Notes on Automatic Parallelization

Computation Structures Group Memo 314
July 17, 1990

Rishiyur S. Nikhil

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988.

Notes on Automatic Parallelization

Rishiyur S. Nikhil
MIT Laboratory for Computer Science
545 Technology Square, Cambridge, MA 02139, USA
nikhil@lcs.mit.edu (617)-253-0237

1 Introduction

In these notes, we give an overview of an important approach to generating code for parallel computers: *automatic parallelization*. Here, the programmer writes an ordinary, sequential program, say in FORTRAN. The compiler then performs sophisticated analysis of an intermediate form of the program (a) to discover what can be done in parallel, and (b) to transform it so that more things can be done in parallel. Finally, using various cost metrics, it partitions the program into parallel tasks and generates code for them, inserting synchronization as necessary.

The analysis techiques are useful for a variety of parallel architectures: pipelined uniprocessors; vector, VLIW, systolic, and superscalar processors; and MIMD machines. In all these cases, the essential issue is the discovery of how instructions can be *re-ordered*.

This *implicit parallelism* approach is attractive for several reasons:

- "Dusty decks": there is a tremendous, existing investment in sequential codes; it would be enormously expensive to rewrite them for parallel machines. However, many people doubt that much parallelism can be extracted from dusty decks (see Section 5.1 for an illustration of the difficulties involved).
- Complexity: Writing a parallel program manually can be hard because we do not yet have adequate mental and formal tools to reason about parallel programs. It can be very difficult to avoid *races* in parallel programs. Such programs can be extremely difficult to debug, because their non-deterministic behavior is often impossible to reproduce.
- Portability: There are many different parallel architectures, and a single parallel form of the program is unlikely to be suitable for all of them. The compiler may have a better chance of picking "good" parallel forms for each architecture.
- Finally, even if the programmer uses explicit parallel constructs, these analysis techniques may be able to draw the programmer's attention to possible races or to possible improvements in parallelization.

To date, most work on automatic parallelization has been for "scientific codes", most programs for which are still written in FORTRAN. The execution time of such programs is dominated by loops that manipulate arrays, and so it is not surprising that the emphasis has been on parallelizing these control and data structures.

2 Structure of a Parallelizing Compiler

Just as the top-level structure of a compiler involves *analysis* (parsing, static semantics) followed by *synthesis* (code generation), a parallelizing code generator itself consists of analysis and synthesis stages. The analysis phase examines the sequential intermediate-language code and attempts to determine all possible things that *could* be done in parallel. The synthesis phase decides which of these things *will* be done in parallel (usually based on some sort of cost criterion), and inserts appropriate code to form parallel threads and to synchronize them.

There are two kinds of analyses performed. In *Control Dependence Analysis*, the compiler tries to determine what are the *control conditions* for each statement. Intuitively, whether a statement is executed or not depends on how it is nested within conditionals and loops. Potentially, execution of this statement could be initiated as soon as its control conditions (predicates of surrounding conditionals and loops) have been evaluated. In fact, potentially, execution of all statements with identical control conditions could be initiated in parallel.

In *Data Dependence Analysis*, the compiler determines how values flow from one statement to another, how storage is re-used for new values, *etc.* This information serves to constrain what might otherwise be done in parallel.

For example, if we have the statement:

```
if p then
    S1;
    S2
endif
```

then control dependence analysis may suggest that `S1` and `S2` can be done in parallel (they have the same control conditions). However, if data dependence analysis determines that `S1` defines a value that is used by `S2`, we may have to retain their sequential structure.

Both control and data dependence analyses begin with the intermediate-language control flow graph[1]. Data dependence analysis annotates this graph with *def-use* and *use-def* information, and control dependence analysis restructures the graph into a *control dependence graph.*

Finally, the synthesis stage produces parallel code from the control dependence graph, inserting appropriate synchronizations or sequentializations to satisfy the data dependence constraints. In addition to just satisfying ordering constraints, it may also take into account the costs of parallelism on a particular architecture in order to decide whether it is worth doing something in parallel or not.

In Section 3 we examine the kinds of data dependence possible between statements in a program, and the methods used to discover this information. In Section 4 we examine control dependence and the methods used to discover it. Finally, in Section 5 we examine how the compiler exploits dependence information to produce parallel programs.

3 Data dependence

3.1 Types of data dependence and data dependence graphs

As described in [1], Chapter 10, the control flow analysis phase of the compiler produces a control flow graph, which describes the sequencing of statements, *i.e.*, the nodes correspond to statements and the (control flow) edges specify sequencing. The compiler then performs data dependence analysis to annotate the flow graph with additional edges called *data dependence* edges.

Suppose statement S_1 normally executes before S_2. We say that there is a data dependence from S_1 to S_2 if reversing this order could lead to incorrect results. In compiling parallel code for the program, therefore, we must ensure that S_2 does not execute before S_1.

A statement may depend on another in one of three ways (the terminology is due to Kuck [8]):

- *Flow dependence*:

  ```
  x  := ...                    (S1)

  ... := ... x ...             (S2)
  ```

  Statement S_1 defines (writes) a value that is subsequently used (read) by S_2. If statement S_2 were executed earlier, it would read the wrong value. We depict flow dependence as $S_1 \longrightarrow S_2$.

- *Antidependence*:

  ```
  ... := ... x ...             (S1)

  x  := ...    ...             (S2)
  ```

  If statement S_2 were executed earlier, it would overwrite the value of x before S_1 used the old value. We depict antidependence as $S_1 \xrightarrow{A} S_2$.

  A key observation: antidependences always come about because we try to *reuse* storage. If we can avoid such reuse, we avoid antidependences which, in turn, increases the potential for parallelism.

