Dependence is an important tool to analyze and transform programs for faster execution on parallel machines. A transformation is a reordering of statements in the program. Not every reordering preserves the correctness of the program; to ensure that the reordering is in fact "correct" we only have to check that the order of source and sink of every dependence in the program is preserved. This can be used to determine if a reordering is allowed.

There are three type of dependencies. There are true dependencies which refer to a write before read, antidependencies which refer to a read before write and outputdependencies which refer to a write before write. Note that there is no read before read dependence because it doesn't matter in which order 2 ready statements are executed. For loops we have more properties to describe the relationship between source and sink. There is for example a direction vector which compares () the loop indices from the source and sink of the dependence. An other special property is the distance vector which holds for every loop index the difference between the index on sink and source. With this in mind we can define a loop independent dependence as a dependence with only "=" entries in the direction vector. All other dependencies in a loop are called loop carried because they rely on the loop executed more than one time to take effect. We can also define a level of loop-carried dependence which is the nesting level of the loop that corresponds to the leftmost non-"=" direction in the direction vector. We can set the level of loop-independent dependence to infinity.

These main definitions allow theorems which can be used to transform programs based on their dependence graph. The main idea is that we search for acyclic parts which can be transformed easily. If all parts a cyclic we leave the outer loop as it is and move on to the next inner loop while removing all parts of the graph containing the outer loop. With this technique we can produce good results with a simple an efficient algorithm.

Open questions:
1) Why do we have to find the set of maximal strongly connected regions? (Figure 2.1)
2) How is each S_i reduced to a singe node? (Figure 2.1)
3) What is a naturally induced dependence graph? (Figure 2.1)
4) How to generate the dependence graph? Consider every tupel of 2 statements?

Summary on Data Dependencies

There are two main types of dependency. Control dependence and Data dependence. Control dependence comes from statements like if/else, such that you can`t change a statement in else with the stamen in if clause. Data dependency comes from writing and reading from same memory.

Data Dependence is an important way to check whether if the transformation of the code is still have same meaning with the original code. If the transformations preserve order of dependency for every source and sink the transformed code will have the same meaning and it will be correct.

True, anti, output, loop dependencies are the different classifications for dependencies. True dependence occurs when first you write to a memory location and then read it from there in following statements. Anti-dependence is visa-versa of true dependence it first read from memory and upcoming statement writes to the same memory. Output dependency occurs when two statements write on to same memory location before any read occurs. Loop dependence is the dependence that occurs in a loop or nested loop. Have two types; loop carried and loop independent.

Loop independent is not something dependent on the iteration of the loop. It means that there is dependency for the statements under the same iteration. For example, in every iteration two statements in the loop produces a value and then consumes it subsequently. Loop carried dependence does not have a dependency if we execute only one iteration of the loop. The value produced in one loop is being consumed in the subsequent iterations. There are two properties used to help in loop dependencies; distance vector and direction vector. Distance vector keeps the track of number of iterations that dependency crosses for each index in loop. Direction vector keeps relation between loop indices on source and sink of the dependence.

Parallelization is in a sense what we want to do. It is converting separate iterations to parallel threads so they can be executed on different cores at the same time without affecting the dependencies/output/meaning of the program.

Questions;

1) In parallelization it says it is possible to convert two sequential loops if loop carries no dependence. In this sense is it always possible to convert loop independent ones to parallel?
2) Are there any corner cases where loop carried and independent dependency occurs at the same time? If so how should we approach to it?
3) How should I approach if there were such situation where two statements in level-2and level-k (k>2) had dependency between them?
4) On the page 52 I didn`t understand why we applied loop reversal. How does it maintain the order iteration?

Summary for

Dependence: Theory and Practice

The book chapter is used as an introduction to dependence and its relationship with compilers. In particular, dependence is a useful tool to argue about the correctness of translations: any program transformation which preserves the order of all dependences in the program preserves correctness. Hence the theory of dependence is integral to any optimization which reorders the sequence of statements.

