Orchestrating Interactions Among Parallel Computations*

Susan L. Graham† Steven Lucco† Oliver Sharp†

Computer Science Division, 571 Evans Hall
UC Berkeley, Berkeley CA, 94720

Abstract

Many parallel programs contain multiple sub-computations, each with distinct communication and load balancing requirements. The traditional approach to compiling such programs is to impose a processor synchronization barrier between sub-computations, optimizing each as a separate entity. This paper develops a methodology for managing the interactions among sub-computations, avoiding strict synchronization where concurrent or pipelined relationships are possible.

Our approach to compiling parallel programs has two components: symbolic data access analysis and adaptive runtime support. We summarize the data access behavior of sub-computations (such as loop nests) and split them to expose concurrency and pipelining opportunities. The split transformation has been incorporated into an extended FORTRAN compiler, which outputs a FORTRAN 77 program augmented with calls to library routines written in C and a coarse-grained dataflow graph summarizing the exposed parallelism.

The compiler encodes symbolic information, including loop bounds and communication requirements, for an adaptive runtime system, which uses runtime information to improve the scheduling efficiency of irregular sub-computations. The runtime system incorporates algorithms that allocate processing resources to concurrently executing sub-computations and choose communication granularity. We have demonstrated that these dynamic techniques substantially improve performance on a range of production applications including climate modeling and x-ray tomography, especially when large numbers of processors are available.

*This research was sponsored in part by the Defense Advanced Research Projects Agency under contract DABT63-92-C-0026 and by a Hertz Fellowship to Oliver Sharp. The content of the paper does not necessarily reflect the position or policy of the Government and no official endorsement should be inferred.

†Email: {graham, lucco, oliver}@cs.berkeley.edu

To appear in Proceedings of the ACM SIGPLAN'93 Symposium on Programming Language Design and Implementation, Albuquerque, NM, June 23–25 1993.

1 Introduction

The manual parallelization of large scientific applications is a complex process; a common goal is to encapsulate some of this complexity within automatic tools. Research toward this goal has focused on recognizing parallelism within a particular loop nest and reorganizing loops to improve the use of vector hardware, memory hierarchies, and multiple processors [2, 5, 8, 12, 21].

In this paper, we consider the effect of *interactions* among sub-computations (including loop nests) on the efficiency of parallel program execution. Several application studies [7, 15, 19] have identified such interactions as a key target for manual optimization. Our contribution is to automate these transformations.

For example, suppose we have two loop nests with the potential to execute concurrently. If each computation is handled independently, with synchronization between the two loop nests, efficiency of CPU usage is limited by the less regular of the two. One possible remedy is to use loop fusion, coalescing the two loops into one. However, the resulting parallelization is incomplete, since fusion discards information about the more regular component of the new loop. With the computations separated, a runtime system can use the additional parallelism from the more regular loop nest to smooth the load balance of the computation as a

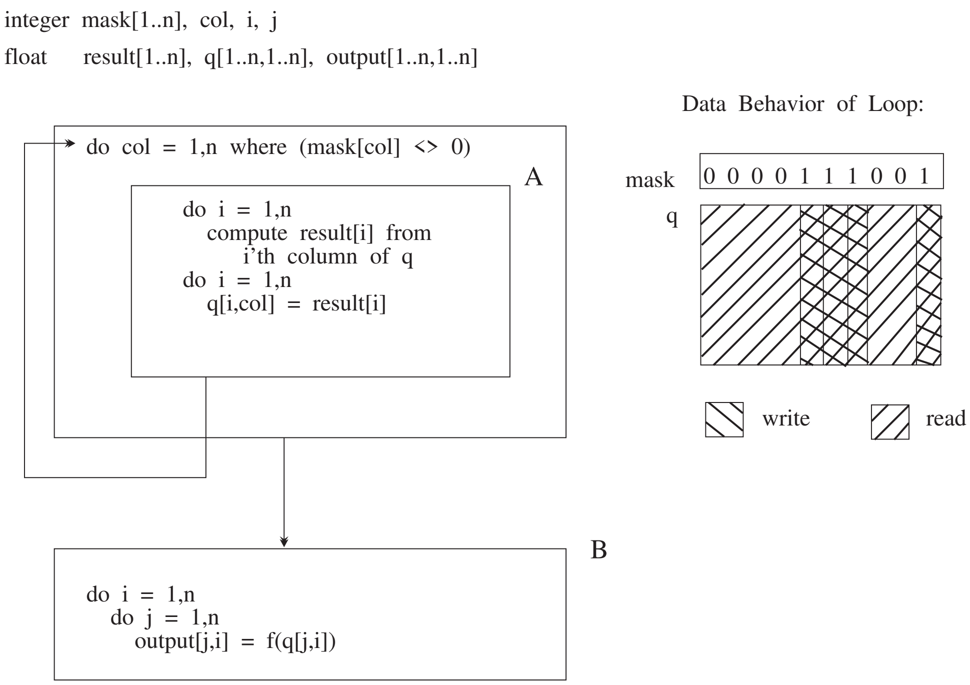

Figure 1: Sample Interacting Computations

whole.

To expose this type of interaction automatically, we use a new method for data access summarization called a *symbolic data descriptor*. We use symbolic data descriptors to implement a key transformation, called *split*, which reduces synchronization constraints by sub-dividing computations. We have incorporated *split* into a compiler which outputs FORTRAN 77 augmented with calls to library routines written in C and a coarse-grained dataflow graph summarizing the exposed parallelism.

The compiler encodes symbolic information such as loop bounds and data sizes. The runtime system uses this information, as well as information gathered from the running program, to guide two key scheduling decisions: grain-size selection and processor allocation. The grain-size selection algorithm is detailed elsewhere [14]. In this paper, we present the runtime processor allocation algorithm, which uses sub-computation finishing time estimates to correctly ration processing resources among concurrently executing sub-computations.