- *Output dependence*:

  ```
  x  := ...    ...             (S1)

  x  := ...    ...             (S2)
  ```

  If statement S_2 were executed earlier, then subsequent statements would would use the wrong value. We depict output dependence as $S_1 \xrightarrow{o} S_2$. At first sight, this may appear to be a silly program— why should there be two definitions without an intervening use? However, this situation is not unusual when S_2 is inside a conditional.

Flow dependence is often called "true" dependence, because it represents the genuine communication of data from one statement to another. Anti- and output-dependences arise because we are trying to *reuse* storage: S_1 is not communicating data to S_2, rather, S_2 wants to reuse the storage location used by S_1 for some other, new value. The latter dependences can often be eliminated, therefore, by introducing new storage (new variables) to avoid storage reuse.

In basic blocks, with scalar variables, these dependences are normally easy to compute,[1] and is essentially the computation of *definition-use* and *use-definition* chains, as described in the text [1].

3.2 Data dependence in loops with subscripted variables

Things get more complex when we introduce loops and array subscripts. A statement from one iteration of a loop can depend on a statement (even on itself) from another iteration. When array subscripts are complicated expressions, it may not even be possible to tell whether there is a dependence or not and, if there is, what kind it is.

Example 1:

```
for i = 1 to N do
    ...  := ... A[i] ...          (S1)
    A[i] :=     ...               (S2)
    ...  := ... A[i] ...          (S3)
```

There is an antidependence from S_1 to S_2, and a flow dependence from S_2 to S_3, all within an iteration.

□

Example 2:

```
for i = 2 to N do
    ...  := ... A[i-1] ...        (S1)
    A[i] :=     ...               (S2)
```

There is a flow dependence from S_2 in one iteration to S_1 in the next iteration.

□

Example 3:

```
for i = 1 to N-1 do
    ...  := ... A[i+1] ...        (S1)
    A[i] :=     ...               (S2)
```

There is an antidependence from S_1 in one iteration to S_2 in the next iteration.

□

[1]But see discussion on aliasing in Section 3.6.

The dependence in example 1 is called *loop independent*, *i.e.,* it is within each iteration and not really caused by the loop. The dependences in examples 2 and 3 are called *loop carried*, *i.e.,* the dependences are from one iteration to another. A statement may have a loop carried dependence to itself (*e.g.,* if S_1 is the same statement as S_2 in examples 2 or 3).

3.3 Nested loops and direction vectors

Each dependence arc between statements in loops can also be labelled with a *direction vector*, which is a tuple of $=$, $<$ and $>$ symbols. The width of the tuple is equal to the number of common, nested loops within which both statements appear, *i.e.,* there is one direction symbol for each nested loop index. The leftmost symbol in a direction vector corresponds to iterations of the outermost loop, the next symbol corresponds to iterations of the next nested loop, and so on.

A $=$ symbol indicates that a dependence holds within the same iteration of the corresponding loop. A $<$ symbol indicates that a dependence holds from a lower to a higher index of the corresponding loop. A $>$ symbol indicates that a dependence holds from a higher to a lower index of the corresponding loop.

Example (from [6], with modifications):

```
for i = 1 to N do
    for j = 1 to N do
        A[i,j] := B[i,j-1]              (S1)
        B[i,j] := A[i+1,j]              (S2)
        A[i,j] := A[i-1,j+1]            (S3)
```

The dependence edges are:

- $S_1 \xrightarrow{o} S_3$ labelled $(=,=)$, because both define `A[i,j]` within the same i and j iterations.
- $S_2 \longrightarrow S_1$ labelled $(=,<)$ because S_2 defines `B[i,j]` which is used by S_1 in the subsequent `j` iteration.
- $S_2 \xrightarrow{A} S_1$ labelled $(<,=)$ because S_2 uses `A[i+1,j]` that is redefined by S_1 in the subsequent `i` iteration.
- $S_2 \xrightarrow{A} S_3$ labelled $(<,=)$ because S_2 uses `A[i+1,j]` which is redefined by S_3 in the subsequent `i` iteration.
- $S_3 \longrightarrow S_3$ labelled $(<,>)$ because S_3 defines `A[i,j]` which is used by S_3 in the $(i+1, j-1)$'th iteration.

□

The types of data dependence we have described can be viewed as a refinement of the *def-use* and *use-def* relationships given in Chapter 10 of [1], which are just flow and anti-dependences, respectively. In addition, instead of establishing relationships only amongst entire program variables, we have refined it to establish relationships amongst subscripted components of array variables.

3.4 Data dependence analysis

In general, the index expressions can be quite complicated, so that the relationship between statements may not be as obvious as in our examples. In this section, we consider *data dependence analysis*, *i.e.*, the methods by which the compiler can systematically establish the nature of data dependence, if any, between two statements.

Suppose we have a loop:

```
for i = 1 to N do
    ...
    X[$f(i)$] := ...                      ($S1$)
    ...
    ...      := ... X[$g(i)$] ...         ($S2$)
    ...
```

For example, $f(i)$ may be `i+1` and $g(i)$ may be `i-2`, in which case there is a flow dependence from $S1$ to $S2$. In general, there will be a dependence between $S1$ and $S2$ if, for some two values x and y of the loop index `i`, we have $f(x) = g(y)$. Thus, we want to solve the equation:

$$f(x) - g(y) = 0$$

We are only interested in *integer* solutions in the range $1 \leq x, y \leq N$. The kind of dependence that we are looking for imposes additional constraints:

- For a loop-independent dependence, $x = y$;
- for a loop-carried flow dependence, $x < y$, and
- for a loop-carried antidependence, $x > y$.

