Automatic Exploitation Of Concurrency in C: Is It Really So Hard?

by

Lori S. Grob

Ultracomputer Research Laboratory
Courant Institute of Mathematical Sciences
715 Broadway, 10th Floor
New York, NY 10003

Ultracomputer Note #140
June, 1988

This work was supported in part by the US Department of Energy under grant number DE-FG02-88ER25052 and in part by IBM contract number N00039-84-R-0605.

Preliminary versions of this paper were presented at the *Association Francaise des Utilisateurs d'Unix*, Paris, France, March 1988 and at the *European Unix Systems User Group Workshop*, Dublin, Ireland, Autumn 1987.

ABSTRACT

The vast majority of work being done today on the automatic exploitation of concurrency, for both multiprocessors and vector machines, is not being done for C. Yet there are many important applications, written in C, which would benefit immensely from the speedup produced by such techniques. Many more applications would be written in C were there optimizing, vectorizing and parallelizing compilers for the language.

In this paper we consider whether it is possible to have compilers that will automatically exploit concurrency in C. We discuss the relationship between automatic exploitation of concurrency for the purposes of vectorizing and multiprocessing. We review the basic techniques and transformations used and examine the necessary conditions to perform these transformations, with examples in C. Several elements of the C language and programming style, such as pointers and recursion, make it difficult to do the necessary data flow analysis. There are two possible approaches to this problem: to bypass code or blocks of code that contain ‘‘difficult’’ features and be unable to apply optimizations to these fragments, or to suitably restrict the language. We examine the choices made by the few available vectorizing and parallelizing C compilers and consider what the future may hold in the light of current research.

1. Introduction

The goal of automatic parallelization, whether for multiprocessors or for vector machines, is to take a program with serial semantics and have the compiler or preprocessor produce a parallel program.

The programmer avoids having to deal with the difficulties of synchronization and other issues that arise in the writing and debugging of explicitly parallel programs (Grob and Lipkis [86]). There is a guarantee that the semantics of the program will be preserved and that the resulting program will be correct. The arguments that are used for stating the need for and advantages of optimizing compilers are applicable here as well. The compiler will be able to find parallelism that the programmer does not see. Programmers who set out to solve a problem shouldn’t have to be experts in the kinds of data flow techniques used to find non-obvious potential parallelism, or in the transformations necessary to exploit parallelism. A parallelizing compiler will be able to work on library routines and other functions that were not written by the programmer. However, these arguments do not eliminate the usefulness or the desirability of explicit parallel programming. It is often useful for a programmer to be able to manipulate the parallelism and control the asynchrony him or herself.

Because the potential benefits of automatic parallelization are considerable there is a great deal of interest in the topic. However, most work on the compilers that discover and exploit concurrency is being done on FORTRAN and not on C. There are features of the C programming language that make it difficult to do the analysis necessary to optimize, much less exploit concurrency. These features are unconstrained pointers and other forms of aliases. The first few attempts to write parallelizing or vectorizing compilers for C either restrained the language by limiting the use of pointers or didn’t attempt to parallelize statements or blocks of code that contained pointers.

In this paper, we will attempt to explain what it is necessary to know about a program in order to parallelize it safely and why the C language poses problems. We will also give a few approaches to the problem that appear potentially promising. In order to present these issues clearly, we will first present an overview of the necessary background material.

2. Architectural Models

We are concerned with automatic exploitation of parallelism for three basic architectural models: vector processors, multiprocessors and very long instruction word (VLIW) machines. The techniques and constraints involved in finding parallelism for these architectures are similar although the architectures themselves are quite different.

2.1. Vector Processors

Conceptually, the idea behind a vector processor is quite simple. An operation is performed with two arrays as the operands. Instead of a loop iterating through all the elements of the arrays, all the elements of the arrays are processed in parallel. For many kinds of programs, which spend the bulk of their execution time on vector operations, this can be a significant speedup.

Architecturally, what happens in most modern vector processors is somewhat different. The elements of the arrays that are the operands of the vector operation are passed into a pipeline of processors. Each processor performs a part of the primitive operation on whatever data it is given. The data then moves on to the next processor which performs its part of the operation. At the same time the previous processors are performing on other data. It takes some amount of time for the first operands to move through the pipeline. There is a result on every tick thereafter.

