first subscript dimension is checked where a data independence cannot be determined. A dependence distance constraint, $c_1$: $i_1 - j_1 = 1$, is generated. The second subscript dimension is then checked where again a data dependence cannot be determined. A second dependence distance constraint, $c_2$: $i_1 - j_1 = 2$, is generated. Since $c_1 \cap c_2 = \emptyset$, we conclude that the def-ref array variable pair is data independent.

### 3.2.2. The λ test

The λ-test proposed by Li, Yew and Zhu [Li90] is an efficient data independence test for two multi-dimensional array variables. It is one of the few tests that attempts to solve system (2) simultaneously. The test is shown to be especially efficient for two-dimensional arrays. Studies of scientific and engineering programs [Shen90] have shown that two-dimensional arrays are the most common type of array structures used.

The central idea of the $\lambda$-test, for two-dimensional arrays is the determination of a set of $\psi$ and $\phi$ lines, on a $(\lambda_1, \lambda_2)$ plane, which define the dependence system. The $\psi$ and $\phi$ lines are of the form $a\lambda_1 + b\lambda_2 = 0$ where $a$ and $b$ are integer constants. The $\lambda$-test for data independence involves forming linear combinations of the $\psi$ and $\phi$ lines,

$$f_{\lambda_1,\lambda_2} = \lambda_1 S_1 + \lambda_2 S_2$$

called a $\lambda$-plane, and showing that each $\lambda$-plane does not intersect the subspace $V$ defined by the constraining subscript inequalities, as defined in system (3). To check if the $\lambda$-plane intersects $V$ a simple bounds computation is performed. The procedure first determines $\min(f_{\lambda_1,\lambda_2})$ and $\max(f_{\lambda_1,\lambda_2})$ and if $\min(f_{\lambda_1,\lambda_2}) \le 0 \le \max(f_{\lambda_1,\lambda_2})$ we than conclude that $f_{\lambda_1,\lambda_2}$ intersects $V$.

### 3.2.3. The Power test

The $\lambda$-test, like the Banerjee bounds test, determines if there is a constrained *real* solution to the equalities in system (2). A true dependence will only occur if there is a constrained integer solution to the system. The Power test, proposed by Wolfe and Tseng [Wolfe92], integrates the generalized GCD test for unconstrained integer solutions with the Fourier-Motzkin variable elimination technique for constrained real solutions. The test will determine if there is a constrained integer solution if one exists.

The test can also handle triangular, trapezoidal and other convex loop bounds. There is a slight inaccuracy in the test, however, which will be explained further.

The generalized GCD test was proposed by Banerjee to determine if an unconstrained integer solution exist for the system (2). We start with this system which can be concisely described by the matrix-vector product pair

(6) $$\bar{v}\ A = \bar{c}$$

where $\bar{v} = (v_1, \ldots, v_{2n})$,

$$A = \begin{pmatrix} a_{1,1} & \ldots & a_{1,2n} \\ \ldots & \ldots & \ldots \\ a_{m,1} & \ldots & a_{m,2n} \end{pmatrix}$$

and $\bar{c} = (c_1, \ldots, c_{2n})^T$.

In the generalized GCD test, (6) can be factored (by Gaussian elimination for example) into an unimodular matrix $U$ and an echelon matrix $D$ such that $U \cdot A = D$. The system (6) has an integer solution if and only if there exist an integer vector $\bar{t}$ such that $\bar{t}\ D = \bar{c}$. Since $D$ is an echelon matrix, $\bar{t}$ can be found by back substitution. If $\bar{t}$ cannot be determined the system has no solution and the def-ref array pair are independent. If $\bar{t}$ has an integer solution, then

(7) $$\bar{v} = \bar{t}\ U$$

is the solution to the system (6).

The Power test continues by determining if the integer solution for expression (7) satisfies the constraint system (3). It uses the Fourier-Motzkin variable elimination method to determine this.