The structure of the paper is as follows: in Section 2, we present a more complete example of sub-computation interaction. Section 3 introduces techniques for building symbolic data descriptors, and presents the details of *split*. Section 4 describes the adaptive scheduling techniques used by the runtime system. Performance results are given in Section 5.

2 Example Interaction

Figure 1 shows interaction among sub-computations. It is a loop whose induction variable `col` iterates over the columns of a data array `q`. If the associated element of the array `mask` is non-zero, the iteration performs the computation A. A reads all of `q` and modifies column `col`. After the loop is completed, B computes the array `output` from `q`.

Because of the conditional in A, the compiler cannot statically determine an efficient schedule for A. Depending on the values in that array, adaptive techniques may find an efficient schedule. However, on large numbers of processors efficiency may still be poor if there is not enough parallelism (i.e. too few mask elements are non-zero) or if the time to process each column varies too widely.

Our approach is to transform the code to expose further concurrency; there are three sources that our strategy can reveal. The first of these is to divide B into three pieces, B_I, B_D, and B_M, where B_I processes the columns of `q` which are not touched by any of the instances of A and B_D processes the rest. B_M merges the results into a single output array. In Figure 2, we show the effect of applying this transformation.

The notation is equivalent to

```
do ...  where <expr>        do ...
   loop body                  if (<expr>)
                                 loop body
```

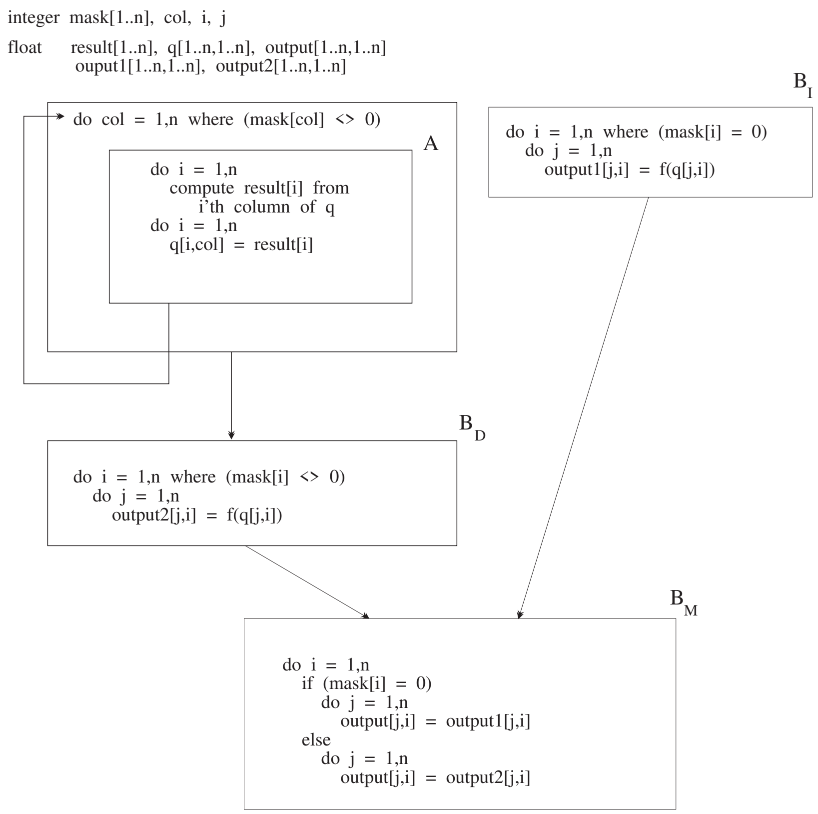

Figure 2: Code After Split

For clarity, we show an explicit merge of the two output arrays in B_M. In practice, merging can often be handled implicitly by the runtime system during data communication.

The second transformation we can apply is to pipeline one iteration of the `col` loop with subsequent iterations. We show the code with this further optimization in Figure 3. The body of the loop has been converted into three computations A_D, A_I, and A_M. A_D represents the code that is dependent on the previous iteration of the loop; the runtime system waits for the previous iteration to complete before scheduling A_D. A_I, on the other hand, is independent of the previous iteration and can be scheduled concurrently. By weakening the synchronization constraint between iterations, we are able to pipeline them.

A_I computes the result vector for all but the missing column of `q` that comes from the previous iteration; A_D computes the one missing element into the variable `prev_val`. A_M takes the almost complete vector and the missing value and combines them into a single vector. Again, this kind of merge can often be done implicitly. Note that we use the form `do var=<range> and <range>` to denote a discontinuous sequence of values, rather than duplicating the entire loop for both ranges.

The third transformation which we could perform is to pipeline iterations of A with corresponding iterations of B_D, exposing a further source of concurrency.

The form of pipelining described in these last two transformations is different from loop pipelining optimizations defined elsewhere. Previous approaches fall into two classes. The first is *software pipelining* [13], which seeks to reduce overhead by reorganizing the code. The transformed loop performs fewer iterations but has a larger body that handles more than one of the original loop's iterations at once. This strategy works well in improving performance of a regular loop nest. The second approach, suggested by Balasundaran and Kennedy [3], uses post and wait primitives to allow more than one loop iteration to execute concurrently. Since this strategy imposes a fixed synchronization discipline, it does not admit adaptive scheduling techniques.