First, the authors introduce multiple attributes of dependencies by which they are distinguished. The *type* of a dependency classifies it as either a true-, anti- or output-dependency. If a variable is written in statement s1 and later used in a statement s2, we call this a true dependency. The reverse situation, so a variable from which was read in s1 is now updated with a new value, is called an anti-dependency. Finally, the situation in which two consecutive statements both write into the same variable is called an output-dependency.

Loops affect dependencies in a peculiar way: a statement of one iteration can have dependencies to statements in previous iterations. This behavior is captured by *direction* and *distance* vectors. Both vectors describe dependencies in terms of loop iteration variables, where each component keeps track of one such variable. Hence, n nested loops would produce n-vectors. The distance vector describes how many iterations of a loop dependence spans. In turn, a direction vector describes the relationship between the values of indices by taking one of three values <, = and >. A value < indicates a dependency to a previous iteration, > to a following iteration and finally = will indicate a dependency to the same iteration. Hence a direction vector of (<,=,>) for a statement means that its first component points to a previous iteration, its second component to the current one and its third to a future one.

After defining those notions the authors proceed to give an implementation of a vectorization algorithm. The aim of vectorization is to determine whether a scalar program (e.g. one consisting of multiple nested for-loops) to a vector implementation, which processes multiple threads at once. Of particular interest is the conversion of loops, in which we want to run each iteration asynchronously in parallel. To this end they first construct a so called *dependence graph*, which has statements as nodes and the dependencies between statements as edges between their corresponding states. The algorithm proceeds by recursively visiting each loop-level from the outermost to the innermost loop. For each level, it selects the relevant strongly connected components of the graph. If a SSC is acyclic (and therefore a singleton), the corresponding statement can be vectorized. The algorithm is then called recursively on the subgraph in which all edges of the current level are removed.

SUMMARY "***INTRODUCTION TO PARALLIZATION AND DEPENDENCE***" 20.11.2012

The exploitation of te paralleslism has ceated a new dimension in computer science. Parallel computing is a form of computation in which many calculations are carried out simultaneously,the ability to execute several program segments in parallel reqires each segment to be independentis of the other segements.The principle strategy for seeking parallelism is to look for the a data decomposition in which parallel tasks performs similar operations on different elements of the data arrays.

In 1966, Bernstein ,defined set of conditions based on which 2 process can execute in parallel.

Let p1,p2 be two process i1,i2 be inputs sets and o1,o2 be the output sets respectively then, p1 ans p2 execute in parallel iff they are independent and does not produce conflit results:

i.e. $i1 \cap o2 = \Phi$,

$i2 \cap o2 = \Phi$ and

$o1 \cap o2 = \Phi$

*In other words if the processes are flow independent, anti dependent and output independent.

*Parallel execution of the two process produce same results regardsless of whether they are excecuted in sequential in any order or in parallel.this is possible only when of one process will not be used as the input to the other process.

*Parallism is commutaive but not associative.

DATA DEPENDENCE There is data dependence from statement S1 to the statement S2 if and only if both the statements access the same memory location and at least one of them stores into them. and there is a fesible run time execution path from S1 to S2.

Here we ensure that the data produced ad consumed in the right order , we must insure that we not interchange the load and store to same location.

Five types of data dependies are listed below:

1.) **Flow Dependence**:A statement S2 is flow dependet on the statemnt S2 if the execution path exist from S1 to S2 and if atleast one output of S1 feeds in as input to S2.

S1. A = 3

S2. B = A

2.)**Anti dependence**:A statement S2 is flow dependet on the statement S2 if S2 follws S1 in program order and if output of S2 overlaps input to S1.

S1. A = B + 1

S2.   B = 7

3.)**Output dependence**:An instruction is control dependent on a preceding instruction if the outcome of latter determines whether former should be executed or not. In the following example, instruction S2 is control dependent on instruction S1. However, S3 is not control dependent upon S1 because S3 is always executed irrespective of outcome of S1.

