THEORY, TECHNIQUES, AND EXPERIMENTS IN SOLVING RECURRENCES IN COMPUTER PROGRAMS

BY

WILLIAM MORTON POTTENGER

M.S., University of Illinois at Urbana-Champaign, 1995
B.S., University of Alaska, Fairbanks, 1989
B.A., Lehigh University, 1980

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1997

Urbana, Illinois

THEORY, TECHNIQUES, AND EXPERIMENTS
IN SOLVING RECURRENCES IN COMPUTER PROGRAMS

William Morton Pottenger, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1997
David A. Padua, Advisor

The study of theoretical and practical issues in automatic parallelization across application and language boundaries is an appropriate and timely task. In this work, we discuss theory and techniques that we have determined useful in solving recurrences in computer programs. In chapter two we present a framework for understanding parallelism in computer applications based on an approach which models loop bodies as *coalescing loop operators*. In chapter three, we perform a case study of a modern C^{++} semantic retrieval application drawn from the digital library field based on the model presented in chapter two. In the fourth chapter we present a summary of several techniques that we believe can be applied in the automatic recognition and solution of recurrences. The techniques have been developed through performing a manual analysis of applications from benchmark suites which include sparse, irregular, and regular Fortran codes. In chapter five we discuss the application of the techniques developed in chapter four on a suite of Fortran codes representative of sparse and irregular computations which we have developed as part of this work. In the sixth chapter, we consider the application of these same techniques focused on obtaining parallelism in outer time-stepping loops. In the final chapter, we draw this work to a conclusion and discuss future directions in parallelizing compiler technology.

ACKNOWLEDGMENTS

There is a person without whom my life would not be possible. Over the years, this person has steadily guided my wife, myself, and our children. Even now, as we near the end of this seven year period of study, He is nearer than before. With grateful hearts and thankfulness to Him, we dedicate this work to our Lord and Savior, Jesus Christ.

I would be remiss not to mention the faithfulness with which my wife and sister in the Lord, Byung Hee Leem, has supported me throughout the 10 years of marriage we have had together. She has been an excellent wife and mother, far beyond what other women do. What she wrote about others in her thesis several years ago has become her own testimony: her vision, perseverance, and long patience will now be rewarded.

In addition, I would like to thank my parents, John L. and Tavia M. Pottenger, for their long-suffering prayers and patience throughout the years.

I must also thank my thesis advisor, Dr. David Padua, for his patience, kindness, and sincere compassion in helping us to reach this goal. Thank you, Professor Padua.

Lastly, I would like to specifically thank Mike Pflugmacher and Nancy Rudins as well as the other hardworking members of the NCSA Advanced Computing and Technology Management Groups.

There are many who remain unmentioned here. To all of you, thank you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

At a recent seminar held on the campus of the University of Illinois, Director Richard Wirt of the Intel Microprocessor Research Lab discussed future directions in microprocessors, multiprocessors, compilers, and applications. As the discussion turned to the automatic parallelization of computer programs, Dr. Wirt emphasized the following point:

The compiler IS the architecture [67]

According to Dr. Wirt, automatic parallelization will play a crucial role in compilers for both existing and upcoming Intel-based multiprocessors. Echoed by Dr. Sam Fuller, Chief Scientist at Digital Equipment Corporation, this fact underscores a trend in multi-process computing which is expected to continue into the 21st century [20].

In recent years, research in the area of automatic parallelization has focused on numeric programs written in Fortran. However, new application areas which employ object-oriented programming models are evolving, and general methods which are applicable across language and application boundaries must be developed.

For example, five years ago the National Research Council Computer Science and Telecommunications Board (CSTB) published a landmark report which examined how Computer Science and Engineering teaching and research are conducted on a national scale. The report emphasized the need for research in the development and construction of application-oriented software systems for parallel computers in an interdisciplinary environment. Examples of such National Challenge applications of high-performance computing and communications technologies include many data-intensive applications such as the processing of financial transactions, improved access to government information, and digital libraries.

The study of theoretical and practical issues in automatic parallelization across application and language boundaries is thus an appropriate and timely task. In this work, we discuss theory and techniques that we have found useful in recognizing and understanding parallelism in applications expressed in various languages across several fields of science.

In the second chapter of this thesis, we present a framework for understanding parallelism in computer applications based on an approach which models loop bodies as associative *coalescing loop operators*. Although associative operations have been the basis for parallelization in both hardware and software systems for many years, to our knowledge no one has developed a framework for understanding loop-level parallelism based on treating the body of the loop as a single associative coalescing operator. The application of this model has yielded useful results: we now understand how to automatically parallelize loops with dynamic last values, and we have also demonstrated that a class of loops with multiple exits can be successfully parallelized within this framework.