For example, consider the system of three constraint inequalities

(8) $$t_1 + t_2 > 1,\ t_1 - t_2 < 2,\ t_2 < -2$$

Transforming system (8), we get

(9) $$t_1 + t_2 > 1 \Rightarrow t_1 > 1 - t_2$$

(10) $$t_1 - t_2 < 1 \Rightarrow t_1 < 2 + t_2$$

The Fourier-Motzkin eliminates $t_1$ by projecting (9) > (10)

$$2 + t_2 > 1 - t_2 \Rightarrow t_2 > -1/2$$

and combining the new inequality with the original system (8), we obtain

$$-1/2 < t_2 < -2$$

which is inconsistent. We therefore conclude no feasible solution to the system of inequalities.

Now the solution to expression (7) for $v_1, \ldots, v_{2n}$ is expressed in terms of the free variables $t_{m+1}, \ldots, t_{2n}$. The Power Test substitutes the relevant terms in system (3) with their free variable solution from expression (7) and solves the new constraint system using the Fourier-Motzkin variable elimination method. If an inconsistent pair of bounds is encountered the test deduces no data dependencies between the def-ref array variable pair.

There is an inaccuracy in the Power test, for an inequality

$$h_1 t_1 + h_2 t_2 + \ldots + h_k t_k \geq U$$

for which we wish to eliminate variable $t_k$,

$$h_k t_k \geq U - h_1 t_1 - h_2 t_2 - \ldots - h_{k-1} t_{k-1} \Rightarrow t_k \geq \frac{U - h_1 t_1 - h_2 t_2 - \ldots - h_{k-1} t_{k-1}}{h_k}$$

The Power test takes the *ceiling* of the division on the RHS, i.e.

$$t_k \geq \left\lceil \frac{U - h_1 t_1 - h_2 t_2 - \ldots - h_{k-1} t_{k-1}}{h_k} \right\rceil$$

The *ceiling* operator in this case is the source of imprecision because it enlarges the solution space and may consequently introduce integer solutions where none exist.

### 3.2.4. General integer programming methods

All the data flow analysis methods which have been described in this section are inexact, in that a dependence between a def-ref array variable pair may be reported when none exists. *An exact answer can be derived through integer programming.* Whereas integer programming requires a vector $\bar{v}$ to be found which minimizes or maximizes some cost function, a data independence test need only determine if there is a feasible solution within a convex set in $\mathbb{R}^{2n}$ and if that solution space contains at least one integer point. Wallace [Wallace88] Lu and Chen [Lu90] and Pugh [Pugh92] have adopted this approach. Note that there is already a very large body of work on integer programming and that the method is generally agreed to be NP-complete.

Wallace proposes the Constraint-Matrix method which uses the simplex method in linear programming modified to solve integer programming problems. The number of pivoting steps in his method is bounded by a limit to prevent a condition know as *cycling*, where the simplex method is known not to converge onto a solution. Lu and Chen describe a new integer programming algorithm but the cost of using their procedure is difficult to assess as they do not provide a metric. Finally, Pugh proposes the Omega test which uses a modified version of the Fourier-Motzkin variable elimination algorithm. Through careful implementation, Pugh claims the Omega test to be practical for commonly encountered def-ref array variable pairs. The Omega test has, however, a *worst case exponential time complexity*, although Pugh claims that the algorithm is actually bounded within polynomial time for common cases.

## 3.3. Concluding remarks

Data flow analysis for multi-dimensional array variables requires complicated techniques to disambiguate the access patterns defined by the array subscript expressions. Table 1 summarizes the characteristics of the data flow analysis methods described in this section.

Almost all the proposed analysis methods have avoided the complexity of an exact method by deeming it sufficient to provide an approximate solution. The exceptions have been methods which aim to solve the data flow analysis problem exactly as programs that employ integer programming.

| Method | Constrained integer | Integer | Constrained real | coupled subscript | Exact |
|---|---|---|---|---|---|

| | | | | | |
|---|---|---|---|---|---|
| GCD test | No | Yes | No | No | No |
| Banerjee test | No | No | Yes | No | No |
| I-test | Yes | Yes | Yes | No | No |
| λ-test | No | No | Yes | Yes | No |
| Delta test | Yes | Yes | Yes | Yes | No |
| Power test | Yes | Yes | Yes | Yes | No |
| Integer programming | Yes | Yes | Yes | Yes | Yes |

**Table 1: Characteristics of data flow analysis algorithms**

However, these integer programming techniques have worst case exponential time complexity which have discouraged many parallelizing tool developers from adopting them. Pugh claims the Omega test [Pugh92], which is an integer programming algorithm, to have a typical performance which is "acceptable".

