Region-based Parallelization Using the Region Test

Jay Hoeflinger, Yunheung Paek, David Padua
Department of Computer Science
University of Illinois at Urbana-Champaign,
1304 West Springfield Avenue, Urbana, IL 61801, USA
{hoefling,paek,padua}@cs.uiuc.edu

Abstract

*In this paper, we discuss the advantages of using array access region summaries to parallelize programs. We reformulate the definition of the types of dependence in terms of array regions, and present a dependence test called the Region Test to use the definitions. The Region Test is designed to detect dependence between arbitrary regions of a program, including loop nests and whole subroutines. This allows us to exploit task parallelism, as well as loop parallelism. The Region Test also facilitates the generation of run-time dependence tests in situations where insufficient information exists at compile time to carry out the dependence test.

Keywords: Compilers, Dependence Test, Parallelization

1 Introduction

Detection of parallelism in sequential codes is the key ingredient in successful automatic parallelization for all types of modern multiprocessors. The increasing use of computers with large numbers of fast processors is making it ever more important for compilers to find large amounts parallelism within a program. In order to do that, compilers must have an efficient and flexible way of representing and using summarized access patterns [1].

For this purpose, a compiler typically uses a standard representation, such as *triplet notation* (also called *regular section descriptor*) [4, 5, 8], or *convex regions* [11, 12, 14]. These representations are based on a standard, simple access pattern, for purposes of efficiency. Through our analysis of the Perfect [24] and SPEC [26] benchmarks, and several full scientific codes from the National Center for Supercomputing Applications(NCSA), we found that these representations sometimes must discard critical information. This prevents the parallelization of certain loops.

This motivated us to develop a new representation called the **Access Region**, a generalization of triplet notation. In this representation, we attempt to retain maximum precision without sacrificing efficiency. This representation can be manipulated in a self-contained, abstract form, which has the desirable effect of allowing summarization and comparison of the access patterns of arrays in arbitrary program sections. In a previous paper [1], we focused on the description of the Access Region form.

The focus of this paper is a new dependence test, called the **Region Test**, which builds upon the Access Region representation. We are implementing the Region Test in the **Polaris** compiler [2]. In Section 2, we first characterize the Region Test in comparison with other dependence tests. In Section 3, we formally describe the Access Region representation and operations on it. In Section 4, we describe how the Region Test deals with the dependence problem at the loop, interprocedural and task levels, and how it generates runtime dependence tests.

2 Related Work

Two of the earliest dependence testing techniques were the GCD Test and the Banerjee Test [18]. In practice, these were simple, efficient, and successful at determining dependence, since most subscript expressions occuring in real scientific programs are very simple. However, the simplicity of the tests results in some limitations. For instance, they are not effective for determining dependence for multidimensional arrays with *coupled* subscripts, as stated in [13]. Several *multiple-subscript* tests have been developed to overcome this limitation: the multidimensional GCD Test [19], the Power Test [25], the λ-test [21], and the Delta Test [13].

The above tests are exact in commonly occuring special cases, but in some cases are still too conservative. The Omega Test [16] provides a more general method, based on sets of constraints representing a convex region, capable of handling dependence problems in terms of integer programming problems. Although integer programming techniques use worst-case exponential time algorithms, the Omega Test has proven to be efficient and successful for solving dependence problems.

All of the preceding tests have the common draw-back that they cannot handle subscript expressions which are non-affine. Non-affine expressions sometimes occur in the original form of programs, and are often exposed during compiler transformations, due to the closed form expressions for induction variables, or the forward propagation of expressions. To solve this problem, the Range Test [5] was built to handle non-affine subscript expressions, through strong symbolic range analysis. The Range Test, the most powerful

*The research described is supported by Army contract #DABT63-95-C-0097. This work is not necessarily representative of the positions or policies of the Army or the Government.

test used by Polaris, has shown good results in parallelizing many loops in real programs, but is not a multiple-subscript test, so is not effective for handling coupled-subscripts.

All of these tests were designed to exploit loop parallelism. They build a *dependence graph* to represent the dependence relationship between pairs of array references within a loop. To do this, they compare all pairs of array references which may contribute to a dependence within loops. So, we say that they are *point-to-point* tests. The dependence graph is very useful for parallelization and enables sophisticated loop transformations [15]. But, point-to-point methods may need $O(n^2)$ comparisons where n is the number of array references in the loop. This can be very expensive if the loop contains many references. Also, these techniques are not appropriate to analyze dependence between arbitrary program sections, such as loop nests and subroutine calls. A common way to address these problems is to summarize the array accesses within program sections, then intersect them to determine dependence between the sections. To implement such summaries, several previous works [12, 14] used a set of constraints representing a convex region.

The Region Test endeavors to combine the flexibility and efficiency of these summarization techniques with the symbolic strength of the Range Test. The Region Test has several advantages over other tests. First, capitalizing on the expressiveness of the Access Region representation, and the powerful manipulation routines defined for Access Regions, the Region Test is able to parallelize loops with very complex access patterns. It tests all dimensions of an array simultaneously, so it can handle coupled-subscripts. The test also uses some of the ideas from the Range Test and can therefore handle non-affine expressions.

Secondly, since this summarization can be performed over arbitrary sections of the program, this allows us to deal with task[†] parallelism as well as loop parallelism. This provides a tool for doing interprocedural dependence analysis without *subroutine inlining* [2, 22].