2.2. Multiprocessors

The multiprocessors referred to in this paper are MIMD[1] machines. MIMD machines consist of many processors working together on a single job. Each processor may operate autonomously, not in lock step with the other processors. The processors may or may not share memory. They may communicate over a bus or a network.

2.3. VLIW Machines

VLIW machines have a word size large enough to hold multiple machine instructions. Instructions loaded into the word at the same time are executed in parallel. The difficulty is in deciding which instructions may be executed in parallel and in restructuring the program to execute as many instructions as possible in parallel without changing the results of the program.

[1] In the taxonomy of Flynn [66], MIMD is the category of Multiple Instruction stream, Multiple Data stream computers (i.e. asynchronous multiprocessors); whereas in SIMD designs there is just a Single Instruction stream for the Multiple Data streams.

2.4. Hybrids

Although these are the three architectures that we are going to discuss, the categories are not absolute. There are vector machines that contain more than one set of vector processors and do vector processing in parallel and there are multiprocessors and VLIW machines that have vector units.

3. Automatic Parallelization

The potential for parallelism arises from the fact that some statements or operations do not have to be executed in the order that they are written. The ordering is often random or at least imposed only by the necessity of putting them in some linear sequence. The relative order of every statement is often not necessitated by the problem. If a group of operations or statements are independent of each other in terms of flow of control and in terms of data, then they can be executed in at the same time.

3.1. Levels of Parallelism

Parallelism can be found at many different levels in a program. Expression level parallelism refers to the evaluation of several different parts of the expression at the same time. For instance, in an expression containing addition operations and multiplication operations the various operands could be evaluated at the same time, subject to the rules of precedence. This is the type of parallelism found in VLIW machines. Further, in VLIW machines, there is expression level parallelism for many expressions happening at the same time. There are also statements that can be run in parallel. There is loop level parallelism, where the iterates are run in parallel. Ideally, it is desirable to find all the parallelism in a program, no matter at what level it exists. It may be questioned whether it is worth executing something in parallel if it is only a single expression. The answer to this depends on the architecture and the implementation of those features that create and control parallelism.

Numerical programs and other scientific applications are often very regularly structured. The computation is often done in a series of loops. The data structures are arrays or matrices. This type of program lends itself very well to automatic parallelization and vectorization as generally practiced.

Non-numeric applications may have a less regular structure. But if a program is written with the idea that it is going to be automatically parallelized, that is, bearing in mind the types of transformations done and the constructs that can be parallelized, these applications can get significant speedup (Lee, Kruskal and Kuck [85]). When non-numeric applications with irregular *shapes* are parallelized the speedup obtained is likely to be less significant. Because VLIW machines exploit fine grain or expression level parallelism they are less susceptible to this problem.

Most vectorizing compilers and parallelizing compilers for multiprocessing look for parallelism at the loop level. The vectorizing compilers attempt to parallelize the innermost loop in a nested construct. Each statement in the innermost loop will be vectorized separately. The compilers for multiprocessing attempt to parallelize the outermost loop in order to minimize synchronization and to achieve optimal parallelism. The goal of here is to execute loops in parallel with all the iterates being done at once. It is loop level parallelism that we are going to discuss.

3.2. When Can A Loop Be Executed In Parallel?

In order to understand the complexities of parallelism of this sort, one must picture all the iterations of a loop executing in parallel and at different speeds. The danger is that there may be a dependence existing between several statements in the same or different iterations. The existence of a dependence may make it necessary to have stores and fetches of various elements completed in a prescribed order. Vectorizing or multiprocessing may allow that ordering to be violated, causing the program result to be incorrect. So it is necessary to discover the dependences that exist and decide if they are of a type that will prevent the vectorization or parallelization of the loop. The dataflow analysis that is necessary to prove that no dependence exists is a superset of the kinds of dataflow analysis that are done for traditional optimization.