3 Compiler Context

Our compilation environment combines *split* with source-to-source transformations like loop fusion [12] and loop interchange [2] to expose additional concurrency. The system begins by performing an extensive symbolic analysis of the input. The analysis, which is

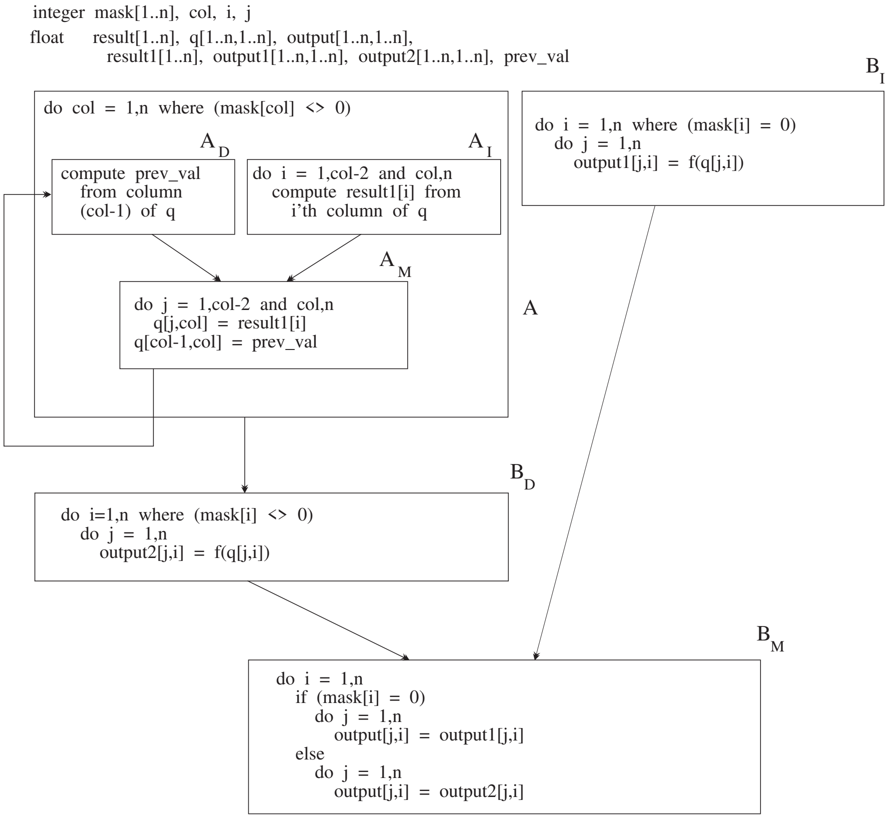

Figure 3: Code After Split and Pipeline

also used to identify independence and improve traditional optimizations like dead code elimination, is crucial for *split*. One particularly useful outcome of *split* is to divide the iterations of a loop into two sets based on memory access behavior, a transformation that rarely can be done without symbolic information. Following the analysis, we calculate summary information and use it to carry out the transformations.

3.1 Symbolic Analysis

The primary purpose of symbolic analysis is to annotate variables with the range of values they can take on during execution. Annotations are determined by converting the code into static single-assignment (SSA) form [6] and propagating branch conditions and assignment information.

Symbolic information comes in a few different forms. An *assertion* is a disjunction of conjunctions of inequalities. In the example, the computation A executes under the assertion that `mask[col] <> 0`. Inequalities express the relationship of an SSA name to an arithmetic *symbolic expression*. We have chosen to limit symbolic expressions to be a sum that may include a set of SSA names, each with an integer coefficient, and a constant (either integer or floating point). A *symbolic value* is either a symbolic expression or a *range*, where a range has a symbolic expression for both starting and ending values and an integer skip.

Analysis consists of several steps:

1. Call Site Analysis – because accuracy is often crucial for the transformations that use symbolic information, the compiler performs aggressive interprocedural analysis. Rather than summarizing a given procedure once and using that summary at every call site, we classify the sites into groups based on profile information and argument characteristics. Call sites that represent a significant amount of computation will only be grouped with others that have the same aliasing pattern and constant values. Less important calls are grouped together less aggressively, based on a tunable heuristic.

2. Memory Usage Analysis – the compiler builds a control flow graph (CFG); each CFG node is annotated with the scalars that it reads and writes and a descriptor for its aggregate usage.

3. SSA – the code is converted into static single assignment form.

4. Aggregate Propagation – the compiler generates temporary SSA names for values that are assigned through aggregates. For example, if a value V is assigned to `A[i]` and then `A[i]` is assigned to a scalar, the compiler creates an SSA name for V.

5. Alias Elimination – when aliases may cause an assignment to overwrite uses of other SSA names, the uses that may be affected are marked invalid. Potential aliases are detected in a top down traversal of the CFG, using the description of each node's memory behavior to determine which SSA names it may invalidate.

6. Value Propagation – the system attempts to convert each conditional branch test into an assertion. If successful, the assertion is propagated through the CFG links controlled by the conditional. In addition, symbolic values are propagated from each def to all use sites that have not been invalidated.

3.2 Symbolic Data Descriptors

Once the code has been annotated with symbolic information, the compiler can summarize memory usage behavior with a *symbolic data descriptor*. Symbolic data descriptors are similar in intent to other summarization strategies that have been proposed, including regular sections [4] and Data Access Descriptors [3]. The key difference is that symbolic data descriptors include symbolic information, which other approaches either limit or discard. Previously, if symbols were incorporated into the summarization, they could only appear as the endpoints of iteration ranges. *Split* manipulates code fragments containing unresolved symbols and requires a summarization strategy that can analyze them.

Descriptors consist of two sets of *triples*, a set representing data locations read and one for data locations written. The read set contains locations which are live on entry to the code being annotated; reads known to be dominated by writes in the write set are not included.

Each triple describes access to a given block of memory and is represented in the form $< G > B[P]$. G is an optional symbolic guard expression; the access represented by the triple is known not to occur if the guard is proven false. B is the memory block accessed. P, also optional, describes the pattern of access; if P is not specified, the triple refers to the entire memory block. Patterns have an expression for each dimension of the memory block, representing the range of data touched. Patterns can optionally include a *masking expression* to further limit access.