Thirdly, the Region Test helps us to move closer to the ideal, which is to use *renaming* to remove all anti and output dependences. This has been possible in only a limited way in the past, since precise region analysis has not been available. With the Access Region representation, we can more accurately represent the region of an array involved in a dependence, which is crucial to eliminating it.

Finally, it incorporates array privatization [4] and run-time dependence testing under a uniform framework.

3 Access Regions

The key to the precision and usefulness of the Region Test is the Access Region representation of array access patterns. In this section we describe the representation and the operations defined for manipulating the representation.

3.1 Description of Access Regions

Despite the apparent complexities of many data access patterns, our hand analysis and some experimental evidence [5] reveal that, quite often, the access regions of interest have a *regularity* of structure. By this we mean that the accesses seldom happen in random or chaotic ways. They are actually very structured, by design. Furthermore, related access regions often have a *similarity* of structure. This is true because several references to a single array within a loop nest are generally accessed using the same loop indices and with similar subscript expressions.

We try to capture the properties of regularity and similarity in our region access representation, in terms of two important characteristics: *strides* and *spans*. A regular access region $\mathcal{R}$ for an array is composed of a finite number of discrete array elements, arranged in a region according to regular strides. The collection of elements separated by the same stride stretches for a finite distance which we call a span. For example, in Figure 1, the access region for `U(2*J+K+I)` in the outermost loop has three strides: 1, 2 and 4, which are due to the indices involved in the access: `I`, `J` and `K`, respectively. By varying only `I` from 1 to `N`, `N-1` is the span, that is, the difference between the maximum and minimum of the range in $\mathcal{R}$. Similarly, `2*(M-1)` and `L` are the spans corresponding to `J` and `K`, respectively.

```
real V(N,N), Y(N,N), Z(-M:M) ···
 ···
do I = 1, N, 1
   do J = 1, M, 1
         Y(I,J) = V(J+1,J) + Z(II(J))
      do K = 0, L, 4
         V(K+1,K+2) = U(2*J+K+I)
      enddo
   enddo
   if (M<N) then
      V(I,I) = W(I+10) * U(C)
   endif
enddo
```

Figure 1: Code example

For the formal description of $\mathcal{R}$, we first define $X(f(\mathcal{I}))$ to be a reference to an array X within a program section such as a loop or a procedure. The *subscript function* $f(\mathcal{I})$ is defined on a set of indices $\mathcal{I} = (i_1, i_2, \cdots, i_m)$ in which each index i_k varies within the program section. If X is multi-dimensional, then we *linearize*[‡] the subscript expression to generate $f(\mathcal{I})$, similar to what was done by Burke and Cytron [17]. The *Access Region* $\mathcal{R}$ for $X(f(\mathcal{I}))$ is a structured, symbolic set of subscript values of X, in the sense that, given a set of constants for all the variables involved, one could generate the precise set of array subscripts involved in the access pattern. $\mathcal{R}$ is represented by the four-tuple

$\mathcal{R} =$ (*Access Descriptor, Accuracy, Access Type, Predicates*)

The access descriptor is represented by a list of the strides and spans of $\mathcal{R}$: $\mathcal{A}^{\sigma_{i_1},\sigma_{i_2},\cdots,\sigma_{i_m}}_{\delta_{i_1},\delta_{i_2},\cdots,\delta_{i_m}} + o$ where σ_{i_k} and δ_{i_k} are the stride and span, respectively, due to the index i_k, and o is the offset from zero of the first element in $\mathcal{R}$. In Figure 1, $\mathcal{A}^{1,2,4}_{N-1,2(M-1),L} + 3$ is the access descriptor for `U(2*J+K+I)` with regard to $\mathcal{I} =$ `(I,J,K)`. The multidimensional array reference `Y(I,J)` would be linearized using the expression `I+(J-1)*N`, to generate the abstract access descriptor $\mathcal{A}^{1,N}_{N-1,N(M-1)} + 1$. Similarly, we accurately represent the diagonal access of `V(I,I)` as $\mathcal{A}^{N+1}_{N^2-1} + 1$. Traditional triplet notation would express such a diagonal access pattern as `V(1:N:1,1:N:1)`, which is inaccurate. The access descriptor here makes it obvious that the access is regular, with a single stride. The access descriptor for the single element `U(C)` can be represented by $\mathcal{A}^{\sigma^*}_0 + C$ where σ^* can be any integer

[†] between loop nests or subroutines invocations

[‡] Instead of using m expressions to denote access within an m-dimensional array, a single expression is formed with the dimensionality factors made plain. For example, for the array `X`, declared `X(10,20)`, the subscript `(I,J)` is linearized to become `(I+10*(J-1))`. Linearization allows the Region Test to deal with coupled subscripts in its dependence testing.

number. In the Access Region, a zero span indicates a single element region, and a negative span indicates an empty region.

The accuracy is MUST if the access descriptor of $\mathcal{R}$ accurately represents the set of elements accessed in the code, and otherwise, MAY. The access type is READ or WRITE depending on whether $\mathcal{R}$ is read or written by the corresponding reference(s). The predicate is a condition under which $\mathcal{R}$ is valid. Predicates add accuracy to $\mathcal{R}$. For example, in Figure 1, even when the compiler cannot determine if M<N is true, the access region for W can be represented accurately by ($\mathcal{A}^1_{N-1}+11$,MUST,READ,M<N). Alternatively, if we can afford to lose accuracy, we can represent this as ($\mathcal{A}^1_{N-1}+11$,MAY,WRITE,$\mathbf{T}$) where $\mathbf{T}$ represents TRUE, which means that there is no constraint on this region.