If f and g are arbitrary functions, then finding such a solution is not tractable, and the compiler must assume the worst case, *i.e.*, that the two statements *do* depend on each other. In practice, however, f and g are usually simple linear functions of the index (such as `i`, `i-2`, `2*i+1`, ...). In this case, various tests can be applied to obtain an approximate solution. The tests are approximate in that they either return "independence" or "possible dependence". In the latter case the compiler again must assume the worst— that the two statements do depend on each other (for this reason, this analysis is also termed "conservative").

One test is the GCD test, which is based on a well-known theorem on Diophantine equations. Being linear terms, we can write $f(x)$ as $a_0 + a_1x$ and $g(x)$ as $b_0 + b_1x$. There can be a dependence only if $gcd(a_1, b_1)$ divides $b_0 - a_0$. Thus, if it does not divide, we can certify independence and, otherwise, there is a possible dependence.

A second test is the so-called *geometric* test [6], which is based on the intermediate value theorem. The constraints on x and y specify three *regions* in the xy plane that are of interest (see Figure 1). For $x = y$, zeroes of the term $f(x) - g(y)$ must lie on the line from (1,1) to (N,N). For $x < y$, zeroes must be in the triangle bounded by (1,2),(1,N) and (N-1,N). For $x > y$, zeroes must be in the triangle bounded by (2,1),(N,1) and (N,N-1). Further, because the functions are linear, if $f(x) - g(y)$ has a zero in the region of interest, it cannot have the same sign at all corners of the region, *i.e.*, it must cross from positive to negative. Thus, we

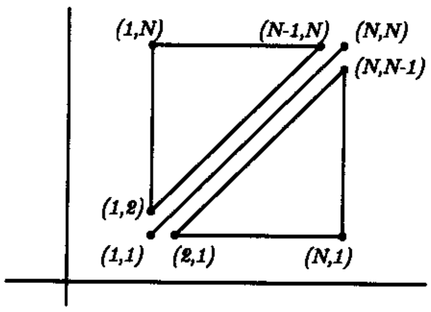

Figure 1: Regions in geometric test for data dependence

evaluate $f(x) - g(y)$ at all the corners of the region. If it has the same sign at all corners, it cannot have a zero in the region, thus guaranteeing independence.

The third test is the *Banerjee-Wolfe* test, which we state here without elaboration:

$$-b_1 - (a_1^- + b_1)^+(N-2) \le b_0 + b_1 - a_0 - a_1 \le -b_1 + (a_1^+ - b_1)^+(N-2)$$

where the superscripts have the following meaning:

$$\begin{aligned} t^+ &= t, & \text{if } t \ge 0 \\ &= 0, & \text{if } t < 0 \\ t^- &= -t, & \text{if } t \le 0 \\ &= 0, & \text{if } t > 0 \end{aligned}$$

This magic formula is taken from [3], which also includes a brief discussion of the mathematics behind it. It is basically a refinement of the geometric test, expressed as a formula.

These tests can be suitably generalized to handle nested loops. The variables x and y become vectors of variables. For example, x_1, ..., x_k (and y_1, ..., y_k) represent values of the index variables in k nested loops. Again, we consider only linear combinations of these variables:

$$\left(a_0 + \sum_{i=1}^{k} a_i x_i\right) - \left(b_0 + \sum_{i=1}^{k} b_i y_i\right) = 0$$

and we look for integer solutions constrained by the loop bounds ($1 \le x_i \le N_i$) and by the kind of dependence ($x_i = y_i$, or $x_i < y_i$, or $x_i > y_i$) that we are checking. All the tests generalize from tests about intersection with a planar region to tests about intersection with a higher-dimensional volume. For more details, please see [3, 5].

3.5 Loop normalization for data dependence analysis

It is often the case that loop indices and subscripts are not in a convenient form to allow immediate application of the above tests. To bring them into the appropriate form, the compiler first performs a transformation called *loop normalization*, so that all loops have indices with a lower bound of 1 and a step of 1, and all subscript expressions are functions of the induction variables.

Example:

```
for k = 2 to N by 3 do
    ... A[k] ...
```

is transformed into:

```
NN := 1 + floor((N-2)/3)
for kk = 1 to NN do
    ... A[2+3*(kk-1)] ...
```

□

Example:

```
j := 1
for k = 1 to N do
    ... A[j] ...
    ...
    j := j + 5
```

is transformed into:

```
j := 1
for k = 1 to N do
    ... A[1+(k-1)*5] ...
    ...
    j := j + 5
```

□

Note that these transformations often go counter to the "reduction in strength" optimization. We accept this here because of the potential for much larger payoff in parallelizing the loop.

3.6 When does dependence analysis fail?

Dependence analysis does not really "fail", since it always errs on the conservative side to produce a correct analysis. However, its result may be *so* conservative that it does not reveal any useful parallelism.

Dependence analysis is usually abandoned when the index expressions are "too complicated". Examples include index expressions that contain arbitrary function calls; index expressions that are themselves array references (*i.e.*, indirection through another array), *etc.* By and large, dependence analysis is attempted only when the array-index expressions are linear functions of the loop index.

Aliasing is another major problem. For example:

```
P: proc(ref A,B: array ...)
       for i = 1 to N do
           A[i] := f(B[i-1])                 (S)
```

There is no dependence from S to itself if A and B are different arrays. However, being reference parameters, it is possible that they refer to the same array, in which case the dependence does exist. There are various options available to the compiler:

- Assume the worst, *i.e.*, that a dependence exists, and produce sequential code for the loop.
- Perform global flow analysis to ensure that no call to the procedure is of the form `P(X,X)`, and compile the body as a parallel loop.
- Perform global flow analysis and, if some calls are of the form `P(X,X)` and some are not, compile two versions of the procedure, one parallel and one sequential, and insert the appropriate reference at each call site.