3.2.1. Dependences

Dependence analysis is important in many areas of optimization and critical in automatic parallelization, including deciding which statements may be loaded into a single VLIW instruction and executed together. A data dependence exists between one statement and another or between a statement and itself when the same memory location or elements of the same array are accessed in those statements. This may enforce an ordering or serialization of the statements in which the dependence exists.

Not all dependences prohibit parallelization or optimization. In some cases, the code must be transformed and the dependence removed before any parallelization or optimization can occur. Sometimes all the dependent code must be moved or treated as a unit. In other cases, the dependence will preclude any parallelization or other potential alteration in the sequence of execution.

The data dependences which occur between statements have been put into three classifications: flow dependence, anti-dependence, and output dependence (for more background on dependence see Wolfe[82] and Burke and Cytron[86]).

A flow dependence means data will ‘‘flow’’ forward from one statement to another. An example of a flow dependence is:

```
for (i=0; i<=100; i++) {
        x = w + z;
        a[i] = x;
          .
          .
          .
}
```

The second statement is dependent upon data from the first. The store must be completed before the load.

An anti-dependence is the reverse of a flow dependence. The load precedes the store. In order to get a correct value stored the order of the statements must be preserved.

```
for (i=0; i <= 100; i++) {
        a[i] = x;
```

```
x = w + z;
            .
            .
            .
}
```

There is no true computational dependence between the statements. But there is a storage dependence between them. The fact that load of *x* must complete before the store into *x* could be a problem if this loop were run in parallel or if these instructions were executed together.

Recent research suggests a solution to the problem of anti-dependences or storage related dependences called *variable renaming* (Cytron and Ferrante[87]). The use and reuse of variables is sometimes a matter of happenstance or storage conservation. Once it is determined that there is no computational dependence between two statements, only a storage dependence, then the dependence can be eliminated of by renaming the variables. Of course, all future references to the variable must use the last variable name. So the transformed code would look like this:

```
for (i=0; i<= 100; i++) {
        a[i] = x1;
        x2 = w + z;
            .
            .
            .
}
```

An output dependence is caused by several stores to the same location. Again, in order to get the correct result and not alter the observed behavior of the program the stores must be completed in the order that they are written.

```
for (i=0; i<= 100; i++) {
        x = a[i];
          .
          .
        x = w + z;
          .
          .
}
```

As with anti-dependences there is no data dependence here only a storage dependence. This too can be resolved with variable renaming. When removing output dependences the last assignment to a variable must have the original variable name or all later occurrences must be renamed. This is so that if the variable is printed immediately after the loop the observed behavior will be the same.

Flow dependences are more of a problem. The semantics of the program may require that data be communicated from one statement to a subsequent statement. Sometimes forward substitution can be used to remove the dependence while retaining the flow of data.

```
for (i = 0; i <= 100; i++) {
        a = x + 5;          (1)
        b = a - 10;         (2)
        c = a + b;      (3)
        x = c + y;      (4)
}
```

There are flow dependences between (1) and (2) and between (1) and (3) because the value of *a* is carried forward, likewise between (2) and (3) with respect to *b* and between (3) and (4) with respect to *c*.

After forward substitution this looks like this.

```
for ( i = 0; i <= 100; i++) {
        a = x + 5;
        b = x - 5;
        c = 2 * x;
        d = 2 * x + y;
}
```

3.2.2. Analysis of Array Subscripts

In the previous section dependences involving scalars were discussed. But in many programs the majority of the computing is done by manipulating arrays and elements of arrays. When there are loops, possibly nested, and arrays with one or more subscript variables, it is not always obvious when two or more statements will be accessing the same locations on the same or different iterations. The most conservative approaches for discovering dependences do not attempt to analyze the subscript of the arrays to see if a dependence exists. Instead, a store to any element of an array is considered to have changed the entire array. This may unnecessarily prevent parallelization or vectorization since it may appear that there is a dependence where none exists. This is erring on the side of conservatism. It is much safer to see dependences where none actually exist and therefore not to parallelize a section of code, then to execute it concurrently and violate a dependence.

More sophisticated dependence analysis is based upon forming dependence equations based upon linear subscript expressions (Wolfe [82]). The corresponding expressions are set equal to each other and if there is a solution to the equation then there may be a dependence.