In chapter three, we perform a case study of a modern C^{++} semantic retrieval application drawn from the digital library field. We analyze the application based on the model presented in chapter two, and characterize the performance of the resulting parallelized version.

In the fourth chapter we present a summary of several techniques that we believe can be applied in the automatic recognition of parallelism. The techniques have been developed through performing a manual analysis of applications from benchmark suites which include sparse, irregular, and numeric Fortran codes. Historically, sparse and irregular codes have been considered difficult to parallelize due to the complex nature of dependence relationships. The results of our study have revealed, however, that several of these techniques have a significant impact on performance across a range of application areas.

In chapter five we introduce a suite of Fortran codes representative of sparse and irregular computations which we have developed as part of this work. We then discuss the application of the techniques presented in chapters two and four to the codes in this suite, and characterize the performance of the parallelized versions.

In the sixth chapter, we consider the application of the techniques from chapter four that we have determined useful in the automatic parallelization of outer, time-stepping loops containing recurrences. Two applications from the SPEC CFP95 benchmark suite are analyzed and manually parallelized based on these techniques, and the performance of the resulting parallel versions is characterized.

In the final chapter we draw this work to a conclusion and discuss future directions in parallelizing compiler technology.

CHAPTER 2

Coalescing Loop Operators

2.1 Introduction

In the course of investigating solutions to recurrences in loops the desire to develop a recurrence recognition scheme based on technology more general than pattern-matching arose. This in turn led to an investigation of the principle property determining the parallelizability of a given operation.

Consider, for example, a loop of the following nature[1]:

```
do 400 $j = search, neqns$
        $node = perm(j)$
        if $(marker(node).lt.0)$ goto 400
        $ndeg = deg(node)$
        if $(ndeg.le.thresh)$ goto 500
        if $(ndeg.lt.mindeg)$ $mindeg = ndeg$
400 continue
500 ...
```

Here we have conditional expressions guarding updates to scalar variables. This is a classic case of a loop with reduction semantics. Such patterns commonly occur in computationally important loops in a wide range of codes [8].

[1]This example is drawn from the HPF-2 benchmark *cholesky*

Many frameworks have been developed for recognizing parallelism of this nature based on both syntactic and semantic schemes. All of these frameworks have one thing in common, however: the enabling factor underlying the transformation is the associative nature of the operation being performed. In this case, for example, *min* is an associative operation.

The loop above, however, is slightly different from a "standard" minimum reduction in that the presence of a conditional exit in the loop impacts the parallelizing transformation. When the above code is executed serially, there is an invariant point in the iteration space at which the loop will exit. This point may be an early exit, depending on the conditional `if (ndeg .le. thresh)`. However, the exit point may not be invariant when the loop is executed in parallel. To understand the reasons for this, consider the following parallelized version of the loop:

```
doall 400 $j = search, neqns$
        private $node, ndeg$
        $node = perm(j)$
        if $(marker(node).lt.0)$ goto 400
        $ndeg = deg(node)$
        if $(ndeg.le.thresh)$ goto 500
        critical section
                if $(ndeg.lt.mindeg)$ $mindeg = ndeg$
        critical section
400 continue
500 ...
```

When this code is executed, the iteration space $j = search, neqns$ will be partitioned amongst the p processors participating in the computation. However, regardless of the particular partitioning employed, it is possible that a given processor p_i may execute some iteration j which is not executed when the computation is performed serially. If $deg(perm(j))$ is a global minimum across the entire iteration space, the final result will be incorrect.

This presents a puzzle: we know that *min* operations are parallelizable, yet the application of a well-known transformation resulted in parallel code which is conditionally correct! The key to understanding this lies in realizing the fundamental property which enables parallelism in this loop. Earlier we pointed out that associativity is the underlying factor which enables parallelism, and indeed *min* is an associative operation. However, when the conditional exit is added to the mix, we now have an operation which is *not commutative*. The key point to realize is that there is a class of loops which perform non-commutative, associative operations. As a result, any transformation which parallelizes a loop of this nature must not commute the execution order of the iterations.

In light of this discovery, we have developed a framework for understanding parallelism in a loop based on the associativity of operations which accumulate, aggregate or *coalesce* a range of values of various types into a single conglomerate. In the following section we discuss related work, and then proceed to the introduction of the concept of an associative operation based on a *coalescing operator* of this nature.