Pointer variables and heap-allocated objects also thwart dependence analysis, because it provides an extreme case of aliasing. After an application of `car` or `cdr`, the compiler may have no idea where the result is pointing to. Again, it must make the worst-case assumption that it could aliased with anything else. A language with a static type-system is useful here. For example, in a Pascal or CLU program, it is clear that two pointers to objects of different types cannot point at the same storage. In languages like C or Lisp, on the other hand, such analysis is very difficult. The study of dependence analysis for languages with pointers is still in its infancy.

The sharper the dependence analysis, the more scope there is for parallelization. Some languages/compilers allow the user to assist the compiler by inserting pragmas (advice) that assert that two variables are not aliased. The compiler may or may not generate code to check these assertions at run time (if it doesn't, such assertions are unsafe).

4 Control Dependence

4.1 What is control dependence?

In general, the total time to execute a program is minimized if each subcomputation can be initiated as early as possible (so that it completes as early as possible). Of course, this may mean that two subcomputations must run in parallel.

If we temporarily ignore data dependencies, what is the earliest that a subcomputation can be initiated? Answer: as soon as we know whether or not this subcomputation should be executed at all. More specifically,

- We know whether a subcomputation inside an arm of a conditional statement needs to be executed as soon as the predicate of the conditional determines that the arm is to be evaluated.
- We know whether a statement in the body of a loop needs to be executed as soon as the loop predicate determines that an iteration is necessary.

These intuitions are the essence of the notion of *control dependence*.

Formally: A statement or predicate Y is *control dependent* on a predicate X if and only if one outcome of the predicate X causes Y definitely to be executed, while the other outcome of the predicate X causes Y definitely *not* to be executed.

Example:

```
while p1 do
    S1
    if p2 then
        S2;
        S3;
    else
        S4;
        if p3 then S5;
    S6
```

Here, S1, the outer if and S6 are control dependent on p1; S2, S3, S4 and the inner if are control dependent on p2; and, S5 is control dependent on p3.

□

4.2 Control Dependence Analysis and Control Dependence Graphs

The control dependence information can be computed by analyzing the control flow graph of a program. The outcome of this analysis is a *control dependence graph*, which describes, for each predicate, all the predicates and statements that are control dependent on it.

This first step is to compute *immediate post-dominators* for every node in the control flow graph. Intuitively, the immediate post-dominator Y for a node X is the earliest node (in the direction of control flow) for which we can say that if X is executed, then Y *must* be executed sometime later. If we leave out the "earliest" qualification, then Y is just a post-dominator of X (*i.e.,* not necessarily the immediate post-dominator).

In our example program above, p1 is the immediate post-dominator of S6, because if S6 is ever executed, then p1 is the earliest node after S6 that we know must be executed.

Similarly, the immediate post-dominator of p2 is S6. If p2 is executed, we cannot say for sure whether S2, S3, S4, p3 or S5 will be executed (depends on outcome of p2), but we *do* know that S6 will be executed.

In general, if a node X in the control-flow graph has just one successor Y, then Y is the immediate post-dominator for X. A predicate X will have more than one successor; its immediate post-dominator Y is the earliest node where all paths from X come together again. Note that this works even for loops; in our example above, the immediate post-dominator for p1 is STOP, the end of the program, because ultimately the path out of the true-branch of p1 loops around through p1 itself and joins the path out of the false-branch at the next "instruction", which is STOP.

Once we know the post-dominator relationships in a control flow graph, we can identify control dependence information. Let p be a predicate node, and let T_1 be its *immediate* true-successor node. Then, the nodes that are control dependent on p on its true side are:

> T_1
> T_2, the immediate post-dominator of T_1,
> T_3, the immediate post-dominator of T_2,
> ...
> and so on, upto but not including Y, the immediate post-dominator of p.

Intuitively: we know that T_1 is control dependent on p, trivially— it is the first thing executed if we take the true-branch out of p. If T_1 is executed, T_2 is the earliest node after it that we know *must* be executed, *i.e.,* it is also identically control dependent on p. Similarly for T_3, and so on, until we reach the immediate post-dominator of p, which is the earliest node we know that is *not* control-dependent on p.

Similarly, we can also identify the nodes that are control-dependent on p on its false side.

These definitions work for all flow graphs. However, for *structured* flow graphs, *i.e.,* flow graphs in which all control structures correspond to `while` and `for` loops and `if-then-else` statements, the control dependence relationship is especially simple (trivial).

For a conditional statement:

```
if p then S1 else S2
```

then all statements in `S1` and `S2` that are not themselves nested inside conditionals or loops are control dependent on `p`. For a loop:

```
while p do S
```

all statements in S that are not themselves nested inside conditionals or loops are control dependent on `p`.

5 Parallelization based on dependence graphs

After computing the control and data dependence graphs, the compiler can produce parallel code. If we ignore data dependencies for a moment, the method is very simple— we convert every loop into a parallel loop and, for all statements that are identically control dependent in a conditional, we invoke them all in parallel. In our previous example, we would have:

```
while p1 do
    FORK S1
    FORK if p2 then
             FORK S2;
             FORK S3;
         else
             FORK S4;
             FORK if p3 then S5;
    FORK S6
```

i.e., if **p1** is true, we fork **s1**, **p2** and **s6** in parallel. If **p2** is true, we fork **s2** and **s3** in parallel, whereas if **p2** is false, we fork **s4** and the **p3** in parallel.

Obviously, unless the program is quite trivial, this will not be possible because we must also take care of the *data dependencies* between the various statements. For example if there is a flow dependence between **s1** and **s6**, we must ensure that the def in **s1** is executed before the use in **s6**. In other words, what we have described above is the *maximal* parallelism due to control dependence only, and this must be watered down so that the data depencies are respected.