3.2.2.1. Context

It is possible to get a more precise answer to the question of the existence of a dependence if the *context* of the loop is considered. Context means that only integer solutions within the bounds of the loop are considered as solutions to the dependence equations. So if it can be proved that a dependence exists, but not within the bounds of the loop, then the loop can be parallelized.

```
for (i = 0; i <= 49; i++) {
        a[i] = b[i];
        d[i] = a[100 - i];
```

}

The stores into *a* in this loop are into elements 0,1,2,...,49 and the fetches from *a* are from elements 100,99,98,...,51. When i equals 50 there is a dependence between these two statements but since the bounds of the loop go from 0 to 49 no dependence exists within the context of this loop. Even if the loop bounds are symbolic they may well be available by compile time. Clever analysis and constant propagation may maximize the chances of parallelizing such a loop.

3.2.2.2. Plausiblity

There might be many solutions to the dependence equations. But only some of them will fall within the context of the dependences. The context is also a function of the direction that the dependence runs in. If there is a dependence that exists only if one loop is running forward while the other loop runs backward and both are actually running in the normal fashion from one to n then that dependence is *implausible* and doesn't have to be considered. Information about the chronological relationship between the subscripts of the two elements that are being tested is summarized in a *direction vector* (Wolfe [82]), (Burke and Cytron [86]). There is one entry in the direction vector for each subscript of the array. This vector can be interpreted according to the direction that the loop is running. This can help decide if a dependence that exists is plausible.

3.2.2.3. A Hierarchy of Tests

There are several different tests for independence. They can be thought of as forming a hierarchy that ranges from the most general that looks for any integer solutions to the dependence equations, to tests that look for only integer solutions within the loop bounds, to tests that look for integer solutions within loop bounds and consider the direction vectors. In order to minimize the cost of the analysis the cheapest test is done first. If this test proves independence it is not necessary to do any further analysis. If this test fails to prove independence then a more expensive test is applied. If there is still a dependence then finally the most exact test is applied. By using the tests in a hierarchy there is a possibility of holding down the amount of analysis necessary. This is because all the tests are conservative and will err on the side of not proving independence. Once independence is proved no further testing is necessary.

3.2.2.4. Flow Dependences

The dependences that prevent parallelization or vectorization are flow dependences. A flow dependence is one that occurs when a store of a variable precedes a fetch. It can also exist between iterations as when there is is a store into an element in one iteration that is fetched in the next or in any succeeding iteration. This can be in a single statement or between two statements and can be thought of as a recurrence.

3.2.2.5. Vectorization

Vectorization works by running one statement at a time in parallel over all its iterates. It cannot take place if a cycle of dependences exists in the statement or statements being vectorized.

If there exists a flow dependence between two statements in a loop, and the statements are to be vectorized then one statement must be completely executed before the other is begun or the answer will be incorrect. Given the following code fragment:

```
for (i = 0; i <= 100; i++) {
        a[i] = b[i] + c[i];
        d[i] = a[i] + e[i];
}
```

If this was executed serially then in every iterate the store of *a[i]* would be completed before the access of *a[i]*. But if both of these statements were vectorized, then there might be at least one iterate where the load of a[i] in the second statement would be executed before the store of the same a[i], in the first statement. The solution to this problem is to vectorize the first statement and when it completes to do the second statement also in parallel.

Early vectorizers would parallelize all or none of a loop. More recent compilers try to do as much as they can.

```
for (i=0; i <= 100; i++)
        a[i] = a[i-1] + c[i] + d[i];
```

A clever vectorizer would separate the last piece of the statment and vectorize the addition of *c* and *d*.

3.2.2.5.1. Conditionals

Conditionals that limit which elements of an array are used as operands can be dealt with by taking the test and making a conditional array out of it.

```
for (i=0; i <= 100; i++)
        if (a[i] < n )
                d[i] = a[i] + b[i];
```

Can be transformed as follows:

```
Vectorize: c[i] = a[i] < n;
Vectorize: d[i] = a[i] + b[i] where c[i];
```

Then the vector operation is performed on all the elements where the conditional array contains a one. If the conditional array is very sparse it may not be worth it to do the vectorization. This can be applied to nested conditionals as well. The more deeply nested the conditionals are, the more important it is to

estimate the sparsity of the conditional array. Since each layer of nesting probably eliminates some number of iterates that can be vectorized, it may be that it can become unprofitable to perform the vectorization.