An array subscript expression need not be affine, but it is required to be a *monotonic function* [18] within the index ranges. Fortunately, most subscript expressions encountered in scientific programs are monotonic [5] (at least over the index ranges involved), and those few which are non-monotonic may be converted to a monotonic function with a possible accuracy loss. For instance in Figure 1, the subscript function for Z is II(J). We cannot determine whether II(J) is monotonic unless we have knowledge about the contents of II. When we do not have such knowledge, we may convert the subscript function for Z into a monotonic function J′ such that J′ is an artificial index ranging from -M to M, covering all elements of Z. Based on the new index, we could generate the approximate region ($\mathcal{A}^1_{2M}-M$,MAY,READ,$\mathbf{T}$). In [1], we detailed many other issues relating to the generation of Access Regions for array references in a program.

3.2 Operations on Access Regions

Some of the basic operations [1] defined for Access Regions may be described briefly as follows:

aggregation is a set union operation. All references to an array of the appropriate type (READ or WRITE) within a certain section of code are collected and represented as one or more Access Regions.

subtraction is a set subtraction operation. All elements in one Access Region are removed from another Access Region.

intersection is a set intersection operation. After the operation, only elements which are common to both Access Regions remain in the result.

coalescing is a technique for simplifying the structure of the region representation by reducing the number of strides and spans in the Access Region.

To assist with the understanding of *coalescing*, consider the region $\mathcal{R}$ with the access descriptor $\mathcal{A}^{1,5}_{4,20}$. Here, we can find that $\mathcal{R}$ can be equivalently represented by using another descriptor $\mathcal{A}^1_{24}$, yet the latter is simpler because it has only one pair of stride and span. In this case, coalescing would replace the original descriptor by the simpler one in order to simplify the region representation for $\mathcal{R}$.

We are implementing the Access Region operations in a module which we call the *Region Processor*. Since the access regions being operated on will often involve unknown values, the Region Processor is designed to proceed with its operations by making assumptions, and returning the expressions representing those assumptions as conditions under which the result is correct. The condition expressions could be evaluated at runtime, when totally accurate information is available, to choose between alternative transformations.

The work of the Region Processor is supported by two crucial features of Polaris. First, the program is represented in Gated Single Assignment(GSA) form [4]. The GSA form makes it easy to determine which definition of a variable is used at any point in the program, and the conditions under which the definition is used. Second, Polaris has a rich symbolic expression environment [5], consisting of expression simplification, range propagation and the range dictionary, which makes the value ranges for variables available at any point in the program. These features provide a mechanism which can determine relationships between variables even when their exact values are unknown.

4 The Region Test

The Access Region representation and the Region Processor support the work of the Region Test. In this section we will show how to reformulate the definition of dependences in terms of regions, and how to use that reformulation to parallelize programs.

4.1 Dependences

There are three types of dependence generally considered between array references inside a loop: *flow*, *anti* and *output* [18]. Each of these can exist between two different iterations of the loop (*cross-iteration*), or within a particular iteration of the loop (*intra-iteration*).

A cross-iteration flow dependence occurs when an array element is written on one iteration, then the same element is read on a later iteration. A cross-iteration anti dependence occurs when an array element is read on one iteration and written on a later iteration. A cross-iteration output dependence occurs when the same element is written on two different iterations. Cross-iteration dependences prevent parallelization if nothing is done to remove them.

The Region Test, as presented here, determines the existence of these types of dependence in cross-iteration form. It has variable granularity, in that it can test for loop-based dependence between individual array references, dependence for whole loops, or dependence between arbitrarily large code sections for a particular array.

4.2 Intraprocedural Loop-Based Dependence Analysis

In the following discussion, we will assume a loop

```
do I = l, u, s
   ··· X(f(I)) ···
enddo
```

whose index starts at the value l and strides in steps of s to an upper bound of u.

The definitions given in Section 4.1 for the various types of cross-iteration dependences may be reformulated in terms of array regions. The basis for the reformulation is that a cross-iteration dependence exists if and only if the portion of an array accessed prior to an arbitrary iteration, call it iteration t, and the portion accessed in iteration t and later will overlap. From now on, we will refer to the iterations from l to $t-s$ as the *prior* iterations, and the iterations from t to u as the *later* iterations.

We write the region-based definition of a cross-iteration flow dependence for an array X in such a loop as follows:

$$\texttt{FLOW}: [\mathbf{W}(\texttt{X})_{l:t-s:s} \cap \mathbf{R}(\texttt{X})_{t:u:s}]_{t=l+s:u:s}.$$

```
detect-loop-dependences(L)
  foreach variable V in L do
    compute-dependences(V,L,FLOW,ANTI,OUTPUT)
    if (FLOW ≠ ∅)
      return 'L is serial'   !has flow dependences
    if (ANTI ≠ ∅ ∨ OUTPUT ≠ ∅)
      eliminate-dependences(ANTI,OUTPUT,V,L)
    if (ANTI ≠ ∅ ∨ OUTPUT ≠ ∅)
      return 'L is serial'   !can't eliminate all dependences
  endfor
  return 'L is parallel'   !no variable causes dependences
end
compute-dependences(V,L,FLOW,ANTI,OUTPUT)
  W(V)_{l:t-s:s} ← aggregate(L,V,WRITE,l : t − s : s)
  R(V)_{l:t-s:s} ← aggregate(L,V,READ,l : t − s : s)
  W(V)_{t:u:s} ← aggregate(L,V,WRITE,t : u : s)
  R(V)_{t:u:s} ← aggregate(L,V,READ,t : u : s)
  FLOW ← [W(V)_{l:t-s:s} ∩ R(V)_{t:u:s}]_{t=l+s:u:s}
  ANTI ← [R(V)_{l:t-s:s} ∩ W(V)_{t:u:s}]_{t=l+s:u:s}
  OUTPUT ← [W(V)_{l:t-s:s} ∩ W(V)_{t:u:s}]_{t=l+s:u:s}
end
```