When assembling a descriptor for a set of CFG nodes, the analyzer chooses the set of SSA names that may remain unresolved. Here is a sample code fragment representative of irregular scientific code:

```
do i = 1, 10
  if (miss(i) <> 1) then
    do j = 1, 10
      q[i,j] = q[i,j] + x[j]
```

To determine whether loop iterations are independent, we assemble the descriptor for an individual iteration. The iterations are independent if a change to the value of the induction variable yields a new descriptor that intersects with the original only in their read sets. To describe the behavior of the entire loop, we specify that the induction variable may not appear in the descriptor so it is *promoted* to be its entire range. Ignoring scalar variables, here is the descriptor for the i'th iteration of the entire loop nest in the example:

write: $< miss[i] \neq 1 > q[i, 1..10]$
read: $< miss[i] \neq 1 > q[i, 1..10]$
$x[1..10]$

The expression within angle brackets is a guard and the variable i is an unresolved constant within the iteration. When the descriptor for the entire loop nest is computed, the guard is converted into a mask across the first dimension of `q`:

write: $q[1..10/(miss[*] \neq 1), 1..10]$
read: $q[1..10/(miss[*] \neq 1), 1..10]$
$x[1..10]$

The expression following the slash is a *mask* on the range accessed in that dimension of the array. The symbol $*$ in $miss[*]$ is a pattern matching symbol that represents the current element in a range. Accesses to $q[5, 1..10]$, for example, are masked by the element $miss[5]$.

Given a descriptor D, we will refer to its read set as D_{read} and its write set as D_{write}. A descriptor A *interferes* with a descriptor B if:

$$(A_{write} \cap B_{write} \neq \emptyset) \text{ or}$$
$$(A_{write} \cap B_{read} \neq \emptyset) \text{ or}$$
$$(A_{read} \cap B_{write} \neq \emptyset)$$

Interference captures the notion of dependency between descriptors. The three cases represent output-, flow-, and anti-dependencies respectively. When two descriptors do not interfere, the computations they represent are independent. We compute interference conservatively; descriptors interfere unless we can prove otherwise. To simplify discussion, we will say that two computations do not interfere if their descriptors do not.

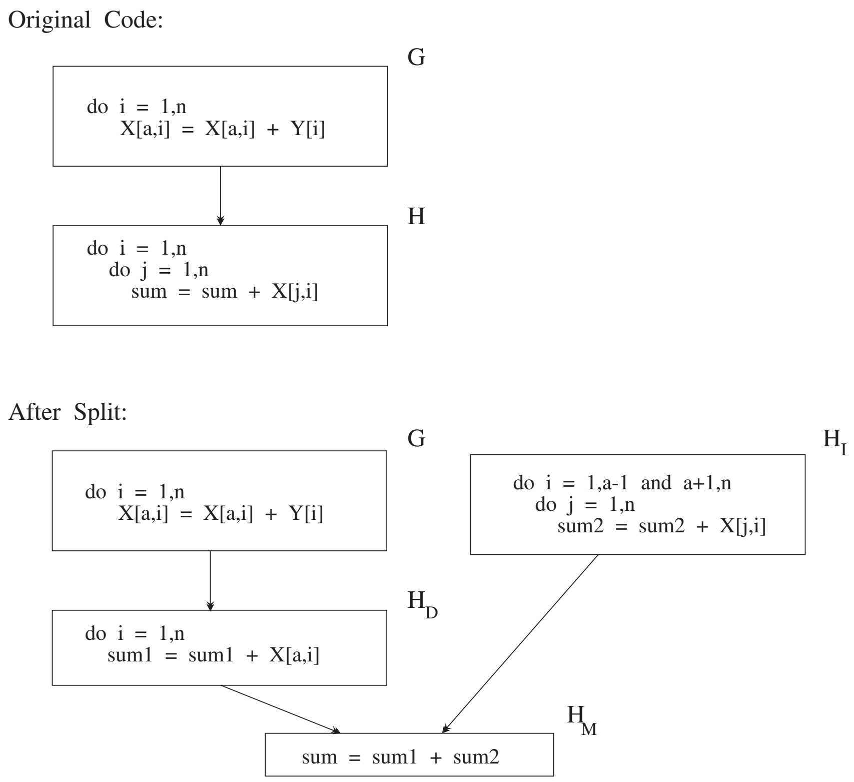

Figure 4: Simple Example of Split

3.3 The Transformations

3.3.1 *Split*

The *split* transformation is used when two computations interfere. Figure 4 demonstrates a simple example, the division of H into three parts (assuming that we can treat addition as an associative operation). Let DG be the descriptor for G and DH be the descriptor for H. Then (omitting induction variables) $DG_{write} =$ { X[a,1..n]}, $DG_{read} =$ { X[a,1..n],Y[1..n]}, DH_{write} = { sum}, and $DH_{read} =$ {X[1..n,1..n], sum}. Since $DG_{write} \cap DH_{read} \neq \emptyset$, the two code blocks interfere.

H is flow dependent on G because part of H reads the column of `X` modified by G. However, the dependency hides potential concurrency because most of H can be computed simultaneously with G; only the computation on column `a` must be deferred until G is done (we assume for simplicity that `a` is known to be between 1 and n). H can be divided into the piece that must wait for G and the piece that need not; after both parts have finished, the results are merged together. This is exactly what *split* does on this example, dividing one computation into three parts based on its interactions with a second computation.

More generally, *split* takes as input a computation C and a descriptor D of a computation (usually a predecessor or successor), and attempts to find sub-computations in C that do not interfere with D. The effect of the transformation is to convert C into three computations: the dependent sub-computation C_D, the independent sub-computation C_I, and the merging sub-computation C_M. C_I contains the sub-computations that do not interfere with the computation whose descriptor is D and could be separated. C_D holds the rest of C, except for those sub-computations that rely on values now computed in C_I. The remaining sub-computations, along with any needed post-processing code, are put into C_M.