Conditionals that cause an exit from the original loop cannot be dealt with this way.

3.2.2.6. Multiprocessing

Multiprocessing of a loop cannot take place if there is a flow dependence between iterations. If there is a store in one iteration that is fetched in a subsequent iteration the two iterations cannot be run completely in parallel. There is every possibility that they will complete out of order and the result will be incorrect. If there is a flow dependence but it is only in the same iteration then that loop can be multiprocessed. Sometimes the dependences can be worked around by synchronizing the references. This provides an added cost and removes some of the benefit, sometimes all of the benefit of parallelization. If the dependence is across iterations of the outer loop or if there are many synchronization points in the outer loop then it may be most profitable to parallelize an inner loop. The desire to obtain the most parallelism by parallelizing the outermost loop must be balanced against the cost of synchronizations in that loop.

3.2.2.6.1. Doacross

If the loops have a flow dependence across iterations that makes it impossible to run them completely in parallel it may still be possible to get some of the benefit of parallelism. There is a construct[2] called the *doacross* (Cytron [86]). It is a parallel loop with a delay. The delay allows a dependence in the form of a reference to a previous iterate be satisfied.

The general idea of doacross is that when the first iterate is executed, the fetch in the second iterate doesn't begin until sufficient time has elapsed to allow the store in the first iterate to have completed. This is repeated for all iterates. This may allow all of the iterates to be running at the same time in an overlapping fashion.

The profitability of doacross is directly linked to the length of the delay. A delay of zero is equivalent to a fully parallel loop.

3.2.2.7. Transformations for Concurrency

Code can be transformed in order to make it vectorizable or parallelizable or in order to increase the benefit of concurrency. These transformations must not cause the program to give an incorrect answer or raise an exception where none was raised before the transformation.

[2] Now in use by Alliant and University of Illinois Cedar Project among others.

3.2.2.7.1. Getting Rid of Dependences

The first group of transformations are architecture independent. They are used to put code into a form which can be either parallelized or vectorized. One goal is to get rid of all output and anti-dependences and as many flow dependences as possible.

The first transformation is variable renaming which removes anti and output dependences by using more storage. This was discussed in a previous section. The next transformation, scalar expansion uses a similar idea. Here a scalar that is used in a loop is *promoted* to an array. This can eliminate a noncomputational dependence that could prohibit multiprocessing or vectorization. The price of this expansion is a large amount of storage which should be reclaimed at the earliest possible time.

3.2.2.7.2. Other Transformations

The number of implied and explicit gotos can be reduced by reproducing the program from the control flow graph. This has the effect of straightening the code, thus exposing more of it for possible parallelization. Also, loops can be normalized by making their lower and upper bounds zero or one through n and adjusting the array subscripts.

Since vectorizing code means vectorizing inner loops, if there is a dependence in the inner loop but not the outer loop it would be desirable to be able to exchange the inner and outer loops. The preconditions for loop interchange are that the exchange must not turn an anti-dependence into a flow dependence or an flow dependence into a anti-dependence.

Another transformation used for vectorization is to convert if statements to loops, where possible, and then vectorize them.

There are many other code transformations used to increase the amount of parallelism possible. These are some of the most important ones.

3.2.2.8. Dataflow Analysis

In order to prove the independence which will allow code to be parallelized and otherwise transformed, dataflow analysis techniques are applied to the code. The more thoroughly the program can be analyzed the more it will be possible to discover places where the code can be profitably parallelized.

Dataflow analysis is done by summarizing information about a program in equations and then using those equations in the analysis. The equations are recomputed along all possible control paths in a program. The information is propagated forward or backward along the control flow graph depending on what the information is. The analysis becomes much more difficult to do on a program that consists of many modules. It is necessary to know which variables are changed or defined in a every module.