Figure 2: Dependence detection for a loop nest $\mathcal{L}$

This refers to aggregating the write ($\mathbf{W}$) accesses to the array over the prior iterations, then intersecting that region with the aggregated read ($\mathbf{R}$) accesses of X in the later iterations. t can be any iteration in the range from $l+s$ to u. No flow dependence exists if the intersection is empty, otherwise, a flow dependence exists.

We can define an anti dependence in a similar way. In this case, the read comes first, so we denote it as

$$\texttt{ANTI} : [\mathbf{R}(\texttt{X})_{l:t-s:s} \cap \mathbf{W}(\texttt{X})_{t:u:s}]_{t=l+s:u:s}.$$

Similarly, for output dependence, we calculate

$$\texttt{OUTPUT} : [\mathbf{W}(\texttt{X})_{l:t-s:s} \cap \mathbf{W}(\texttt{X})_{t:u:s}]_{t=l+s:u:s}.$$

The compute-dependences algorithm in Figure 2 finds the cross-iteration dependences due to variable V in the loop $\mathcal{L}$. This algorithm first aggregates the read and write accesses to the variable V in $\mathcal{L}$ for the prior and later iterations. It uses these aggregated regions to compute cross-iteration dependences within the loop by intersecting those regions in the ways described above.

Using the results from compute-dependences, the routine detect-loop-dependences determines whether $\mathcal{L}$ can be made parallel or must be serial. eliminate-dependences is used to eliminate anti and output dependences based on the ideas discussed by Cytron and Ferrante [10], for all variables proven to have them. If $\mathcal{L}$ is eventually proven to be free of dependence, then it is identified as parallel.

eliminate-dependences is composed of two parts: *array privatization* and *copy-in/out operations*. Our array privatization algorithm is based on the algorithm developed by Tu [4]. One main difference is that we use the Access Region representation, instead of the triplet notation, to summarize the array access patterns, giving us more opportunities to accurately capture the access patterns. This increased accuracy enables us to utilize copy-in/out operations to eliminate anti and output dependences, as shown in [3].

As an example, consider again the loop in Figure 1. Let the I-loop be a loop $\mathcal{L}$ that we are interested in. To determine the dependences carried by $\mathcal{L}$, notice that the only references which could cause a cross-iteration dependence are to V and Y. The access of Y is trivially independent. For V, we aggregate the read and write regions according to the algorithm. For the first write reference, V(K+1,K+2), we get

$$\begin{aligned}\mathbf{W_1}(\texttt{V})_{1:t-1:1} &= \mathbf{W_1}(\texttt{V})_{t:N:1} \\ &= (\mathcal{A}^{N+1}_{N^2-1} + N + 1, \text{MUST}, \text{WRITE}, \mathbf{T}).\end{aligned}$$

These regions are the same, so their intersection is non-empty, proving a self-output dependence. In contrast, the second write reference, to V(I,I), varies with iterations of $\mathcal{L}$, so we get:

$$\mathbf{W_2}(\texttt{V})_{1:t-1:1} = (\mathcal{A}^{N+1}_{(t-2)(N+1)} + 1, \text{MUST}, \text{WRITE}, \mathbf{T}) \text{ and}$$

$$\mathbf{W_2}(\texttt{V})_{t:N:1} = (\mathcal{A}^{N+1}_{(N-t+1)(N+1)} + t(N+1) - N, \text{MUST}, \text{WRITE}, \mathbf{T}).$$

Intersecting these two regions gives an empty result, proving no self-output dependence exists for this reference. Finally, for the read regions in $\mathcal{L}$, we get:

$$\mathbf{R}(\texttt{V})_{1:t-1:1} = \mathbf{R}(\texttt{V})_{t:N:1} = (\mathcal{A}^{N+1}_{N^2-1} + 2, \text{MUST}, \text{READ}, \mathbf{T}).$$

It is easy to see that no anti or flow dependence exists between the read reference and the first write reference, since the strides are the same, with only the starting point being different. The stride $(N+1)$ does not divide the difference between the starting points $(N-1)$, so no overlap between these regions is possible. The same argument can be applied to a comparison between the read reference and the second write reference, or the first and second write references.

As a result of this process, the only dependence found is the self-output dependence caused by the write due to V(K+1,K+2). To remove this, we can privatize the region of V written in this reference, then copy the *downwards exposed uses* [4] out of the loop to the original array. This allows us to fully parallelize this loop.