The split algorithm begins by subdividing C into *primitive computations*. Primitive computations are the blocks of code that are managed by the transformation; the choice of primitive computation determines the granularity of the split. We have chosen to consider basic blocks, function calls, and loops as primitive computations.[1] Thus in Figure 4, code block G has two primitive computations, a loop and a basic block, whereas block H has three. Subdividing basic blocks would yield a more aggressive split, at the cost of more communication overhead and more cleanup code in C_M.

The next step is to categorize each primitive com-

[1] When profiling information indicates that a block of code is executed infrequently, the system may choose not to decompose a loop nest into its constituent computations.

putation into one of three sets based on its memory usage behavior. In defining the sets, we will say that a computation C *(directly) interferes* with a set S if and only if there exists $s \in S$ such that C interferes with s. We will also use the property of *transitive interference*; C transitively interferes with S if there exists a sequence of computations I_1, I_2, ... I_n, $n \geq 0$, such that C interferes with I_1, I_m interferes with I_{m+1}, and I_n interferes with S. We say C transitively interferes with S_1 *using* S_2 when I_1 through I_n are members of S_2. Notice that the property holds when C directly interferes with S.

The three categories of memory usage are:

- $Bound$ – computations that interfere with D.
- $Linked$ – computations that do not interfere with D directly but do transitively interfere with it.
- $Free$ – computations that do not interfere with D directly or transitively.

The following algorithm takes C, a collection of primitive computations, and assigns each member to a usage category with respect to a descriptor D:

$Bound = MaybeFree = \emptyset$
for each $c \in C$
    if $interfere(c, D)$
        $Bound = Bound \cup \{c\}$
    else
        $MaybeFree = MaybeFree \cup \{c\}$
$Linked$ = transitive_interfere($MaybeFree, Bound$)
$Free = MaybeFree$

transitive_interfere($Initial, Target$)
    $Result = \emptyset$, $Testset = Target$
    while ($Testset$)
        $Newbound = \emptyset$
        for each $c \in Initial$
            if $interfere(c, Testset)$
                $Initial = Initial - \{c\}$
                $Result = Result \cup \{c\}$
                $Newbound = Newbound \cup \{c\}$
        $Testset = Newbound$
    return $Result$

The $transitive\_interfere$ procedure is given an initial set $Initial$ and a reference set; it returns a set $Result$ containing the members of $Initial$ that transitively interfere with $Target$ using $Initial$. The members of $Result$ are removed from $Initial$. The procedure iterates to fixpoint; the number of iterations depends on the interference pattern between sub-computations. Each iteration looks at every sub-computation that remains in the $Initial$ set and moves at least one into $Result$ or the algorithm terminates. An upper bound on execution time is $O(n^2)$ $interfere$ or $union$ computations, where n is the number of sub-computations in C. In general, n is small because a computation that is worth moving will be fairly coarse grained.

Applying the algorithm to the computation H and the descriptor DG, all three primitive computations are in $Bound$, reflecting the fact that $DG_{write} \cap DH_{read} \neq \emptyset$.

As in this case, it is often possible to split the iterations of a loop in $Bound$ into two sets, one of which interferes with D and one of which does not. It is legal to split iterations when we have nests of loops that are either independent or computing a reduction; they can be split by placing a conditional on the induction variable. By looking at the intersection of the loop descriptor with D, we determine whether a restriction on the induction variable yields a set of iterations that do not interfere with D.

If a loop is successfully split, we have two sets of loop iterations; one is still in $Bound$, but the other is either in $Free$ or in $Linked$, depending on whether it interferes transitively with D. In our example, by restricting `i` so that it never equals `a`, we are left with a $Free$ computation. Note that we are computing a reduction, so to split the iterations we must replicate the reduction variable and do the final reduction step in H_M.

A simple assignment of sub-computations to output sets could be done based on the three categories, by putting $Free$ computations into C_I and all the rest into C_D. However, it is often important to handle $Linked$ computations more carefully, so the set is subdivided further. This involves one final interference property called *flow interference*, which unlike interference is not symmetric. A successor computation B has a flow interference from a predecessor computation A if $A_{write} \cap B_{read} \neq \emptyset$. This is equivalent to the notion of a *flow* dependency. Transitive flow interference is analogous to transitive interference. The new sub-divisions are:

- $NeedsBound$ – $Linked$ computations with a transitive flow interference from $Bound$.
- $GenerateLinked$ – $Linked$ computations from which $Bound$ or $NeedsBound$ has a transitive flow interference.
- $ReadLinked$ – $Linked$ computations which are neither $NeedsBound$ or $GenerateLinked$.

In Figure 5 we show a more complicated example to demonstrate the types of $Linked$ computations (but do not show transformed code). Suppose we wish to split T with respect to W's descriptor. For this example, we will consider the named computations indivisible.