3.2.2.8.1. Interprocedural Dataflow Analysis

Aliases are a problem in interprocedural analysis. They arise, even in languages without pointers, from parameters passed by reference. Aliases also arise from several structures mapped onto the same

area of memory, such as unions in C and equivalences in FORTRAN. The most conservative forms of interprocedural analysis deal with the presense of aliases by making worst case assumptions. The worst case assumption is that any function call may modify any variable. So after the function call it is assumed that all the variables that are reachable by that function have been modified. This will affect all optimizations as well as any possibility of vectorization or parallelization. But, is it necessary to be that conservative?

There are groups that have applied modern dataflow techniques to this problem and been able to handle the problems of aliasing (Burke [87]). The information about a module is summarized and propagated back to its call point where it is matched with the parameters to the module. This information is then used in the parent module and propagated back to its call point. Thus, it is possible to to determine the actual aliasing.

3.2.2.8.1.1. Separate Compilation

In order to do good interprocedural analysis all code for all the modules must be present. In the presence of separate compilation this is not always feasible. Certainly it would be better not to have to reanalyze and reoptimize the entire program every time one module is changed. If the program calls library functions or links in some module where it does not have access to the code then it cannot do the analysis for that module and must make conservative assumptions in the analysis of the calling modules. For the most part, it is a problem that remains unsolved today. There are several partial solutions and attempts at solutions.

The MIPS solution is to link the intermediate code (Himmelstein, Chow and Enderby [87]). When a module is altered the necessary parts of the code are all there to analyze and optimize. This lets you reoptimize, after changing a module, but forces you to look at the entire program with every change.

There is research being done that says when a single module of a program is changed it isn't necessary to recompile the entire thing (Cooper, Kennedy, and Torczon [86]). Given a database of information containing compilation dependences, this information is compared with changes in the programs interprocedural information and a list of procedures requiring recompilation is produced. A change in one module might not necessitate the reoptimization of the entire program. This can be tremendously complicated, of course, by the presense of global variables and pointers.

4. The C Programming Language

Before any code can be automatically parallelized or vectorized, it is necessary to perform dependence analysis on it. The analysis is used to discover statements and expressions that are independent of each other and therefore can be safely run in parallel. The C programming language contains features that can cause aliasing and make dependence analysis very difficult. Recursion, unions, global variables, parameters passed by address all make dependence analysis difficult. In all these cases, the set of possible aliases is relatively restricted and the analysis can be done using techniques that have been developed for FORTRAN (Burke[87], Allen et al.[87]), which contains similar language constructs. The unconstrained

nature of pointers C is not comparable to any feature of FORTRAN. It is the use of pointers in C that makes the dependence analysis so difficult.

Pointers can provide multiple paths to the same memory location. Dependence analysis is used to discover places where two operations or statements are accessing the same memory location. Pointers can render this difficult or impossible.

4.1. Solutions

In the general case, it is not possible to know, at compile time, what address a pointer will contain. This means that there is no overall solution to the problem. There have been partial solutions.

One approach is to treat all pointer variables, globals and variables that have their address taken as if they are subscripts and references to the same array, which represents all of memory. They are all aliased together and could be pointing almost anywhere. Then whatever dependence analysis is done takes those aliases into consideration. The analysis does not usually include interprocedural analysis, which is difficult and costly. This approach is conservative, but will allow some parallelism even in the presence of pointers.

This approach can be made less conservative by means of directives to pass information to the compiler. The compiler can be told to vectorize or parallelize in places where it may appear that a dependence exists. This will cause a wrong answer if a dependence really does exist. It also relies on the programmer to understand enough about the process of automatic parallelization to know if a dependence exists.

We believe that it may be possible to take a less conservative approach. Although, in the general case, it is not possible to know what value a pointer is holding, in certain specific cases it should be possible to prove a pointer is not pointing to a given location. Knowing that a particular variable is not a possible alias for a given pointer will make it possible to perform better dependence analysis. More detailed dependence analysis will increase the amount of parallelizable or vectorizable code that can be discovered.