He makes this claim based on experiments which employ the Omega test on a collection of commonly used scientific and engineering loop kernels. He does not, however, prove his assertion for general case.

## 4. Symbolic data dependence analysis

Symbolic analysis is needed for solving more systems exactly than previous approaches. That is because symbolic terms play a significant role in the analysis of programs. For advocating this, let's take the following example:

```
read(n);
if n > 10 then
    for i:=1 to 10 do
        a[i+n] := a[i]+5;
    end for
end if
```

If we annotate the two references with the fact that n > 10 this allows us to prove that they are not truly dependent. The more general approach can be very expensive at compile time. It is necessary to track the relationship of all scalar variables in a program. In addition, the data dependence system being solved becomes more complex, containing more variables and more constraints.

A variety of methods for doing symbolic analysis have been proposed in the literature. Reif [Reif78] uses the control flow graph to construct a mapping from program expressions to symbolic expressions for their value holding over all executions of the program. This *cover* is weaker than others that have been done, but the algorithm is much cheaper. Its cost is $O(leng + a\ \alpha(a))$ bit vector operations, where *leng* is the length of the program, $a$ is the number of edges in the control flow graph, and $\alpha$ is Tarjan's function, which is almost linear.

Reif and Lewis [Reif77] use a structure called the *global value graph* to compactly represent symbolic values and the flow of those values through the program. Their algorithm for symbolic evaluation discovers simple constants and minimal fixed point covers and is almost linear in the size of the global value graph. Like [Reif78], this technique deals only with variables that have the same symbolic value along all paths in the program's control flow graph. The global value graph is of size $O((\sigma a + leng))$,

where σ is the number of variables in the program, *a* is the number of edges in flow graph, and *leng* is the length of the program.

Alpern, Wegman, and Zadeck [Alpern88] try to determine equality relations between program variables in the presence of control flow. Their approach uses value numbering and builds a *value graph* that represents the symbolic execution of the program. Determining which nodes in the value graph are congruent is an *O(E logE)* problem, where *E* is the number of edges in the value graph. Each assignment in the static single-assignment form of the program is a node in the value graph, which has two types of nodes: *executable function nodes*, which represent normal assignments, and *φ function nodes*, which represent join points in the control flow graph.

Cousot and Halbwachs [Cousot78] determine inequality as well as equality relations, using abstract interpretation. Their approach converts linear constraints into a convex polyhedron and tries to deduce assertions about the relationships between program variables from the semantics of the program. This method is intended to be applied to tasks such as array bound checking; precision will be lost when the numbers are reals instead of integers. The cost of their analysis is almost linear in the length of the program but exponential in the number of variables involved in the analysis.

Haghighat and Polychronopoulos [Hagh90] propose a framework for doing dependence analysis in the presence of symbolic terms, using abstract interpretation based on a lattice model. Essentially, they handle the simple symbolic cases that PFC takes care of: symbolic loop bounds, symbolic additive terms, and symbolic multipliers [Allen83]. The lattice is used to do constraint propagation and inequalities are solved using the convex hull method of Cousot and Halbwachs.

Lichnewsky and Thomasset [Lich88] evaluate constraints symbolically when it cannot be done numerically, using approximations to reduce the computational cost. A general decision procedure is employed only after the Banerjee-GCD test [Bane79, Allen87, Bane88a] has failed. The decision procedure uses the SUP-INF method for proving Presburger formulas, as described by Shostak [Shos77]. The basic approach is to transform the integer problem into the real domain, solve it, and translate the results back into the integer domain. The complexity of this algorithm is $O(2^N)$, where N is the length of the formula.

Havlak [Havlak94] uses *array section analysis*, a special SSA form, and global value numbering as integral components of his analysis. His approach combines value numbering with graph rewriting and is particularly adept at representing and matching patterns of operators, unlike [Hagh90], whose strength lies in rewriting expressions and solving recurrences.