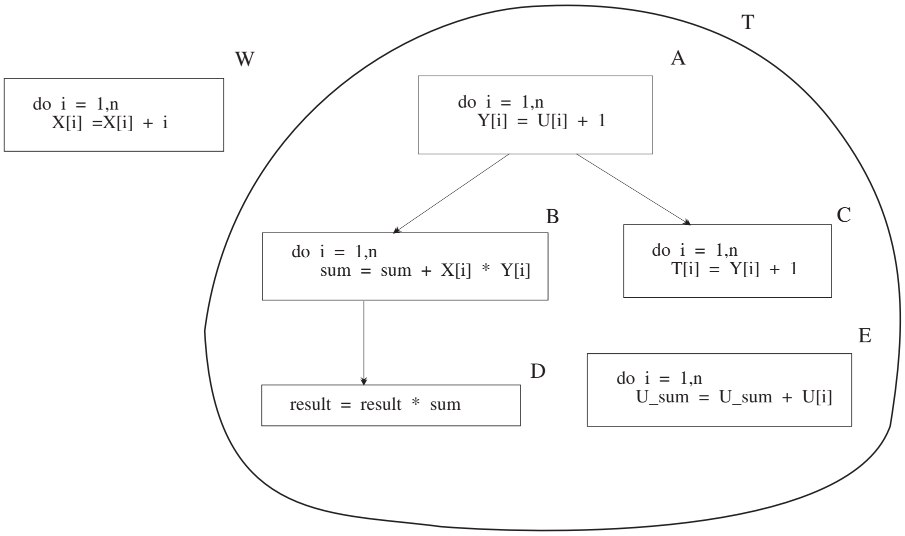

Figure 5: Enhanced Example of Split

E is $Free$ because it does not have any relationship to W; B is *Bound* because it reads the values of array `X` that are written by W. The rest of the sub-computations are *Linked*. A *generates* values of `Y` that are used by the *Bound* computation B and hence is *GenerateLinked*. C is *ReadLinked*, because it also needs values from A. D is $NeedsBound$ because it uses the value of `sum` computed by the *Bound* computation B.

Here is the algorithm that assigns each *Linked* computation to a set:

$Unrestricted = Linked$
$NeedsBound =$ transitive_flow_up($Unrestricted$, $Bound$)
$GenerateLinked =$ transitive_flow_down($Unrestricted$, $Bound \cup NeedsBound$)
$ReadLinked = Unrestricted$

The computation of *transitive_flow_*$\{up, down\}$ is analogous to $transitive\_interfere$, except that the calls to $interfere$ are replaced by calls to $flow\_interfere$. $flow\_interfere$ is not symmetric, so there are two versions of the transitive routine to check for interference from the candidate set (*down*) and from the target set (*up*).

ReadLinked computations can be moved into the independent set at the cost of additional replication or movement. From the example, we could move the *ReadLinked* computation C into T_I if we are willing to replicate A or to move both B and D into T_M. In general, to move a *ReadLinked* computation r into the independent set, every computation s from which r has a transitive flow interference must also be put in that set.

In our current implementation, we use a heuristic to decide whether moving a member of *ReadLinked* is worthwhile. The heuristic goes ahead with the move if both of the following are true:

- the number of floating point and integer computations in the code that is to be replicated can be calculated and it is below a threshold
- profiling data shows that the computation is expensive enough to justify moving it

Note that when a sub-computation created by splitting a loop nest is moved, some code must be added to the merge. If a nesting of independent loops is split, the merge glues the results together. If a reduction is involved, a temporary scalar or array must be created and the results stored in it (as was done in Figure 4 for the reduction variable `sum`). As a final step in merging, the last reduction is performed.

3.3.2 Applications of *Split*

In the discussion of the example in Section 2, three sources of concurrency are described, the first of which is a simple split. The other two, however, are pipelining transformations that go beyond *split*.

To pipeline a loop with *split*, first the descriptor for one iteration of the loop is computed. If the induction variable is i, D_{i-1}, the descriptor for iteration $i-1$, is computed. Then the loop body is *split* using D_{i-1}; the resulting independent computation does not interfere with iteration $i{-}1$. As iteration i is computed, the next iteration's independent computation can be executed concurrently. By transforming the code, that opportunity is exposed to the runtime scheduler. If deeper pipelining is desired, the descriptor for iteration $i-2$ can be computed, etc.

3.4 The Intermediate Form

The compiler outputs the transformed program in three forms. The first is a dataflow graph representing the parallel control structure. The graph is expressed in the coordination language Delirium, a functional language with special support for describing data parallel operations [16].

The second form of output is a series of parallel and sequential sections in the original source language. The sequential computations can invoke Delirium functions representing parallel computations via the re-entrant runtime system. Dataflow within each parallel section is described in Delirium.

The final form of output is a set of annotations on each argument and return value of the Delirium functions, giving data size and type information. The Delirium compiler translates this information into runtime code for estimating communication costs.

4 Runtime Support

One consequence of using Delirium as an intermediate form is that the source language compiler fixes the minimum grain size of the computation. The set of non-re-entrant operators determines the minimum units of scheduling. Henceforth, we'll call these indivisible scheduling units *tasks*.

While the minimum grain size of the computation is fixed by the front end analysis tool, the Delirium compiler and runtime system retain crucial flexibility in determining the grain size at which a data parallel operation is scheduled. In a previous paper [14], we developed a quantitative relationship between total available parallelism, optimal grain size, and execution time variance of a parallel operation. We applied this relationship to the problem of choosing grain sizes for parallel operations, and demonstrated that adaptive scheduling is often both necessary and sufficient for efficient execution of irregular parallel operations.

In this section and the next, we demonstrate that interactions among parallel operations, both regular and irregular, can also benefit from adaptive strategies.

4.1 Orchestrating Interactions Among Parallel Operations

Consider a runtime scenario in which the transformed parallel operations A and B_I from Figure 3 are executing simultaneously. A begins executing first and has partially completed when B_I begins executing. The runtime system must decide how many processors to reallocate. If we change the example so that all processors must synchronize upon completion of A and B_I, an ideal processor allocation would minimize the expected finishing time of these two parallel operations.

The expected finishing time of a parallel operation is a function of the number of tasks that make up the operation (N), the number of processors cooperating to execute the operation (p), the variance in task execution times, and the overhead of scheduling [11, 14]. Thus, approximating the ideal processor allocation requires information available only at runtime.

For this reason, we extended adaptive algorithms developed for single irregular parallel operations to manage interactions among multiple, simultaneously executing parallel operations. There are three main extensions. First, we developed a method for improving the accuracy of estimated finishing times that works for a wide range of scheduling algorithms. Second, we applied this method to the runtime processor allocation problem, using an iterative algorithm to equalize finishing time estimates. Finally, we combined finishing time estimates with runtime communication cost estimates to choose communication granularity for pairs of pipelined parallel operations.