The Region Test can parallelize loops with very complex access patterns. It accomplishes this by summarizing accesses at a high level, which removes some of the low-level complexity of the access pattern. For example, the loop nest in Figure 3 has several complications. First, the K-loop is triangular. Also, the subscripting patterns for Y have non-affine subscript expressions and multiple strides (1 and 2^J).

```
do I = 1, M
   do J = 0, M
      do K = 0, 2**J-1
         Y(K+2**J) = ···
      enddo
   enddo
   do J = 0, M-1
      do K = 0, 2**J-1
         ··· = Y(K+2**J)
         ··· = Y(K+2**J+2**M))
      enddo
   enddo
   ···
enddo
```

Figure 3: Code example for dependence elimination

Here, the summarized writes into Y within the I-loop are

$$\mathbf{W}(\texttt{Y})_{\texttt{prior}} = \mathbf{W}(\texttt{Y})_{\texttt{later}} = (\mathcal{A}^{1,2^J}_{2^{J-1},2^N-1} + 1, \cdots).$$

The reads from Y are summarized as the union of read regions

$$\mathcal{A}^{1,2^J}_{2^{J-1},2^{N-1}-1} + 1 \;\cup\; \mathcal{A}^{1,2^J}_{2^{J-1},2^{N-1}-1} + 2^N + 1,$$

resulting in $\mathbf{R}(\texttt{Y})_{\texttt{prior}} = \mathbf{R}(\texttt{Y})_{\texttt{later}} = (\mathcal{A}^{1,2^J}_{2^{J-1},2^N-1} + 1, \cdots)$. The operation coalescing discussed in Section 3.2 simplifies both the read and write regions to the same region,

$\mathcal{A}^1_{2^{N+1}-2} + 1$, which proves that the read region is equal to the write region regardless of the value of t. This implies that there exist anti and output dependences in the I-loop. As discussed earlier, `eliminate-dependences` can remove these dependences by privatizing `Y`. Since `Y` has no cross-iteration flow dependence in the I-loop, it no longer blocks the loop from being parallelized.

None of the current dependence tests in Polaris (GCD Test, Range Test, and Omega Test) could parallelize the I-loop of Figure 3. This is mainly because of the complex exponential subscript expressions of `Y` within the multiply-nested loop. We found that the operations of aggregation and coalescing on the Access Region representations greatly simplify the complex exponential expressions to help the Region Test identify the I-loop as parallel.

The original loop from which the loop in Figure 3 was derived can be found in TFFT2 from the SPEC benchmarks. In fact, this loop is the most important in TFFT2. Therefore, without parallelizing this loop, good speedups are not possible. To measure the effectiveness of the Region Test, we tested the performance of the TFFT2 benchmark on the Cray T3D [20] both with and without application of the Region Test, and the results, shown in Table 1, demonstrate that the Region Test allows us to significantly reduce the parallel execution times.

PEs	No Region Test (sec)	Region Test (sec)	Speedup improvement
4	764	188	4.06
8	634	114	5.56
16	563	69.0	8.16
32	540	47.0	11.5
64	522	36.0	14.5

Table 1: Comparison of parallel execution times

4.3 Interprocedural Dependence Analysis

In order to parallelize a loop containing subroutine calls, Polaris currently inlines the subroutines due to its lack of ability to summarize access patterns in arbitrary code sections within a program. Inlining often increases the program size significantly. For instance, although Polaris can parallelize the most important loops in the fully-inlined TURB3D code from the SPEC benchmarks, full-inlining introduces an increase of the code size from 1400 lines to 6100 lines, along with other complexities that did not exist in the original code. This parallelization comes at a price - the inlined code requires a compilation time of about 20 hours, running on an SGI Challenge system with a 256 MB main memory and 200 MHz MIPS 4400 processors.

Using the Access Region representation, we are able to summarize access patterns in order to avoid inlining for parallelization in a loop with subroutine calls. Figure 4 shows a code example similar to some loops in TURB3D. The array `U` undergoes reshaping as it is passed to the subroutine, but this is not a problem since all array references are linearized by the Access Region representation anyway.

The access descriptor for the region accessed in the subroutine `foo` is $\mathcal{A}^{1,NM}_{N-1,(L-1)NM} + 1$. This same pattern happens regardless of the argument passed to the subroutine, making this a useful summary of the access. Arguments passed to the subroutine supply different starting points for that access pattern. In the Figure, the I-loop containing the call to `foo` produces an additional stride (`N`) for the access pattern, making the write accesses for prior iterations of the outer loop

$$\mathbf{W}(\mathtt{U})_{1:t-1} = (\mathcal{A}^{1,NM,N}_{N-1,(L-1)NM,(t-2)N} + 1, \cdots),$$

```
subroutine simplified
common U(N,M,L)
 ...
do I = 1, M
   call foo(U(1,I,1))
enddo
  ⋮
subroutine foo(X)
common X(*)
 ...
do J = 0, L-1
   do K = 1, N
      X(J*N*M+K) = ...
   enddo
enddo
 ...
return
```

Figure 4: Simplified code example from TURB3D program

and the write accesses for later iterations

$$\mathbf{W}(\mathtt{U})_{t:M} = (\mathcal{A}^{1,NM,N}_{N-1,(L-1)NM,(M-t+1)N} + (t-1)N + 1, \cdots).$$

Since $\mathbf{W}(\mathtt{U})_{1:t-1} \cap \mathbf{W}(\mathtt{U})_{t:M}$ is empty, there is no dependence in the I-loop, thus we make the loop parallel.

Interprocedural access summarization often involves understanding of how variables change with arbitrary control flow in a section of code. When a subscript expression contains a variable whose value cannot be expressed in terms of the loop indices, we must resort to using its symbolic range, obtained by something like interprocedural value propagation [5] (IPVP) and range propagation [7]. These techniques use abstract interpretation on the program to discover the range of values which a given variable may take on.