Suppose there is a data dependence (flow or anti-) from statement S_1 to statement S_2. We can deal with it in several ways:

- Execute them sequentially, *i.e.,* execute S_1 before S_2. This is called *implicit synchronization.*
- Insert *explicit synchronization* code, *i.e.,* intiate S_1 and S_2 in parallel, but insert code (*e.g.,* semaphores) in each statement so that the relevant part of S_2 will wait for the relevant part of S_1.
- Remove the dependence, *e.g.,* by introducing new variables, so that the statements can run in parallel without constraint.

In subsequent sections, we will see examples of all these approaches. We will describe them as source-to-source transforms, *i.e.,* the compiler will restructure the source code and also annotate loops as being **PARALLEL** or **VECTOR** loops.

But first, in the next section, we show an example of how "clever hacks" for sequential programs can be detrimental to good parallelization.

5.1 An example illustrating the difficulties of parallelization

It is interesting to note that many tricks that programmers play to make their sequential programs more efficient are, in fact, counter-productive for parallelization. Consider the "1-dimensional relaxation" problem[2]. Here, we are given an initial array X_0. We wish to repeatedly "relax" this array, *i.e.,* construct arrays X_1, X_2, ... where the X_{s+1} depends on X_s in the following manner:

$$X_{s+1}[i] = f(X_s[i-1], X_s[i], X_s[i+1])$$

i.e., the contents of a location is a function of the contents of itself and of its neighbors in the previous version of the array. For example, each array may represent the temperature profile in a thin, long metal bar, and the sequence of arrays approximates the evolution of this temperature profile over time— the change in temperature at a point is a function of the flow of heat towards or away from the point, which, in turn, depends on the current temperature of the point and its neighbors.

Here is a first attempt:

[2] I am grateful to Arvind for showing me this example.

```
for s = 1 to steps do
    for i = 2 to N-1 do
        temp[i] := f(X[i-1],X[i],X[i+1])

    for i = 2 to N-1 do                  --- copy back into X
        X[i] := temp[i]
```

Of course, the second `i`-loop is not doing any "useful" work, so a more efficient version would be to alternate the roles of `X` and `temp` on alternate `s`-iterations.[3]

However, a clever programmer may still be bothered by the use of two arrays and may use the following trick so that only one is used:

```
for s = 1 to steps do
    t1 := X[1]
    t2 := X[2]
    for i = 2 to N-1 do
        X[i] := f(t1,t2,X[i+1])
        t1 := t2
        t2 := X[i+1]
```

Apart from making the program much more difficult to read, there is a serious problem. The original version is trivial to parallelize– there are no loop carried dependencies in the `i` loops, so both loops can immediately be parallelized. The "clever" version is much more difficult to parallelize. The reuse of `t1` and `t2` from one `i`-iteration to the next has introduced loop carried flow dependencies and antidependencies.

In general, it is almost axiomatic that it is much more difficult to parallelize programs that have been highly hand-optimized for serial execution! Reuse of storage almost invariably means sequentialization. The compiler has a much better chance of parallelizing code that has been written in a simple and clean, high-level manner.

5.2 Introducing explicit synchronization

A data dependence from statement S_1 to statement S_2 implies that they must be performed in that order. If the statements are in separate parallel threads, we can force the right sequence by introducing a new semaphore associated with the storage location causing the dependence. After the S_1, we `signal` the semaphore, and before S_2, we `wait` for it.

Suppose we began with:

```
for i = 1 to N do
    ...
    loop body prelude
    ...
    X[i] := ... X[i+1] ...               (S)
    ...
    loop body postlude
    ...
```

[3]If we wanted to impress a funding agency, we would call this the Polyphase Oscillating Boustrophedonic Mono-Dimensional Relaxation Technique.

Here, there is an antidependence from the S of one iteration to the S of the next iteration. We can force the right sequencing by introducing an array of semaphores:

```
sem: array [1:N] of semaphore;

signal(sem[1])

for i = 1 to N do PARALLEL
    ...
    loop body prelude
    ...
    wait(sem[i])
    X[i] := ... X[i+1] ...                    (S)
    signal(sem[i+1])
    ...
    loop body postlude
    ...
```

The iterations can execute in parallel, except that the i+1'st iteration is held up at S until the i'th iteration has completed its S. Note that only the S statements are sequentialized— the loop body preludes and postludes can still execute in parallel. In fact, since the S statements execute sequentially, we will gain some parallel behavior only if the preludes and postludes involve some substantial computations.

5.3 Renaming

An antidependence arises when we reuse a storage location, *i.e.,* we define, then use, then redefine the location. The antidependence specifies that the first use must precede the second definition.

In the last section we introduced explicit synchronization to force the program to respect an antidependence. An alternative method is to remove the antidependence entirely by using fresh storage. This transformation is called *renaming*, because the fresh storage location is named by a new identifier.

In our example, the right-hand side of S used the old values of X, *i.e.,* the values that were in X prior to the loop. The left-hand side of X defines new values of X, *i.e.,* values in X after the loop. We can use a new array XX to hold the new values:

```
XX: array [1:N] of real;

for i = 1 to N do
    loop body prelude
    XX[i] := ... X[i+1] ...                   (S)
    loop body postlude
endfor

...
use XX instead of X in loop sequel ...
...
```

Now, there is no dependence between loop iterations, so they can all be executed in parallel.