4.1.1 Individual Parallel Operations

To provide a basis for describing these extensions, we review in this section our adaptive runtime algorithms for executing single parallel operations on distributed memory machines.[2]

We use a probabilistic algorithm called TAPER to select the grain-sizes at which tasks are scheduled. The runtime system samples task execution times to compute their statistical mean (μ) and variance (σ^2). It uses this information to reduce overhead by scheduling large chunks (groups of tasks) at the beginning of a parallel operation and successively smaller chunks as the computation proceeds. If we define a scheduling event as the moment when a processor finishes executing a chunk, then for each scheduling event i, TAPER computes K_i, the number of tasks in the i^{th} chunk.

The runtime system does additional sampling of task costs to build a *cost function*, which estimates task execution times as a function of iteration number within the parallel operation. We use the cost function to scale a chunk size K_i by $s = \mu_g/\mu_c$. In this expression, μ_g is the global mean execution time and μ_c is the mean for the tasks in the current chunk.

Regular parallel operations execute according to the "owner-computes" rule described in [9]. To execute irregular parallel operations, we begin with some original data decomposition and assign tasks to processors

[2] The shared memory algorithm is described in [14]

according to the owner-computes rule. During execution, as the runtime system gains information about the work distribution, it refines the data decomposition.

In the distributed TAPER algorithm the p processors are logically connected as a binary tree with p leaves. Some of the processors act as both leaves and internal nodes of the tree. All processors start in *epoch* 0. When a processor begins executing a chunk it sends its current epoch value (called a *token*) to its parent, which passes the token to its parent (possibly combining messages from both children). When the root receives p tokens from the same epoch, it increments the global epoch value and broadcasts a message through the tree to all processors. The message tells the processors to increment their epoch value and may also tell some processors to transfer a chunk of tasks and their associated data to another processor.

Processors compete for the p chunks of each epoch. If processor a can get two tokens of value i to the root before processor b can send one token of value i, then the root will re-assign processor b's chunk of size K_i to processor a. Processor b is then forced to re-interpret the chunk it is currently executing as belonging to some later epoch (and thus containing fewer tasks). If most of the actual task cost is on a few processors, this scheme will degenerate into the centralized TAPER algorithm. If task costs are independent then we expect most tasks to remain on the processor owning them at the beginning of the parallel operation; thus, the algorithm reduces task transfer costs and maintains communication locality.

4.1.2 Interacting Parallel Operations

Now, we return to our runtime scenario in which the parallel operations A and B_I are executing concurrently. When B_I begins executing, the runtime system must reallocate some of the p processors executing A. To accomplish this, the runtime system uses the following iterative algorithm:

$epsilon = 5\%$
$p1 = p/2,\ p2 = p - p1,\ count = 0$
$eA = \text{finish\_estimate}(A, p1),\ eB = \text{finish\_estimate}(B, p2)$
while $((count < max\_count)$ and $(|eA - eB| > epsilon))$
    if $(eA > eB)$
        $p1 = p1 + p2/2$
        $p2 = p - p1$
    else
        $p2 = p2 + p1/2$
        $p1 = p - p2$
    $eA = \text{finish\_estimate}(A, p1)$
    $eB = \text{finish\_estimate}(B, p2)$
    $count = count + 1$

We limit the number of iterations to control the amount of overhead imposed. In practice, using a max_count of four has been sufficient.

In estimating finishing time, the runtime system uses the expression:

$$finish = setup + compute + lag + comm + sched \quad (1)$$

$setup$ is the maximum of the time to contract the data required by A onto p_1 processors and the time to expand the data required by B_I onto p_2 processors. $compute$ is the expected mean time to perform a portion of the computation: $N\mu/p'$ (where $p' = p_1$ for A and $p' = p_2$ for B_I). lag is the expected maximum finishing time to perform a portion of the computation; it is related to the distribution of task execution times for the computation (μ,σ) [14]. $comm$ represents the communication overhead of executing the given parallel operation on p' processors.

To estimate $comm$ at runtime, we use an algorithm like that suggested by Sarkar and Hennessy [18], which performs a weighted sum of dataflow graph edges that cross processor boundaries. Rather than perform this computation statically, the Delirium compiler generates code blocks that perform the estimate given runtime parameters such as N and p'.

Finally, we must estimate the scheduling overhead ($sched$). To do so, we need to predict, at runtime, the number of chunks that will be scheduled for the parallel operation (hence the number of epochs in the distributed algorithm given above). The method for predicting this parameter is discussed elsewhere [14]; it applies to TAPER and many other algorithms for generating chunk sizes [10, 17, 20].

By balancing the estimated finishing times of A and B_I, the runtime system uses the extra concurrency from B_I to compensate for A's irregular execution behavior.

5 Performance Results

We validated the runtime algorithms using a set of applications including the EMU circuit simulator [1], the UCLA General Circulation Model, an adaptive vortex method for modeling turbulent fluid flow, and an image reconstruction program for x-ray tomography (Psirrfan). More detail on these applications is given elsewhere [14].

We hand-transformed each program with *split* [3]. The result was a Delirium description of the exposed parallelism and a transformed FORTRAN program (or

[3] At the time we took the measurements, the integration between the run time system and compiler had not been finished

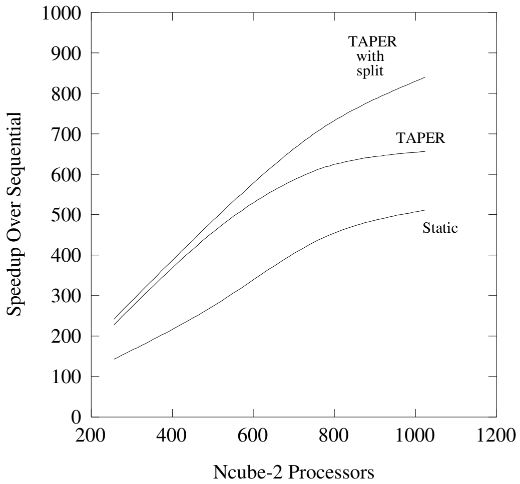

Figure 6: Psirrfan performance

C program in the case of EMU). The Delirium compilation was done automatically. With the runtime system using the processor allocation algorithm described above, we were able to double the number of processors used for each application, with a loss of only five to fifteen percent in efficiency.