For example, consider the loop in Figure 5. The variable `PROP` cannot be expressed in terms of any loop variable, yet range propagation can determine that its value has the range `[1:M:1]`. This fact can be used to show that the access descriptor for that loop is $\mathcal{A}^{1,M}_{M-1,(N-1)M} + M + 1 = \mathcal{A}^1_{NM-1} + M + 1$. We depend on this range analysis to help us summarize arbitrary regions of a program.

```
      PROP = M
100   do I = 1, N
         X (I*M+PROP) = ...
      enddo
      PROP = PROP-1
      if (PROP>0) goto 100
```

Figure 5: Summarization using IPVP

4.4 Inter-Task Dependence Analysis

Parallelism need not be confined to loop nests. For traditional point-to-point dependence tests, the dependence problem has been conveniently cast into an equation-solving problem involving loop indices. But when we take the approach of casting the dependence problem into an access region handling problem, we can just as easily summarize access regions in a loop as in an arbitrary code section. This leads naturally to a more general approach to parallelizing programs, one able to take advantage of parallelism wherever it is found in a program.

For task parallelization, we try to determine whether there is any access region of an array, say `V`, overlapped between two arbitrary program sections $\mathcal{C}$ and $\mathcal{D}$. For this purpose, we compute the three types of dependences much as we did in Section 4.2. Figure 6 shows the algorithm that computes the dependence between section $\mathcal{C}$ and section $\mathcal{D}$. In the algorithm, the access regions of all variables used in

```
detect-task-dependences(C,D)
  foreach variable V used in both C and D do
    compute-dependences(V,C,D,FLOW,ANTI,OUTPUT)
    if (FLOW ≠ ∅)
      return 'C and D has dependence'
    if (ANTI ≠ ∅ ∨ OUTPUT ≠ ∅)
      eliminate-dependences(ANTI,OUTPUT,V,L)
    if (ANTI ≠ ∅ ∨ OUTPUT ≠ ∅)
      return 'C and D has dependence'
  endfor
  return 'C and D have NO dependence'
end
compute-dependences(V,C,D,FLOW,ANTI,OUTPUT)
  W(V)_C ← aggregate(C,V,WRITE)
  R(V)_C ← aggregate(C,V,READ)
  W(V)_D ← aggregate(D,V,WRITE)
  R(V)_D ← aggregate(D,V,READ)
  FLOW ← W(V)_C ∩ R(V)_D
  ANTI ← R(V)_C ∩ W(V)_D
  OUTPUT ← W(V)_C ∩ W(V)_D
end
```

Figure 6: Dependence detection between program sections

each section are first summarized, then compared between the program sections. Since each program section usually contains multiple references to a certain variable, this summarization of the access region will help us avoid many of the tests that a point-to-point test would have to make.

```
subroutine corr
common V(N,M),W(N,M)
 ...
call goo(10,S)
call goo(11,S)
 ...
call goo(17,S)
  :
subroutine goo(I,S)
common V(N,M),W(N,M)
 ...
do J = 1, M
   W(I,J)=V(I,J)*S
enddo
 ...
return
```

Figure 7: Simplified code from SU2COR benchmark

```
      subroutine corr
      common V(N,M),W(N,M)
       ...
cdir$ parbegin
cdir$ task 1
      call goo(10,S)
cdir$ task 2
      call goo(11,S)
       ...
cdir$ task 8
      call goo(17,S)
cdir$ parend
        :
      subroutine goo(I,S)
      common V(N,M),W(N,M)
       ...
cdir$ parallel (J)
      do J = 1, M
         W(I,J)=V(I,J)*S
      enddo
       ...
      return
```

Figure 8: Task and loop parallelism in Figure 7

To illustrate task parallelism, consider the code section in Figure 7. Although the J-loop inside goo is parallel, we can still increase parallelism by using task parallelism. Notice that the subroutine goo is invoked consecutively eight times. Using access summarization for the global arrays V and W, we can easily determine that each invocation of goo accesses a disjoint region of the arrays. So, in this case, we divide the processors into eight different subgroups and cause each subgroup to execute an invocation of goo, running the J-loop in parallel. By doing so, we can increase parallelism 8-fold. Figure 8 shows the result after being parallelized and annotated with parallel section directives to direct processors to execute the program in parallel. Here, we use the parbegin/parend construct as suggested by Dijkstra [23].

For task parallelization, we build a *task dependence graph*, similar to that proposed in the paper by Balasundaram and Kennedy [14]. The main difference between our technique and theirs is that we use Access Regions to summarize the array access of a given program section, so we can handle non-unit stride accesses efficiently.

Figure 9 shows the task dependence graph built for the program in Figure 7. Each node i correponds to the i-th call to goo in the program. Following the control flow, each node in the graph is initially connected by a directed edge which represents potential dependence between the nodes. To each pair of nodes connected by an edge, we apply the routine detect-task-dependences. In principle, we need at most $O(p^2)$ applications of the routine for p program sections. If detect-task-dependences finds no dependence between the nodes, it eliminates the edge, which means there are no contraints on the order of excution of the two nodes. As discussed above, since there is no dependence between the calls to goo in Figure 9, all edges will be eliminated from the task graph, resulting in eight parallel tasks.