5.4 Expansion and Privatization

A generalization of the renaming technique above is *expansion*. Suppose we had the following loop:

```
for i = 1 to N do
    y := f(X[i])                     (S1)
    Z[i] := y * y                    (S2)
```

Here, the programmer has used the temporary variable y because he wants to use the value of f(..) twice, but he does not want to compute f(..) twice. In principle, the computation in any iteration of the loop is independent of the computations in all other iterations. Thus, we may be tempted to annotate the loop as a parallel one:

```
for i = 1 to N do PARALLEL
    y := f(X[i])                     (S1)
    Z[i] := y * y                    (S2)
```

Unfortunately, this will not do; the iterations will step all over each other because they all share the same location y. More formally, there is a loop carried antidependence from S_2 in one iteration to S_1 in the next, *i.e.,* the use of y in S_2 in one iteration should occur before its redefinition by S_1 in the next iteration. Introducing semaphores to force it to respect this antidependence will not do— the loop will simply be sequentialized.

To solve this problem, since the uses of y are all independent, we would like to do some sort of renaming, *i.e.,* to use separate storage locations for each use. Since the number of such locations is N, the number of iterations, it is most convenient to used an indexed name, *i.e.,* we *expand* y into an array:

```
y: array [1:N] of real

for i = 1 to N do parallel
    y[i] := f(X[i])                      (S1)
    Z[i] := y[i] * y[i]                  (S2)
```

In general, the expansion technique replaces an n-dimensional variable by an n+1-dimensional variable. Our example is a special case called *scalar expansion*, where y went from a 0-dimensional variable (a scalar) to a 1-dimensional variable (an array).

An alternative to expansion is *privatization*, where we extend the language to permit declaration of local variables private to each iteration:

```
for i = 1 to N do parallel
    y: real                          --- declare private y
    y    := f(X[i])
    Z[i] := y * y
```

The effect is similar: a separate location y is allocated for each iteration of the loop, so that the antidependence is removed and the loop can execute in parallel. In effect, the previous N-array of y's is now distributed as N individual locations in the N stack frames of the parallel loop iterations.

Privatization has many advantages over expansion. Aside from making the program easier to read, it may have better cache behavior. When y is expanded into an array, y[i] and y[i+1] are likely to be in adjacent memory locations. Unfortunately, the unit of transfer to a processor cache is often not an individual location but a *line*, which is a collection of adjacent locations. Thus, the cache in the processor for iteration i may compete for location y[i+1] even though it is not really going to use it. In the case of privatization, on the other hand, there is no such issue of adjacency.

Second, privatization does not involve any index computation. Third, since y is a single, local variable in each iteration, and each iteration may run on a separate processor, it makes it easier for the register allocator to keep it in a register.

Finally, privatization may be much more storage efficient. For example, suppose there are only 16 processors but the loop has 1000 iterations. Using scalar expansion, we allocate an array with a 1000 locations. However, using privatization, we may allocate only 16 frames and reuse them many times, so that the private variable costs only 16 locations.

5.5 Loop distribution

Loop distribution is the fundamental technique used to vectorize code. The basic idea is to take an inner loop that has several statements in it:

```
for i = 1 to m do
    $S_1$
    ...
    $S_n$
```

and copy it into several loops, one per statement:

```
for i = 1 to m do
    $S_1$

for i = 1 to m do
    ...

...

for i = 1 to m do
    $S_n$
```

The objective is to produce several small inner loops, each of which contains just one statement where the right-hand side contains just one operator that is supported by vector hardware. Such a loop can then be implemented with vector instructions.

Of course, the original dependences must be respected during this transformation. This can be tricky, because loop distribution changes the order in which statements are executed. Before distribution, the order of statements was:

$$S_{1_1}, S_{2_1}, ..., S_{n_1}, \qquad S_{1_2}, S_{2_2}, ..., S_{n_2}, \qquad S_{1_m}, S_{2_m}, ..., S_{n_m}$$

After loop distribution, the order is:

$$S_{1_1}, S_{1_2}, ..., S_{1_m}, \qquad S_{2_1}, S_{2_2}, ..., S_{2_m}, \qquad S_{n_1}, S_{n_2}, ..., S_{n_m}$$

Example:

```
for i = 1 to N do
    B[i] := A[i] + 1                          (S_1)
    A[i] := C[i] * D[i]                       (S_2)
```

There is a loop independent antidependence from S_1 to S_2. After loop distribution, we get:

```
for i = 1 to N do
    B[i] := A[i] + 1                          (S_1)

for i = 1 to N do
    A[i] := C[i] * D[i]                       (S_2)
```

The antidependence is respected. The loop can then be expressed in vector form:

```
    B[1:N] := A[1:N] + 1                      (S_1)
    A[i:N] := C[i:N] * D[i:N]                 (S_2)
```

□

Example:

```
for i = 2 to N do
    B[i] := A[i-1] + 1                        (S_1)
    A[i] := C[i] * D[i]                       (S_2)
```

There is a loop carried flow dependence from S_2 in one iteration to S_1 in the next. Directly distributing the loop, we get:

```
for i = 2 to N do
    B[i] := A[i-1] + 1                        (S_1)

for i = 2 to N do
    A[i] := C[i] * D[i]                       (S_2)
```

but this is incorrect because the flow dependence goes from S_2 to S_1, opposite to the control flow. We must therefore reverse the two statements, and we can also then express them in vector form:

```
    A[2:N] := C[2:N] * D[2:N]                 (S_2)
    B[2:N] := A[1:N-1] + 1                    (S_1)
```

□

5.5.1 Loop distribution method

In loop distribution, we work from inside to out, *i.e.,* we attempt to distribute the innermost loop over the statements in the loop. We assume that all loops except the innermost one run sequentially. Then, we compute the dependence graph relating the statements in the loop. In general, the dependence graph may contain cycles (statements depend on themselves). For example (from [6]):

```
for i = 1 to N do
    for j = 1 to N do
        A[i,j] := B[i,j]                          (S1)
        D[i,j] := B[i,j-3] + A[i-1,j]             (S2)
        B[i,j+1] := A[i,j-1]                      (S3)
```

Assuming the i loop runs sequentially, the dependence graph contains:

- $S1 \longrightarrow S3$ (j-loop carried, because of A)
- $S3 \longrightarrow S2$ (j-loop carried, because of B)
- $S3 \longrightarrow S1$ (j-loop carried, because of B)

The dependences between $S1$ and $S3$ form a cycle.