For example, using just the TAPER algorithm with cost functions, we could run the UCLA climate model on 512 processors of an Ncube-2 multiprocessor at 87% efficiency, where efficiency is defined as the performance given the 512 processors divided by the sequential performance of the program. When we modified the climate model using *split* wherever applicable, we were able to run the same input data set (about 3200 latitude-longitude grid cells) at 83% efficiency on 1024 processors. Hence the total speedup increased from 445 to 850. Without this modification, the climate model's speedup on 1024 processors is only 581 (57% efficiency) because of the irregular task execution times found in the cloud physics section of the code.

As shown in Figure 6, Psirrfan with just the TAPER algorithm and cost functions is highly efficient on 512 processors but does not sustain this efficiency (with same input size) through 1024 processors. However, by exposing additional coarse-grained parallelism and two opportunities for pipelining, we transformed Psirrfan to achieve sustained efficiency of over 80% using up to 1024 processors.

6 Conclusion

Many parallel applications contain multiple, concurrently executing sub-computations with distinct execution behaviors. To achieve efficient execution of such programs on large numbers of processors requires a new approach to orchestrating the interactions among program sub-computations. When two sub-computations are allowed to execute concurrently, a runtime scheduler can use the additional parallelism of one sub-computation to compensate for communication constraints or load imbalance in the other.

To support this type of optimization, we have developed symbolic data summarization techniques and used these techniques to implement the *split* transformation, which can identify and expose concurrent and pipelined interactions among sub-computations. We have also developed runtime algorithms which take advantage of this transformation by adaptively assigning processing resources to concurrently executing sub-computations. We have demonstrated the benefits of the transformation on a set of application programs including x-ray tomography, circuit simulation, and climate modeling.

7 Acknowledgements

We gratefully thank David Bacon for his helpful comments on an earlier draft.

References

[1] Bryan Ackland, Steven Lucco, Tom London, and Eric DeBenedictis. "CEMU: A Parallel Circuit Simulator,". In *Proceedings of the International Conference on Computer Design*, October 1986.

[2] John R. Allen and Ken Kennedy. "Automatic Loop Interchange,". In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 233–246, Montreal, Canada, June 1984.

[3] Vasanth Balasundaram. "A Mechanism for Keeping Useful Internal Information in Parallel Programming

Tools: The Data Access Descriptor,". *Journal of Parallel and Distributed Computing*, 9:154–170, 1990.

[4] David Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Rice University, April 1987.

[5] Ron Cytron and Jeanne Ferrante. "What's in a Name? -or- The Value of Renaming for Parallelism Detection and Storage Allocation,". In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 19–27, St. Charles, Illinois, August 1987.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kennth Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph,". *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[7] Geoffrey C. Fox. "Domain Decomposition in Distributed and Shared Memory Environments,". In *Proceedings of the First International Conference on Supercomputing*, pages 1042–1073, June 1987.

[8] Milind Girkar and Constantine Polychronopoulos. "Compiling Issues for Supercomputers,". In *Supercomputing '88*, pages 164–173, Orlando, Florida, November 1988.

[9] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. "Compiler Optimizations for Fortran D on MIMD Distributed Memory Machines,". In *Proceedings of Supercomputing '91*, pages 86–100, November 1991.

[10] Susan Hummel, Edith Schonberg, and Lawrence Flynn. "Factoring: A Practical and Robust Method for Scheduling Parallel Loops,". In *Proceedings of Supercomputing '91*, pages 610–619, November 1991.

[11] C. Kruskal and A. Weiss. "Allocating Independent Subtasks on Parallel Processors,". *IEEE Transactions on Software Engineering*, SE-11, October 1985.

[12] David J. Kuck, Robert H. Kuhn, David Padua, Bruce Leasure, and Michael Wolfe. "Dependence Graphs and Compiler Optimizations,". In *Eighth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 207–218, Williamsburg, Virginia, January 1981.

[13] Monica Lam. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines,". In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, pages 318–328, Atlanta, GA, June 1988.

[14] Steven Lucco. "A Dynamic Scheduling Method for Irregular Parallel Programs,". In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI)*, San Francisco, CA, June 1992.

[15] Steven Lucco and Oliver Sharp. "Delirium: An Embedding Coordination Language,". In *Supercomputing '90*, pages 515–524, New York City, New York, November 1990.

[16] Steven Lucco and Oliver Sharp. "Parallel Programming With Coordination Structures,". In *Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, Orlando, Florida, January 1991.

[17] Constantine D. Polychronopoulis and David Kuck. "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers,". *IEEE Transactions on Computers*, C-36(12), December 1987.

[18] Vivek Sarkar and John Hennessey. "Partitioning Parallel Programs for Macro Dataflow,". In *ACM Conference on Lisp and Functional Programming*, pages 202–211, Cambridge, Mass., 1986.

[19] Jaswinder P. Singh, C. Holt, Takashi Totsuka, Anood Gupta, and John L. Hennessy. "Load Balancing and Data Locality in Hierarchical N-body Methods,". Technical Report CSL-TR-92-505, Stanford University, February 1992.

[20] Peiyi Tang and Pen-Chung Yew. "Processor Self-Scheduling for Multiple Nested Parallel Loops,". *Proceedings of the 1986 International Conference on Parallel Processing (ICPP)*, pages 528–535, August 1986.

[21] Michael E. Wolf and Monica S. Lam. "An Algorithmic Approach to Compound Loop Transformation,". In *Proceedings of the 3rd Workshop on Programming Languages and Compilers for Parallel Computing*, August 1990.