S1.   if (a == b)

S2.   a = a + b

S3.   b = a + b

4.)**I/O dependence**: it occurs not because the same variable involved but because the sae file referenced by both the I/O operations.

5.)**Unknown dependence:** The dependence relation between the statements cannot be determined in the following situations:

a. the subscript of a varibale is itself subscribed(indirect addresing).

b. A variable apperas more than once with subscripts having different coeffcients of the loop variable.

c. The subscript is nonlinear in the loop index variable.

Whe one or more of these conditions exist, a conservative assuption is to claim unknown dependence exists.

DEPENDENCE IN LOOPS There exist a dependence from stmt S1 to S2 in a common nest of loops iff:

*There exist two iteration vectors i and j for the nest s.t. i<j or i=j.

*There is path from S1 to S2 in loop body.

*S1 access the memory location on iteration i and S2 access the memory location M on iteration j.

*One of these access is write.

TRANSFORMATION A program transformation is any operation that takes a computer program and generates another program and has same meaning as original program.

In many cases the transformed program is required to be semantically equivalent to the original, relative to a particular formal semantics.

REORDERING TRANSFORMATION Transformation that merely changes the order of the execution of the code, without adding or deleting any statement.

Two statements refers to the memory location M before tansfrmation will refer to the same memory location M after the transformation however, it ay reverse the order in which the statements refer to the memory locationand therefore reversing the dependencies.

A reordering transformation preserve a dependence if it preserve relative execution order of the source ad the sink of the dependence.

FANDAMENTAL THEOREM OF DEPENDENCE Any reordering trasformation that preserve every dependence in a program preserves the meaning of that program.

DIRECTION VECTOR Suppose there exist a data dependence from S1 on iteration i of a loop nest to the statement S2 on iteration j; the dependence distance vector d(i,j) is defined as a vector length n s.t. $\mathbf{d(i,j)_k = j_k - i_k}$

i.e "<" if $d(i,j)_k > 0$

"=" if $d(i,j)_k = 0$

">" if $d(i,j)_k < 0$

*Direction vectors are very useful for understanding loop reordering transformations because they summarize the realtion between index vector and source and sink of the dependence.

*They are also beneficial in calculating the level of loop-carried dependences.

DISTANCE VECTOR TRANSFORMATIONS

Transformation that is applied to the loop nest and does not rearrage the statements in the loop body, then transformation is valid if , none of the direction vectors for the dependences with the source and sink in the nest has non "=" component i.e. ">".

SUBSCRIPT SEPARATIBILITY

A subscript said to be zero index variable (ZIV) if the subscript position contains no index in either reference. A subscript siad to be sigle index variable(SIV) if only one index occurs in that position. Any subscript with more than one index is said to b multiple index variable(MIV).

example

```
Do i= L1,U1
                              Do j=L2,U2
                                                            Do k=L3,U3


          A(5,i+1,J) = A(N,i,k) + C
```

here first subscript is ZIV because 5 and N are constants, the second is SIV since only index i appears in the dimension an last one is MIV because j and k both appear in the dimension.

In testing the multidimenstional aarays we say that subscrit position is separale if its indices do not occur in the other subscripts. if two different subscripts contains the same index , we say that they are coupled. If all the subscripts are separable , we may compute the direction vector for each subscript independently and merge the direction vector on a propostional basis.

SUBSCRIPT PARTITION

We need to classify all the subscripts in a pair of array references as separable or as part of some minimal coupled group. A coupled group is minimal an cannot be partitioned into into two non-empty subgroups with distinct sets of indices. Once a partioned is achieved , each saparabe subscript and each coupled group has completely disjoint sets of indices.