The nodes in the graph can be partitioned into *strongly-connected* components, or π-blocks. A π-block consists of nodes such that there is a cycle between any pair of them. Further, they are maximal, *i.e.,* there is no cycle between any pair of nodes in different π-blocks. In our example, $S1$ and $S3$ form a π-block and $S2$ forms a (trivial) π-block.

Treating each π-block as an atomic unit, we can topologically sort the π-blocks so that dependences from one π-block to another are always in the direction of the statements of the loop. In our example, we can achieve this as follows:

```
for i = 1 to N do
    for j = 1 to N do
        A[i,j] := B[i,j]                              (S1)
        B[i,j+1] := A[i,j-1]                          (S3)

        D[i,j] := B[i,j-3] + A[i-1,j]                 (S2)
```

Finally, we distribute the j loop around each π-block:

```
for i = 1 to N do
    for j = 1 to N do                             (T1)
        A[i,j] = B[i,j]                               (S1)
        B[i,j+1] := A[i,j-1]                          (S3)

    for j = 1 to N do                             (T2)
        D[i,j] := B[i,j-3] + A[i-1,j]                 (S2)
```

The second j loop can now be vectorized easily.

Loop distribution does not have to stop at the innermost loop. Suppose we now consider the two j loops as atomic statements $T1$ and $T2$, and compute the dependence graph for them. There are two dependences $T1 \longrightarrow T2$; one is loop independent (due to B) and the other is loop carried (due to A). Either way, $T1$ and $T2$ are trivial π-blocks, and they are in the right order, so the i loop can also be distributed:

```
for i = 1 to N do
    for j = 1 to N do                             (T1)
        A[i,j] := B[i,j]
        B[i,j+1] := A[i,j-1]

for i = 1 to N do
    for j = 1 to N do                             (T2)
        D[i,j] := B[i,j-3] + A[i-1,j]
```

Both i-loops can be done in parallel.

5.6 Loop fusion

Earlier, in improving sequential code, we studied loop fusion (also called loop jamming), where we tried to collapse two loops into one to reduce the overhead of loops (predicate tests, pipeline bubbles) and to enable optimizations across two loop bodies.

Loop fusion is, in a sense, the inverse of loop distribution. While parallelization for vector machines prefers loop distribution, parallelization for MIMD machines may prefer loop fusion because, as in sequential code, it can reduce overhead, except that in this case the overhead is that of forking parallel tasks.[4] Thus,

```
for i = 1 to N do parallel
    loop body 1

for i = 1 to N do parallel
    loop body 2
```

may involve much more overhead than

```
for i = 1 to N do parallel
    loop body 1
    loop body 2
```

because the former program incurs the overhead of $2N$ forks and two barrier synchronizations whereas the latter incurs the overhead of only N forks and one barrier synchronization.

5.7 Loop interchange

Consider the following program:

```
for i = 1 to N do
    for j = 1 to N do
        X[i,j] := f(X[i,j+1])                (S)
```

The statement S in the loop body has an antidependence to itself, from its instance in one j iteration to its instance in the next. Thus, the j loop must run sequentially. However, it is perfectly all right for the i loop to run in parallel.

If we were running on a vector machine, this is unfortunate, because only inner loops can be converted into vector form. So, we perform a *loop interchange* transformation:

```
for j = 1 to N do
    for i = 1 to N do
        X[i,j] := f(X[i,j+1])
```

This brings the parallel loop inside, so that it can be vectorized.

Conversely, if the inner loop had been parallel and we were running on a multiprocessor, we could interchange the loops to bring the parallel loop outside in order to minimize the overhead of forking parallel tasks.

[4]This is an example where the choice of parallelization may best be left to the compiler.

Under what conditions is it safe to perform a loop interchange? In general, n nested loops specify an n-dimensional "iteration space", and the loop headers specify a particular linear ("control") traversal through this space. Reversing a loop, or interchanging two loops basically changes the linear traversal through the iteration space. For example, given the loop headers:

```
for i = 1 to 3 do
    for j = 1 to 3 do
        ...
```

the 2-dimensional iteration space is the following:

```
(1,1)    (1,2)    (1,3)
(2,1)    (2,2)    (2,3)
(3,1)    (3,2)    (3,3)
```

and the loop headers specify a left-to-right, top-to-bottom traversal. Interchanging the loops changes it to a top-to-bottom, left-to-right traversal. Reversing the `j` loop specifies a right-to-left, top-to-bottom traversal.

Interchanging two loops is safe whenever the (control) traversal respects the dependence arrows. In our example, the dependence arrows are antidependence arrows going from left to right, from an (i,j) point to an $(i,j+1)$ point in the iteration space. When the loops are interchanged, these antidependences are obviously respected.

As we have already seen, dependence arrows are labelled with a direction vector, a tuple of $<$, $=$ and $>$ symbols. Reversing a loop inverts the "sign" of the appropriate component of the dependence arrow's label, $(..., <, ...)$ to $(..., >, ...)$ and vice versa. Interchanging two loops exchanges the corresponding label components, $(..., l_i, l_j, ...)$ to $(..., l_j, l_i, ...)$. In general, we can reverse loops and interchange loops if the *leftmost* symbol that is $<$ or $>$ symbol does not change sign. In our example, the label of the dependence edge is $(=, <)$; interchanging loops makes the label $(<, =)$); in both cases, the leftmost symbol is $<$, so the interchange is allowed.