Blume and Eigenmann [Blume94] describe a new dependence test, known as *the Range Test*, which proves independence by determining whether symbolic inequalities hold for a permutation of the loop nest. The Range Test is slower than Banerjee's Inequalities but faster on average than the Omega Test, although the Omega Test can break cross-iteration dependences in some cases where the Range Test cannot.

Some research has avoided the more traditional memory-based notion of data dependence in favor of a value-based approach. Instead of testing every pair of array references which may access the same memory location, a value-based approach eliminates pairs separated by an intervening write, thus eliminating spurious dependences from consideration. Collard, Barthou, and Feautrier [Collard95] present a

method which gives exact information in some cases and which can handle nonlinear terms in loop bounds and IF conditions but not in array subscripts. Maslov [Maslov94] exactly computes value-based dependence relations for program fragments where all subscripts, loop bounds, and IF conditions are affine. The algorithm, which is lazy, also handles some nonlinear terms. Pugh and Wonnacott [Pugh94] handle more nonlinear constraints than Maslov's algorithm and their work also has the ability to relate terms in the dependence analyzer back to program expressions which can be more readily understood by the programmer.

Finally, Shen, Li, and Yew [Shen90] present the results of an empirical study on array subscripts and data dependences. This study concludes that many of the subscripts contained symbolic terms with unknown values. An average of 33.97% of all subscripts were nonlinear, and of these, 85% to 100% of the nonlinear or partially linear subscripts were caused by an unknown variable. User assertions pertaining to the value of some symbolic variables were employed in an effort to reduce the number of nonlinear subscripts for six programs. Using user assertions, only 27.38% of one-dimensional array references and 14.7% of two-dimensional array references were nonlinear, as opposed to 47.44% of one-dimensional array references and 44.91% of two-dimensional array references when assertions were not used.

## 5. Run-time parallelization

Polychronopoulos proposes a technique called *run-time dependence checking* (RDC) [Poly86, Poly87, Poly88] that uses program information available at compile-time to generate code that resolves dependences at run-time. The goal of RDC is to determine at run-time whether a particular iteration of a loop depends on one or more previous iterations, and synchronize the loop iterations involved in dependences. A *dependence source vector* is created for each true dependence in the loop. Non-zero elements in the dependence source vector indicate the iterations of the loop that may be involved in a dependence. For each potential dependence source, a statement is inserted at the beginning of the target loop that causes the program to wait on the *synchronization vector* until the statements corresponding to the dependence sources have executed. Iterations which cannot be involved in a dependence are executed in parallel without any constraints. He gives no time complexity for the algorithm used to construct the dependence source vectors, but estimates that the storage requirement for the dependence source vectors is O(kN), where k is the number of dependences and N is the number of loop iterations.

Saltz et al. [Saltz91] describe a compiler that transforms a loop into two separate code segments, an *inspector* and an *executor*. The inspector reorders loop iterations into concurrent wavefronts, then the executor carries out the scheduled work. This method applies only to loop nests in which inter-iteration dependences do not depend on the results of computations carried out in the loop nest. If the run-time parallelization is being done for a distributed memory architecture, then the inspector phase will also determine the location of distributed array elements and calculate which communication calls will be needed to retrieve the data. Information about communication patterns is saved between executions in an effort to amortize the cost of the inspector.

Rauchwerger and Padua [Rauch94] present their Privatizing DOALL test, which identifies fully parallel loops at run-time and dynamically privatizes scalars and arrays.

The test can be applied to any loop, regardless of control flow. The test can either be performed before the loop is executed, determining whether the loop can be executed as a DOALL, or it can be done while the loop is being speculatively executed as a DOALL. They conclude that the test should be applied in almost all cases because the cost of encountering a less-than-fully-parallel loop is minimal compared to the expected speedup.

Leung and Zahorjan [Leung93] present an inspector-executor method with two possible approaches. The first, which targets inspector efficiency, achieves nearly linear speedup relative to a sequential execution of the inspector, but may produce a suboptimal schedule for the executor. The second approach emphasizes executor efficiency and is guaranteed to produce the best schedule, at the cost of a potentially slower inspector loop.