Each partition can be tested in isolation and the resulting distance or direction vector merged withot loss of any precision.Since each variable and coupled subscript group contains a unique subset of indices , a merge may be thought of as a cartisian product.

example Do i=L1,U1

Do j=L2,U2

A(i+1,5)=A(i,N)+C

since j does not appear in any sunscript , we must assume the full set of direction vectors for the j loop and thus a merge yields the following set of direction vectors for both the dimention. {(<,<) (<,=) (<,>)}

LOOP CARRIED DEPENDENCIE

Statement S2 has loop carried dependence on the statemnt S1 iff S1 refernce memory location M on iteration i and S" reference the memory location M on iteration j and $d(i,j)_k > 0$.

The level of the loop carried dependence is the index of the leftmost non-"=" of the D(i,j) for the dependence.

LOOP INDEPENDENCE Statement

S2 has loop carried dependence on the statemnt S1 iff S1 refernce memory location M on iteration i and S" reference the memory location M on iteration j and i=J.

No common loop is necessary for the loop independent dependence since it arise from statement position.

ITERATION REORDERING

A Transformation that reorders the iterations of a k-level loop, without making any other changes , is valif if the loop caries no dependence.

VECTORIZATION

Vectorization is a special case of parallelization, in which software programs that by default perform one operation at a time on a single thread are modified to perform multiple operations simultaneously.

Vectorization is the more limited process of converting a computer program from a scalar implementation, which processes a single pair of operands at a time, to a vector implementation which processes one operation on multiple pairs of operands at once. The term comes from the convention of putting operands into vectors or arrays.

The task of the vectorization is to determine whether statements in inner loop can be vectorized by directly rewriting them in FORTRAN 90.

Do I 1,N
X(I)=x(I)+C can be vectorized as X(1:N)=X(1:N)+C

LOOP VECTORIZATION

A statement contained in at least one loop can be vectorized by directly rewriting in FORTRAN 90 if statement is not in cycle dependence.

QUESTIONS

1.) Can data dependence can arise dynamically? if yes how to deal with it.

2.)How Loop carried dependencies inhibit paralliesim?

3.)what is number of dependencies? and how it affects tansformations.

Optimizing Compilers for Modern Architectures
Chapter 2: Dependence, Theory & Practice

As base of loop vectorization or in general loop parallelization Allen and Kennedy introduce notations and techniques to detect and categorize data/memory dependencies. Guided by the imperative programming language FORTRAN and the underlying memory model three types of dependencies are distinguished.

True Dependencies
Also known as RAW (read after write) or flow dependencies, where a statement reads a memory location written by a statement prior in the code.

Antidependencies
Also known as WAR (write after read) dependencies, in case a statement later in the program code writes a location read by a statement prior in the code.

Output Dependencies
Also known as WAW (write after write) dependencies, which occur when two statements write at the same memory location.

Furthermore, the authors differentiate between loop-carried and loop-independent dependencies. The former ones are introduced by statements within a loop which access the same memory location in different iterations. The latter ones are caused by the top-to-bottom execution order of the source code and may or may not be surrounded by loops. In case they are, the memory location in question needs to be accessed within the same iteration by both statements and—as for all dependencies—at least one of them must be a writing one. While loop-independent dependencies are primarily needed for statement reordering, loop-carried dependencies are crucial for (loop) parallelization.

To determine the type of a dependencies as well as to argue about the iterations involved (in case they are loop-carried) so called distance and direction vectors are defined. They describe the relative offset and, respectively, the order between the executions of two dependent memory accesses with regards to all surrounding iteration variables. A simple dependence detection algorithm, also base of the Delta test, is presented before, in the end, the correctness of transformations, especially loop parallelization, is proven for certain shapes of those vectors. This means that they provide information sufficient for sound parallelization/vectorization as performed by the presented algorithms.

Questions

- How to compute the dependencies for non linear array subscripts (efficiently)?
- How to utilize loop transformations, e.g., loop splitting, to simplify/eliminate dependencies?
- How to extend the given algorithms to handle control dependencies too?