2.2 Related Work

Over the years, the study of loops which perform coalescing operations has often focused on the solution of recurrence relations. The parallel solution of recurrences in Fortran, for example, has been a topic of study for several years (e.g., [32, 12, 35, 42, 1, 26, 22, 44]). Early work included parallel recurrence solvers which were implemented in hardware [12, 35]. More recently, techniques based on powerful symbolic analysis have been employed in the recognition and solution of recurrences statically at compile-time as well as dynamically at run-time.

2.2.1 Run-time Solution of Recurrences

The run-time solution of recurrences also has a fairly rich history. Suganuma, Komatsu, and Nakatani, for example, recognize and transform scalar reductions based on the detection of reduction semantics in the data dependence graph [59]. Rauchwerger and Padua test for the presence of privatizable arrays and reduction operations involving arrays in [50]. By treating individual array elements as scalars and recording at run-time whether a use of a given array element occurs outside the *expanded reduction statement* (or statements) involving the reduction variable, reduction operations can recognized and executed in parallel. Fisher and Ghuloum model loop bodies containing reductions and recurrences as composable functions in [17]. They determine whether, for a given loop, the composition of the function representing the loop body yields a function isomorphic to the original model. From this, a parallel prefix solution of the reduction/recurrence is generated. They are able to handle both scalar and array-based recurrences in singly-nested loops.

In our previous work [44] we take a general pattern-matching approach to the recognition and transformation of reductions. This approach has the advantage that complex conditional structures occurring in multiply-nested loops pose no particular problem. This approach is, however, inherently limited to the particular patterns programmed into the compiler.

2.2.2 Compile-time Solution of Recurrences

Compile-time solutions include [1], in which Harrison and Ammarguellat use abstract interpretation to map each variable assigned in a loop to a symbolic form, and match these against template patterns containing the closed-forms for commonly occurring recurrences. They are able to handle both induction variables and array-based recurrences in singly-nested loops. Haghighat and Polychronopoulos symbolically execute loops and use finite difference methods in conjunction with interpolation to determine closed-forms for recurrences involving scalar variables [26]. Their approach is capable of multiple scalar transformations including induction variable substitution. Other compile-time solutions include [22] in which Wolfe *et al* derive relations between variables by matching against cycles in the SSA graph, and then use matrix inversion (among other methods) to determine closed-forms for induction variables.

We also treat the solution of *generalized induction variables* at compile-time in [44]. A general pattern-matching approach is employed in the recognition phase of our induction solution algorithm. In the transformation phase, mathematical closed-forms for inductions are computed across the iteration spaces of potentially multiply-nested loops enclosing induction sites.

2.2.3 Associative Operations

Associative operations have been the basis for parallelization of reduction operations in both hardware and software systems for many years [34, 12, 35, 42, 30, 1, 17, 47, 50, 44, 59].

In most of these cases, the associativity is limited to a single binary operation involving the operator $+$ (addition) or $*$ (multiplication). For example, in [44], recurrence relations are solved using a run-time technique that is based on the associativity of the underlying operator within either a single reduction statement or a group of reduction statements which access the same reduction variable.

Recognition techniques based on the underlying semantics of reduction operations have been implemented in the Velour vectorizing compiler [30]. Similar to the techniques implemented in [17], these approaches identify variables which are computed as recurrent associative functions derived from statements in the body of the loop. Harrison also treats the parallelization of associative inductions, reductions, and recurrences in functional languages in [27].

2.2.4 Commutative Operations

In [51], Rinard and Diniz present a framework for parallelizing recursive function calls based on commutativity. Similarly, Kuck has shown that simple expressions (e.g., right-hand-sides of assignment statements) can be reordered based on combinations of both associativity and commutativity in *tree-height reduction* [33].

2.3 Associativity in Coalescing Loop Operators

In this section we introduce the concept of a *coalescing loop operator*. Following this, we demonstrate how associativity in a coalescing loop operator enables loop-level parallelism.

2.3.1 Coalescing Operators in Loops

Consider the following definition of a *loop operator*:

Definition 1: Loop Operator

Given a loop $\mathcal{L}$, a *loop operator* of $\mathcal{L}$ is defined as the body of L expressed as a function α of two arguments: $\alpha(X_i, X_j)$. X_i and X_j represent sets of operands. The binary operator α returns the result of operating on operand sets X_i and X_j.

□

The collection of multiple objects to a single data structure in a loop in an object-oriented language is an example of a loop operator which coalesces many objects into a conglomerate whole. In this case, the operand X_i is the data structure used to collect the objects.