4.1.1. Description

We begin at pointer declaration by placing each pointer declared into a matching with its aliases. A matching is a group containing all the known pointers to an object and the object. If a pointer is not set to point anywhere in its declaration then it is matched with, $\varnothing$, the empty set. We continue, statement by statement, changing the matchings by looking at assignment statements. Pointers whose alias information cannot be determined in this fashion, such as globals or pointers returned from functions are placed into the matching whose matched object is *, symbolizing that they may point to almost anything.

If the pointer is a parameter to the function it can be handled in two different ways. When no interprocedural analysis is done, pointers that are passed in as parameters are placed into the same matching as globals and other pointers about which nothing can be determined. An assignment to one of these pointers will remove it from the * matching set. A dereference of one of them must be considered as

potentially changing any variable. This can be quite limiting in C, since arrays are always passed as pointers to the first element. In effect, this means that it will be difficult to prove that no dependence exists in statements that use these pointers.

If interprocedural analysis is done then there is more information available. Using the interprocedural analysis techniques that were developed for the more restricted aliases that arise in FORTRAN (Burke[87]), the alias sets for the parameters and some other information such as the relative starting addresses of arrays and a token identifying a pointer as pointing to an array can be passed into the function from the call site. With this information it is possible to avoid considering the parameters as potential aliases for everything else.

If these techniques are followed without alteration, pointers into arrays will be considered to be pointing to the entire array. This will create a situation that corresponds to dataflow analysis before subscript analysis, when a reference to one element of an array was considered a reference to all of them. This is safe and conservative but severely limits the amount of vectorization and parallelization obtainable, especially in the case of the less fine grain architectures that attempt to discover parallelism at the loop level.

The solution to this problem is to convert the pointers into array subscripts, just for the analysis, and perform subscript analysis (Burke and Cytron[86], Wolfe[82]) on them. The alias information computed with the matchings is used to check for the existence of dependences in conjunction with the subscript analysis. After the analysis is completed the subscripts are reoptimized back into pointer references.

Pointers incremented in loops can be converted into induction variables. An induction variable is a variable that changes by a fixed amount on every iteration of a loop. The incrementing or decrementing of a pointer as it iterates through an array usually corresponds to this. Once again, after the analysis is completed the induction variable is reoptimized back to a pointer reference.

It is up to the programmer to ensure that the pointer does not iterate off the end of the array.

4.1.2. Example

The following example will clarify the previous explanation. No interprocedural analysis is presumed.

```
test(l,p)
int *l, *p;                          {l,p→*}        /* l and p may point anywhere */
{
        int x, *y=&x;                {y→ x}
        int *q;                      {q→∅}
        int a[100],b[100],c[100];
                                     /* Summary */  {l,p→*}{y→ x}{q→∅}
(1)     x = 6;
        *y = 5;
        p = y;                       {p,y→x}        /* p and y point to x */
                                     /* Summary */ {l→*}{p,y→x}{q→∅}
        .
```

```
        .       /* Code initializing arrays a and b */
          .
(2)     q = a;                          {q→a[0]}
        y = b;                          {y→b[0]}
        p = c;                          {p→c[0]}
        while(q!=NULL) {
                *p = *q + *y;
                q++;                    {q→a[*]}
                y++;                    {y→b[*]}
                p++;                    {p→c[*]}
        }
}
```

The matchings are shown on the right. Because there is no interprocedural analysis in this example *l* and *p* are assumed to be able to point to almost anything, which is signified by the *. For the sake of clarity, the matchings are shown only on the lines that affect them and when they are summarized.

In section *(1)*, there is an output dependence created by the assignment to **y* which follows the assignment to *x*. The matching clearly shows that *y* points to *x*. There is no flow dependence between the assignment of **y* and the access of *y* in the next statement. There is no ordering implyed by the two statements. The assignment to *p* removes it from the * matching and places it in a specific matching. It is no longer necessary to assume that *p* can point to almost anything. *P* is set equal to *y* and *y* is set to point to *x*. So *p* is put into the matching that contains *x*.