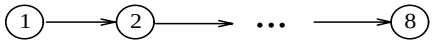

Figure 9: Initial task dependence graph for Figure 7

We have found that task parallelization increases parallelism in many benchmarks, such as HYDRO2D, and OCEAN.

4.5 Conditional Parallelization

Many compilers give up when confronted with unknown values. Run-time parallelization techniques have been proposed [6, 9] for these cases, but until now, these have required the potentially high overhead involved in checking individual array accesses for dependence at run-time. We suggest a different approach, where the conditions, which we call *safe conditions*, can be extracted from the code to test at run-time. Safe conditions are those under which it is safe to parallelize a section of code.

Given a loop with safe conditions, we can produce a *two-version loop*, which will check the safe condition at run time, choosing between a parallel version of the loop and a serial version as follows:

```
if (safe-conditions) then
      parallelized loop
else
      serialized loop
endif
```

The use of predicates of the Access Region inside the Region Processor gives the Region Test a way to extract the safe conditions from the code. The original algorithms described in Figures 2 and 6 can be augmented to include conditional parallelization by simply collecting the predicates from the Access Regions FLOW, ANTI, and OUTPUT. Unlike the original algorithms, the augmented algorithm checks if each non-empty Access Region result has any predicate other than the trivial condition $\mathbf{T}$, and collects all non-trivial conditions as safe conditions. For instance, consider the loop

```
do I = 1,25
   if (P) ··· = X(3*I)
   if (Q) X(2*I) = ···
enddo
```

We have two access regions for X: ($\mathcal{A}^3_{72}$+3,MUST,READ,P) and ($\mathcal{A}^2_{48}$+2,MUST,WRITE,Q). When the two regions are intersected to calculate a cross-iteration flow dependence of the loop as discussed in Section 4.2, the result will be

$$\texttt{FLOW}: (\mathcal{A}^6_{42} + 6, \text{MUST}, *, \texttt{P} \wedge \texttt{Q}),$$

which means that when the predicate P ∧ Q is true, the region of intersection is $\mathcal{A}^6_{42} + 6$. Otherwise, the intersection

is empty. The symbol * represents the access can be both read and write. Taking this result as input, the original routine `detect-loop-dependences` would identify this loop as *serial* since `FLOW` is non-empty. But, the augmented algorithm checks if the predicate is trivially true or false before determining whether this loop is parallel or serial. So, as long as we can prove that `P` $\wedge$ `Q` is false, we can still parallelize the loop even though `FLOW` is non-empty. If the value of the predicate cannot be determined at compile-time, the predicate becomes a safe condition for a run-time test.

Similarly, predicates can be used in subtraction. When subtracting $\mathcal{R}_1 - \mathcal{R}_2$, the result is correct as long as the predicate on $\mathcal{R}_1$ implies the predicate on $\mathcal{R}_2$. Subtraction is involved in array privatization, since we must show that the region written for a variable `V` completely covers the region read in an iteration, so we do the subtraction $\mathbf{R}(\mathtt{V})_t - \mathbf{W}(\mathtt{V})_t$ and check whether the result is empty. For instance, in the loop

```
do I = ···
   if (C>5) then
      do J = 0,M
         X(J) = ···
      enddo
   endif
   if (C>6) then
      do J = 0,M
         ··· = X(J)
      enddo
   endif
enddo
```

the access regions for `X` are $\mathcal{R}_1 = (\mathcal{A}^1_N$,MUST,READ,`C>6`) and $\mathcal{R}_2 = (\mathcal{A}^1_M$,MUST,WRITE,`C>5`). So, when subtracting $\mathcal{R}_1 - \mathcal{R}_2$, the result is $(\mathcal{A}^1_{N-(M+1)} + M + 1$,MUST,*,$\mathbf{T})$ since `C>6` implies `C>5`. This region is empty when the span $(N - (M+1))$ is negative, meaning $N < M + 1$. This gives us a safe condition for the privatization of `X`, which could allow parallelization of the loop.

Another example of making a safe condition involves the following loop

```
do K=L,U
   Y(S*K) = ···
   Y(S*K+1) = ···
enddo
```

The aggregated access region for `Y` is $\mathcal{A}^{1,S}_{1,S(U-L)} + SL$. In order to determine the dependences involved, we must show that the second stride (`S`) is greater than the first span (1), which allows the access to `Y` on each iteration to stride beyond the array elements accessed on the previous iteration. Since `S` is unknown, the Region Processor would generate the condition `S` > 1 as a safe condition.

We have observed that this simple run-time test with predicates is useful for various programs, such as MDG, and OCEAN, from the Perfect benchmarks. Figure 10 shows the multiprocessor speedups of MDG on the Cray T3D. First, we used only the Range Test and other tranditional tests such as GCD test without expensive interprocedural constant propagations [7], but these tests failed to parallelize the loops similar to the ones explained above. Although the remaining serial loops are not important on fewer than 8 processors, they become important on larger number of processors in the spirit of Amdahl's law, which causes the speedup drops on more than 32 processors due mainly to increased communication overhead, as indicated in the figure.

5 Conclusion

We have presented a new dependence test called the Region Test, which is built upon the Access Regions representation.