## 6. Run time anomaly detection

*Access anomalies*, also referred to as *data races*, occur when one thread of a parallel program modifies a shared variable concurrently being accessed by another thread. Dinning and Schonberg [Dinn90] present a debugging algorithm known as *task recycling*, for detecting access anomalies at run-time. The expected space complexity is $O(V)$, but the worst-case bound is $O(TV)$, where $T$ is the maximum number of threads in the program and $V$ is the number of variables monitored. This *on-the-fly* approach detects anomalies by monitoring program execution. When a variable is accessed during execution, a check is made to see if the current access conflicts with a previous access. If a conflict does occur, then an error is reported. Their empirical estimate of the cost of monitoring is a 3-fold to 6-fold slowdown if every shared variable reference is monitored. This cost can be reduced by using static analysis or user assertions to limit the number of variables that are monitored.

Hood, Kennedy, and Mellor-Crummey [Hood90] also use an on-the-fly detection method, but their algorithm is more efficient than the one presented in [Dinn90]. The overhead attributed to monitoring program execution is estimated to be less than 40%, and a single instrumented execution will either identify sources of schedule-dependent behavior or verify that all executions of the program on the same set of data compute the same result. Their approach uses dependence analysis to reduce the amount of run-time checking, and they also reduce the cost of detection by using a method that works for programs with no nested parallel constructs using only the structured synchronization primitives in a disciplined way.

## 7. Conclusions and future work

We sustained in this paper the idea that automatic parallelization is the most suited approach at the moment for the large scale integration of parallel program versions of sequential programs. This parallelism is of SIMD type, concentrated on the loop structures of sequential programs and offers the most substantial speedup relative to the sequential versions. Our main final objective is the construction of a restructuring and optimizing compiler. Here we focused on the data dependence analysis phase of such a compiler. The paper made a complete and up to date overview together with a critical analysis of the data dependence testing techniques and claimed that symbolic analysis is

also needed for better results in a more general framework. The substantial list of references given can be of a great help to the reader interested in these topics.

Our future efforts will be devoted to the establishment of a practical approach of efficient data dependence testing (known to be NP-complete in the general case), not focusing on only one particular methodology, but cascading existing techniques in particular manners based upon the main features of the class of problems taken into consideration.

## REFERENCES

[Aho86] **A.V.Aho, R.Sethi and J.D.Ullman** - ***Compilers: Principles, Techniques and Tools***, Addison-Wesley, Reading, Massachussetts, 1986.

[Allen69] **F.E.Allen** - ***Program optimization***, in *Annual Review in Automatic Programming 5, International Tracts in Computer Science and Technology and their Applications*, vol.13, Pergamon Press, Oxford, England, pp.239-307, 1969.

[Allen83] **J. R. Allen** - ***Dependence Analysis for Subscripted Variables and Its Application to Program Transformations***, Ph.D. thesis, Department of Computer Science, Rice University, Houston, April 1983.

[Allen87] **J. R. Allen and K. Kennedy** - ***Automatic translation of Fortran programs to vector form***, in *ACM Transactions on Programming Languages and Systems*, 9(4), pp.491--542, October 1987.

[Alpern88] **B. Alpern, M.N. Wegman and F. K. Zadeck** - ***Detecting equality of variables in programs***, in *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pp.1-11, San Diego, California, January 1988.

[Bane79] **Utpal Banerjee** - ***Speedup of ordinary programs***, PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1979, Report No. 79-989.

[Bane88a] **Utpal Banerjee** - ***Dependence Analysis for Supercomputing***, Kluwer Academic Publishers, Norwell, Massachussets, 1988.

[Bane88b] **Utpal Banerjee** - ***An Introduction to a Formal Theory of Dependence Analysis***, *The Journal of Supercomputing* 2 (1988), pp.133-149.

[Bane97] **Utpal Banerjee** - ***Dependence Analysis***, Kluwer Academic, 1997.

[Blume94] **William Blume and Rudolf Eigenmann** - ***The Range test: A dependence test for symbolic, non-linear expressions***, in *Supercomputing '94*, IEEE Computer Society, 1994.