In section *(2)*, *q* is set to point to *a[0]*. *Y* is set to point to *b[0]* and *p* is set to point to *c[0]*. Inside the loop, because of the increment, *q*, *y* and *p* may be pointing anywhere in their respective arrays. As long as the pointers do not go beyond the end of the arrays, it is easy to see that this loop could be safely parallelized The dependences that exist between the assignment statement and the various increments will disappear when the loop is rewritten as an array with subscripts. The matching information shows that there is no aliasing between the pointers.

4.2. Conclusions

We think using these techniques in combination, it should be possible to discover dependences and therefore statements and expressions that can be safely parallelized and vectorized, even though they contain pointers. It will not be possible in all cases to prove independence. But we think that in a number of cases, given the relatively regular structure and use of pointers in the kind of numerical and scientific applications that drive this kind of work, it will be possible to discover significant parallelism.

There are real difficulties involved in writing optimizing, parallelizing or vectorizing C compilers. It appears to be possible to to work around these difficulties without restricting the language or totally giving up the advantages of parallelism. The cost of this is a great deal of analysis. It remains to be seen whether the benefits from the added parallelism will outweigh the costs of this analysis.

As dataflow techniques improve, it will become possible to make less pessimistic assumptions about data relationships without sacrificing program correctness. These improved techniques will be applied first to FORTRAN, which offers a much more static environment for experimentation and which has a much higher demand for parallelism. Inevitably these improved techniques will be applied to C and an amount of parallelism and optimization which was deemed miraculous a short time ago will appear to be unacceptable in the light of what can be demanded.

We do not believe that a general solution to the problems of pointers will be found. But we believe that it is possible to obtain a useful amount of parallelism from C even under present conditions. We also believe that the approaches that we have discussed here and other approaches that will arise from later dataflow work and be applied to this problem promise a greater amount of obtainable concurrency without restricting the language.

5. Acknowledgements

This work was inspired by and stemmed from conversations with Ron Cytron of IBM Research, who encouraged me to pursue this and whose own work contributed to the foundation upon which this rests. Stefan Freudenberger of Multiflow Computer, Inc., participated in later discussions. Jim Lipkis of the NYU Ultracomputer Project contributed to all these discussions and provided encouragement and insight. My heartfelt thanks to these three people.

References

F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, ‘‘An Overview of the PTRAN Analysis System for Multiprocessing’’, *Proc. International Conference on Supercomputing*, 1987.

M. Burke, ‘‘An Interval-Based Approach to Exhaustive and Incremental Interprocedural Dataflow Analysis’’, *IBM Research Report RC12702*, 1987.

M. Burke and R. Cytron, ‘‘Interprocedural Dependence Analysis and Parallelization’’, *Proc. the SIGPLAN ’86 Symposium on Compiler Construction*, 21(7):162-175, July 1986.

K. Cooper, K. Kennedy, and Linda Torczon, ‘‘Interprocedural Optimization: Eliminating Unnecessary Recompilation’’, *Proc. the SIGPLAN ’86 Symposium on Compiler Construction*, 21(7):58-67, July 1986.

R. Cytron ‘‘Doacross: Beyond Vectorization for Multiprocessors’’, *Proc. International Conference on Parallel Processing*, August 1986.

R. Cytron and J. Ferrante, ‘‘What’s in a Name?: or The Value of Renaming for Parallelism Detection and Storage Allocation’’, *Proc.*International August 1987.

M. J. Flynn, ‘‘Very High Speed Computing Systems’’, *IEEE Trans.*, **C-54**, pp. 1901-1909, 1966.

L. Grob and J. Lipkis, ‘‘Approaches to Parallel programming on Multiprocessors’’, *Proc. of the EUUG Autumn 1986*,

M. Himmelstein, F. Chow, and K. Enderby, ‘‘Cross-Module Optimizations: Its Implementation and Benefits’’, *Conf. Proc. Summer 1987 Usenix*, June 1987.

G. Lee, C. Kruskal, and D. Kuck, ‘‘An Empirical Study of Automatic Restructuring of Nonnumerical Programs for Parallel Processors’’, *IEEE Trans. Comp.*, October 1985.

M. Wolfe, ‘‘Optimizing Supercompilers for Supercomputers’’, *PhD thesis, University of Illinois at Urbana-Champaign*, 1982.