To use a real-world example, think about this in terms of how a pot of soup is made. First, many things go into a pot of soup: vegetables like carrots and onions, potatos, spices, perhaps some kind of chicken or beef, and water. If we think of the process of putting things into the pot as a loop, then one item can be added to the pot each iteration - the loop might look something like:

```
$pot = empty$
do $i = 1, number\_of\_ingredients$
        $pot = pot$ add $ingredient_i$
enddo
```

In the end we have a pot of soup. It's a collection of objects (vegetables, potatos, etc.) which have been added one by one - in effect, the "many" have been reduced to the "one" (pot of soup). There are two operands involved each iteration: the pot, and the ingredient being added. The operation, or the reduction of many vegetables etc. to soup, thus involves two operands. The binary operator *add* is the act of adding to the soup.

Thus, in a coalescing operation, the first argument X_i to α represents rvalues of conglomerate operands. The second argument, X_j, is the source set of operands which are agglomerated with X_i. The new conglomerate is then returned by α.

Suppose now we wished to view the whole process in its entirety at one shot. Then the above job of making soup might look like this:

$$add(\ add(\ \ldots\ add(\ add(empty\ water)\ celery)\ldots\ potatos)\ salt)$$

This is the same process expanded (or unrolled, for those of you familiar with the terminology), and expressed in a functional form. The operator α is *add*. In each case, α takes two arguments; the left argument represents the value of the (initially empty) pot, and the right argument is the ingredient being added[2].

These concepts can be generalized as follows:

Definition 2: Coalescing Loop Operator

Let α be defined as the body of loop $\mathcal{L}$ where $\mathcal{L}$ is represented in the form

$$\mathcal{L}(X_0, X_1, \ldots, X_{k-1}, X_k) = \alpha(\ \alpha(\ \ldots\ \alpha(\ \alpha(X_0, X_1), X_2), \ldots X_{k-1}), X_k)$$

This defines $\mathcal{L}$ in terms of the loop operator $\alpha(X_i, X_j)$ for the entire iteration space. The binary operator α is termed a *coalescing loop operator*. The left operand X_i is the *conglomerate operand set*, or simply the *conglomerate operand*. Each right operand $X_j,\ j = 1, k$ is an *assimilated operand set*, or simply an *assimilated operand*.

□

Let's reflect for a moment on this framework. What types of loops can be represented in such a model? Certainly well-known reductions based on operations such as *min, max, SUM*, etc. can be represented in a straightforward manner. However, we have determined that indexed data structures such as arrays can also be viewed as conglomerate operands to coalescing loop operators. We will address this point in more detail in section 2.3.8 following.

We will now move on to consider coalescing loop operators which are associative in nature. First we define associativity, then proceed to develop a framework for understanding associativity in coalescing loop operators.

2.3.2 The Definition of Associativity

Given an operator $\oplus$ and operands a, b, and c, $\oplus$ is an *associative operator* if the following holds:

[2]If all this talk of soup is making you hungry, perhaps we should move on. :)

$$((a \oplus b) \oplus c) \equiv (a \oplus (b \oplus c))$$

A computation which is based upon an associative operator is termed an *associative operation*.

2.3.3 Associative Coalescing Operators

The concept of associativity can be extended to include coalescing loop operators as associative operations. The central idea is that a coalescing operator can be considered a single operator consisting of a collection of associative operations performed within a loop. In this light, the collection of the various individual operators can be considered a single *associative coalescing loop operator*.

Let's now consider the case where we have a coalescing loop operator which is associative. The central question is "What parallelizing transformation does the property of associativity enable?". If we consider the case in which α is an associative coalescing loop operator, then the law of associativity may be applied to $\mathcal{L}$ to yield:

$$\alpha(\ \alpha(\ \ldots\ \alpha(\ \alpha(X_0, X_1), X_2), \ldots X_{k-2}), \alpha(X_{k-1}, X_k))$$

In this case we used the fact that α is an associative operator to perform the transformation

$$((a \oplus b) \oplus c) \equiv (a \oplus (b \oplus c))$$

to $\mathcal{L}$ where $a = \alpha(\ \ldots\ \alpha(\ \alpha(X_0, X_1), X_2), \ldots X_{k-2})$, $b = X_{k-1}$, and $c = X_k$. This process could be repeated to, for example, reassociate X_{k-3} with X_{k-2}, etc.