[Collard95] **J.F. Collard, D.Barthou, and P.Feautrier** - ***Fuzzy array dataflow analysis***, in Jeanne Ferrante and David Padua, editors, *Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP, volume 30(8), pp. 92-101, August 1995.

[Cousot78] **Patrick Cousot and Nicholas Halbwachs** - ***Automatic discovery of linear restraints among variables of a program***, in *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pp.84-96, Tucson, Arizona, January 1978.

[Dinn90] **A. Dinning and E. Schonberg** - ***An empirical comparison of monitoring algorithms for access anomaly detection***, in *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 1-10, Seattle, WA, March 1990.

[Dowl90] **M.L.Dowling** - ***Optimum code parallelization using unimodular transformations***, in *Parallel Computing*, 16, 1990, pp.155-171.

[Goff91] **G.Goff, K.Kennedy and C.W.Tseng** - ***Practical Dependence Testing***, in *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 26-28, 1991, pp.15-29.

[Hagh90] **M. Haghighat and C. Polychronopoulos** - ***Symbolic dependence analysis for high-performance parallelizing compilers***, in *Advances in Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990. The MIT Press.

[Havlak94] **Paul Havlak** - ***Interprocedural Symbolic Analysis***, PhD thesis, Department of Computer Science, Rice University, May 1994. Also available as CRPC-TR94451 from the Center for Research on Parallel Computation and CS-TR94-228 from the Rice Department of Computer Science.

[Hood90] **R. Hood, K. Kennedy and J. Mellor-Crummey** - ***Parallel program debugging with on-the-fly anomaly detection***, in *Proceedings of Supercomputing '90*, New York, NY, November 1990.

[HPF93] ***High Performance Fortran Forum***, in *High Performance Fortran Journal of Development*, CRPC-TR93300, Center for Research on Parallel Computation, Rice University, Houston, May 1993.

[Irig88] **F.Irigoin, R.Triolet** - ***Supernode partitioning***, in *Conference Record of the 15th ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988, ACM Press, New York, pp.319-329.

[Karp67] **R.M.Karp, R.E.Miller and S.Winograd** – ***The organization of computations for uniform recurrence equations***, in *Journal of the ACM*, 14(3), pp.563-590, July 1967.

[Kenn93] **K.Kennedy, K.McKinley, C.W.Tseng** - ***Analysis and transformation in an interactive parallel programming tool***, in *Concurrency Practice and Experience*, 5, 7 October 1993, pp.575-602.

[Kong91] **X.Kong, D.Klappholz and K.Psarris** - ***The I Test: An Improved Dependence Test for Automatic Parallelization and Vectorization***, in *IEEE Transactions on Parallel and Distributed Systems*, vol.2, no.3, July 1991, pp.342-349.

[Kuck72] **D.Kuck, Y.Muraoka and S.Chen** – ***On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup***, in *IEEE Transactions on Computers*, 21(12), pp.1293-1310, December 1972.

[Lamp74] **Leslie Lamport** - ***The parallel execution of DO loops***, in *Communications of the ACM*, 17(2), 1974.

[Leung93] **S. Leung and J. Zahorjan** - **Improving the performance of runtime parallelization**, in Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, volume 28(7), pp. 83-91, July 1993.

[Li90] **Z.Li, Pen-Chung Yew and C.Zhu** - ***An Efficient Data Dependence Analysis for Parallelizing Compilers***, in *IEEE Transactions on Parallel and Distributed Systems*, vol.1, no.1, January 1990, pp.26-34.

[Lich88] **A. Lichnewsky and F. Thomasset** - ***Introducing symbolic problem solving techniques in the dependence testing phases of a vectorizer***, in *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[Love77] **D.B.Loveman** - ***Program improvement by source-to-source transformation***, in *Journal of the ACM, 1*, 24, January 1977, pp.121-145.

[Lu90] **L.Lu and M.Chen** - ***Subdomain Dependence Test for Massive Parallelism***, in *Proceedings of Supercomputing '90*, New York, November 1990.

[MaHL91] **Dror E. Maydan, John L.Henessy and Monica S.Lam** - ***Efficient and Exact Data Dependence Analysis***, in *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 26-28, 1991, in SIGPLAN Notices 26(1991), pp.1-14.