Example:

```
for i = 1 to N do
    for j = 1 to N do
        A[i,j] :=       ...                     (S1)
         ...   := ... A[i-1,j] ...              (S2)
```

Here, we have flow dependence arrows going top-to-bottom in the iteration space: $S1 \longrightarrow S2$ labelled $(<, =)$. Clearly, it is safe to interchange the loops.

□

Example:

```
for i = 1 to N do
    for j = 1 to N do
        A[i,j] :=       ...                     (S1)
         ...   := ... A[i-1,j-1] ...            (S2)
```

Here, we have flow dependence arrows going diagonally from top-left to bottom-right: $S1 \longrightarrow S2$ labelled $(<, <)$. Again, it is safe to interchange the loops.

□

Example:

```
for i = 1 to N do
    for j = 1 to N do
        A[i,j] :=     ...                        (S1)
         ...   := ... A[i-i,j+1] ...             (S2)
```

Here, we have flow dependence arrows going diagonally from top-right to bottom-left: $S1 \longrightarrow S2$ labelled $(<,>)$. Thus, it is *not* safe to interchange the loops, *i.e.*, giving $(>,<)$. However, it is safe first to reverse the j loop, *i.e.*, giving $(<,<)$:

```
for i = 1 to N do
    for j = N downto 1 do
        A[i,j] :=     ...
         ...   := ... A[i-i,j+1] ...
```

and then to interchange them, *i.e.*, still $(<,<)$.

□

5.8 Conditionals in loops: loop unswitching and IF-conversion

Loop unswitching is a general transformation that is useful even in sequential programs. Suppose there is a conditional statement inside a loop. Some or all of the predicate expression may be a loop invariant. For example:

```
for i = 1 to N do
    if (A or B[i]) then C[i] := 0
    D[i] := 0
```

Here, the boolean A is a loop invariant, but it is repeatedly tested on each iteration. Loop unswitching performs the following transformation:

```
if A then
    for i = 1 to N do
        C[i] := 0
        D[i] := 0
else
    for i = 1 to N do
        if B[i] then C[i] := 0
        D[i] := 0
```

The transformed program tests A only once instead of N times. Further, the two loops can be vectorized, treating B[i] as a vector mask.

If-conversion is another vectorizing transformation. For example:

```
for i = 1 to N do
    if (A[i] <= 10) then
        A[i] := A[i] + 10
        if (B[i] <= 10 ) then
            B[i] := B[i] + 10
    A[i] := B[i] + A[i]
```

This can be converted into:

```
for i = 1 to N do
    cond1[i] := (A[i] <= 10)
    A[i] := A[i] + 10 where cond1[i]
    cond2[i] := (B[i] <= 10) where cond1[i]
    B[i] := B[i] + 10 where cond2[i]
    A[i] := B[i] + A[i]
```

which can now be vectorized easily, treating the new arrays `cond1` and `cond2` as vector masks.

6 Concluding remarks

The material presented in this section of the notes originates almost exclusively from the work of three research groups: the PARAFRASE project at the University of Illinois at Urbana-Champaign, led by Prof. David Kuck, the "father of dependence analysis" [8]; the PFC project at Rice University, led by Prof. Ken Kennedy, another pioneer in this field [3], and the PTRAN project at IBM Research, Yorktown Heights [9].

Most of the work, to date, has been in trying to parallelize FORTRAN codes which are dominated by loops and arrays, because these are the characteristics of most large, scientific and engineering codes to date, which are the biggest consumers of supercomputer time.

Researchers have just recently begun looking at dependence analysis for languages like C and Lisp, where the use of pointers and heap objects is much more common. This field is still in its infancy, and not much success has been reported to date (leading some to question whether it will ever be viable to try to parallelize sequential programs in these languages).

Another research topic that is still in its infancy and which we have not touched upon at all is the question of choosing, from all the things that *can* be done in parallel, which things actually *will* be done in parallel. If the overhead of forking a thread and synchronizing its data accesses is comparable to the actual work that it does, then it may not be worth it to do it in parallel at all. Of course, this depends on having an accurate cost model for parallelism. We still do not know how to do this well, because it depends on so many aspects of the architecture, system parameters, runtime system, *etc.*

Acknowledgements:

These notes were originally prepared as class notes for 6.035, the MIT undergraduate compilers course. My thanks to my teaching colleagues and the students of that class for their comments on earlier versions. My thanks also to Arvind for reading these notes carefully and suggesting that I convert them into a CSG Memo.

References

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

[2] F. Allen, M. Burke, R. Cytron, J. Ferrante, W. Hsieh, and V. Sarkar. A framework for determining useful parallelism. In *Proc. ACM 1988 International Conference on Supercomputing*, pages 207–215, July 1988.

[3] R. Allen and K. Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[4] R. G. Babb II (*ed.*). *Programming Parallel Processors.* Addison-Wesley, Reading, MA, 1988. Chapters and references on Alliant FX/8, BBN Butterfly, Cray XMP, FPS T Series, IBM 3090, Intel iPSC, Loral Dataflo LDF 100, Sequent Balance. For each machine, code for a program to approximate PI.

[5] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. *ACM SIGPLAN Notices*, 21(7):162–175, July 1986. in Proc. SIGPLAN '86 Symposium on Compiler Construction.

[6] R. Cytron. Compiling for Parallelism, April 3 1989. Tutorial Notes, Third International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[8] D. J. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[9] V. Sarkar. *PTRAN — the IBM Parallel Translation System.* McGraw Hill, 1990 (to appear).

[10] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, MA, USA, 1989. In series: Research Monographs in Parallel and Distributed Computing.

Contents