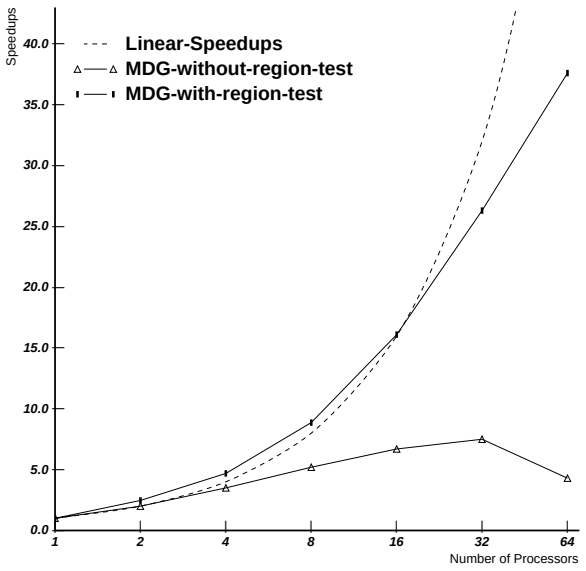

Figure 10: Parallel Speedups of MDG on the T3D

We have shown that it combines the strengths of several existing dependence tests. We have argued that since the Access Region representation uses an abstraction of the program text, it can summarize the effect of arbitrary program sections, and therefore can be used for interprocedural summarization and dependence testing, as well as for loop-based parallelization.

We presented a reformulation of the dependence problem in terms of regions, and showed how a combination of high-level summarization, strong symbolic region manipulation routines and a precise representation can allow the parallelization of program sections which no other existing technique can. We also showed that a by-product of our reliance on symbolic manipulation is the ability to extract the safe conditions for parallelization and to generate run-time dependence tests for some loops from real programs.

References

[1] Y. Paek, J. Hoeflinger, D. Padua, Access Regions: Towards a Powerful Parallelizing Compiler, Tech. Report, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing R & D, 1996, CSRD Report

[2] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, W. Pottenger, L. Rauchwerger, P. Tu, Advanced Program Restructuring for High-Performance Computers with Polaris, *IEEE Computer*, Dec. 1996

[3] Y. Paek, D. Padua, Automatic Parallelization Techniques for Distributed Memory Multiprocessors, Tech. Report, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing R & D, 1996, CSRD Report

[4] P. Tu, Automatic Array Privatization and Demand-Driven Symbolic Analysis, PhD Thesis, University of Illinois at Urbana-Champaign, 1995

[5] W. Blume, Symbolic Analysis techniques for Effective Automatic Parallelization, PhD Thesis, University of Illinois at Urbana-Champaign, 1995

[6] L. Rauchwerger, D. Padua, The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization

and Reduction Parallelization, *Proceedings of ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, Jun. 1995

[7] W. Blume, R. Eigenmann, Demand-driven, Symbolic Range Propagation, *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*, Aug. 1995

[8] S. Chatterjee, J. Gilbert, F. Long, Generating Local Address and Communication Sets for Data-Parallel Programs, *Proceedings of ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, San Diego, May 1993

[9] J. Saitz, R. Mirchandaney, K Crowley, Run-time Parallelization and Scheduling of Loops, IEEE Trans. Computers, Vol 40, May 1991

[10] R. Cytron, J. Ferrante, What's in a Name?, *Proceedings of the 1987 International Conference on Parallel Processing*, Aug. 1987

[11] B. Creusillet, F. Irigoin, Exact vs. Approximate Array Region Analyses, *9th Workshop on Language and Compilers for Parallel Computing*, Lecture Notes in Computer Science, Spring-Verlag, 1996

[12] P. Tang, Exact Side Effects for Interprocedural Dependence Analysis, Communications of the ACM, Vol. 35, No. 8, Aug. 1992

[13] G. Goff, K. Kennedy, C. Tseng, Practical Dependence Testing, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, Ontario, Canada, June 26-28, 1991

[14] V. Balasundaram, K. Kennedy, A Techniques for Summarizing Data Access and its Use in Parallelism Enhancing Transformations, *Proceedings of ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, Jun. 1989

[15] K. Kennedy, K. McKinley, Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution, *Proc. 6th Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, Spring-Verlag, NY, Aug. 1993

[16] W. Pugh, A Practical Algorithm for Exact Array Dependence Analysis, *Communications of the ACM*, Vol. 35, No. 8, Aug. 1992

[17] M. Burke, R. Cytron, Interprocedural Dedependence Analysis and Parallelization, *Proceedings of ACM SGIPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, July 1986, pp 162-175

[18] H. Zima, B. Chapman, Supercompilers for Parallel and Vector Computers, ACM Press, 1992

[19] U. Utpal, Loop Transformations for Restructuring Compilers: The Foundations, Kluwer Academic Publishers, 1993

[20] CRAY T3D System Architecture Overview, Cray Research, 1993

[21] Z. Li, P. Yew, and C. Zhu, Data Dependence Analysis on Multi-dimensional Array References, *Proceedings of the 1989 ACM International Conference on Supercomputing*, Crete, Greece, June, 1989

[22] J. Grout, Inline Expansion for the Polaris Research Compiler, Master's thesis, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., May 1995

[23] E. Dijkstra, Cooperating Sequential Processes, Programming Languages, Academic Press, NY, 1968

[24] M. Berry, et. al, The Perfect Club Benchmarks: Effective Performance Evalution of Supercomputers, *Int'l Journal of Supercomputer Applications*, Vol. 3, No. 3, Fall 1989

[25] M. Wolfe, C. Tseng, The Power Test for Data Dependence, IEEE Transactions on Parallel and Distributed Systems, Sep. 1992

[26] J. Reilly, SPEC95 Products And Benchmarks, *SPEC Newsletter*, Sep. 1995