[Maslov94] **Vadim Maslov** - ***Lazy array data-flow dependence analysis***, in *Conference Record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 311-325, Portland, Oregon, January 1994.

[Mura71] **Y.Muraoka** - ***Parallelism exposure and exploitation in programs***, Ph.D. thesis, *Tech.Rep. 71-424*, University of Illinois at Urbana-Champaign, 1971.

[Pugh92] **William Pugh** - ***A practical algorithm for exact array dependence analysis***, in *Communications of the ACM*, vol.35, no.8, August 1992, pp.102-115.

[Pugh94] **William Pugh and David Wonnacott** - ***Nonlinear array dependence analysis***, *Technical Report CS-TR-3372*, Department of Computer Science, University of Maryland, November 1994.

[Poly86] **C. Polychronopoulos** - ***On Program Restructuring, Scheduling and Communication for Parallel Processor Systems***, PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, August 1986.

[Poly87] **C. Polychronopoulos** - ***Advanced loop optimizations for parallel computers***, in *Proceedings of the First International Conference on Supercomputing*, Athens, Greece, June 1987. Springer-Verlag.

[Poly88] **C. Polychronopoulos** - *Parallel Programming and Compilers*, Kluwer Academic Publishing, 1988.

[Rauch94] **L. Rauchwerger and D. Padua** - ***The Privatizing DOALL test: A run-time technique for DOALL loop identification and array privatization***, in *Supercomputing '94, International conference*, July 1994, Manchester, pp. 33--43, Manchester, UK, 1994. Univ of Manchester.

[Reif77] **John H. Reif and Harry R. Lewis** - ***Symbolic evaluation and the global value graph***, in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pp. 104-118, Los Angeles, California, January 1977.

[Reif78] **John H. Reif** - ***Symbolic programming analysis in almost linear time***, in *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pp. 76-83, Tucson, Arizona, January 1978.

[Saltz91] **J. Saltz, H. Berryman and J. Wu** - ***Multiprocessors and run-time compilation***, in *Concurrency: Practice and Experience*, 3(6), pp.573-592, Dec. 1991.

[Sark91] **Vivek Sarkar** - ***PTRAN - The IBM Parallel Translation System***, in *Parallel Functional Languages and Compilers*, ACM Press 1991, pp.309-391.

[Shen90] **Z.Shen, Z.Li and Pen-Chung Yew** - ***An Empirical Study of FORTRAN Programs for Parallelizing Compilers***, in *IEEE Transactions on Parallel and Distributed Systems*, vol.1, no.3, July 1990, pp.356-364.

[Shos77] **R. Shostak** - ***On the sup-inf method for proving Presburger formulas***, JACM, 24(4), pp.529--543, October 1977.

[Schr87] **Alexander Schrijver** - ***Theory of Linear and Integer Programming***, John Wiley & Sons, New York, 1987.

[Van94] **Alexandru Vancea** - ***Categorii de limbaje. Aspecte comparative pe arhitecturi convenționale și distribuite***, Referat în cadrul stagiului de doctorat, Universitatea "Babeș-Bolyai" Cluj-Napoca, 1994.

[Wallace88] **D. Wallace** - ***Dependence of Multi-Dimensional Array References***, in *Proc. of the 2nd International Conference on Supercomputing*, St.Malo, France, 1988.

**[Wolfe89]** **Michael J. Wolfe** - ***Optimizing Supercompilers for Supercomputers***, Research Monographs in Parallel and Distributed Computing, MIT Press, Cambridge, Massachussetts, 1989.

**[Wolfe92]** **Michael Wolfe and Chau-Wen Tseng** - ***The Power Test for Data Dependence***, in *IEEE Transactions on Parallel and Distributed Systems*, vol.3, no.5, September 1992, pp.591-601.

Babeş-Bolyai University, Faculty of Mathematics and Informatics, RO 3400 Cluj-Napoca, str. Kogalniceanu 1, România.

*E-mail address*: {moldovan,vancea}@cs.ubbcluj.ro

Babeş-Bolyai University, Faculty of Economics, RO 3400 Cluj-Napoca, str. Kogalniceanu 1, România.

*E-mail address*: vancea@econ.ubbcluj.ro