To understand this transformation more clearly, consider the following example using actual values for the operands:

```
$sum = 0$
do $i = 1, 4$
        $sum = sum + i$
enddo
```

Cast in terms of an associative coalescing loop operator α, this loop can be modeled as $\mathcal{L} = +=(+=(+=(+=(0\ 1)\ 2)\ 3)\ 4)$. Here α is $+=$, a coalescing loop operator with the semantics "add the assimilated argument on the right to the conglomerate argument on the left and return the resulting sum". The value for k in this case is 4, and $X_{0-4} = [\{0\}, \{1\}, \{2\}, \{3\}, \{4\}]$ are the assimilated operand sets accessed during execution of the loop[3].

By assigning $a = +=(+=(0\ 1)\ 2)$, $b = 3$, and $c = 4$, this expression can be reformulated as $+=(+=(+=(0\ 1)\ 2)\ +=(3\ 4))$ using the rule $+=(+=(a\ b)\ c) \equiv +=(a\ +=(b\ c))$. Clearly this loop can be executed in parallel: in fact, the reassociation of operands can be repeatedly applied to regroup invocations of α into a form which distributes the iteration space evenly across multiple processors. For example, if we consider the same loop iterating $i = 1, 8$ the following is a reassociated order which can be executed in parallel on four processors:

$$+=(+=(+=(+=(+=(0\ 1)\ 2)\ \ +=(3\ 4))\ \ +=(5\ 6))\ \ +=(7\ 8))$$

In the following section we present a transformation capable of achieving such a reassociation (or regrouping) of operands.

2.3.4 Transforming Associative Coalescing Loop Operators

Traditional parallelizing transformations involve the use of a critical section to guarantee exclusive access to shared variables. For example, the variable *mindeg* in the code example given in the introduction is a shared reduction variable which must be updated atomically.

When a coalescing loop operator is not commutative, however, the parallelizing transformation must guarantee that different iterations of the loop are not commuted. In other words, when the loop is executed on more than one processor, the iterations must not be interleaved.

In the following discussion we do not treat the theoretical underpinnings which necessitate the given transformation: we save this discussion for section 2.3.5.

[3]Note that the function return values implicitly form the remaining conglomerate operands of the binary operator $+=$.

The following four steps are needed in order to transform a loop based on associativity alone:

- Privatization of shared variables
- Initialization of private variables
- Block loop scheduling
- Cross-processor reduction

Privatization refers to the creation of thread or process-private copies of shared global variables[61]. The second step involves the initialization of the newly created private variables. In the third step, the iteration space of the loop is broken into contiguous slices, and each processor executes a slice of the original iteration space. Within each slice on each processor the iterations are executed in the original serial order. Across processors, the slices are also kept in the original serial order. For example, a loop with iteration space $i = 1, 8$ executing on 4 processors would be scheduled as follows:

$$(\overbrace{(i = 1, 2)}^{p_1}\overbrace{(i = 3, 4)}^{p_2}\overbrace{(i = 5, 6)}^{p_3}\overbrace{(i = 7, 8)}^{p_4}).$$

The final step is a cross-processor reduction. This involves the serial execution of the associative coalescing loop operator with each of the privatized variables in turn. This operation must also preserve the original order of execution and thus insure that iterations of the loop are not commuted.

To understand these four steps, let's consider the following example:

```
do $i = 1, n$
        $sum = sum + a(i)$
enddo
```

The following is the parallelized version of this example. The language used in this code is based on IBM's Parallel Fortran [29] with extensions which we have added to adapt the language to the special needs of the associative transformation.

```
parallel loop, block $i = 1, n$
          private $sum_p$
          dofirst
                    $sum_p = 0$
          doevery
                    $sum_p = sum_p + a(i)$
          enddo
          dofinal, ordered lock
                    $sum = sum + sum_p$
enddo
```

The first two steps in the transformation are the privatization and initialization of the shared variable sum. In the above code, sum_p is the processor-private copy of sum. If p processors participate in the computation, then p private copies of sum are made, one on each processor. In an abstract sense, privatization effectively creates an uninitialized conglomerate operand for use on each processor. In the `dofirst` section of code, sum_p is initialized on each processor. `dofirst` indicates that this section of code is executed once at the invocation of the parallel loop by each processor.

The `doevery` clause indicates the section of code that is to be executed every iteration. The majority of the computation takes place in this loop. Each processor is given a contiguous slice of the iteration space. For example, assuming 4 divides n, if four processors participate in the computation, the iteration space would be divided as in the previous example:

$$(\overbrace{(i = 1, n/4)}^{p_1}\overbrace{(i = n/4 + 1, 2n/4)}^{p_2}\overbrace{(i = 2n/4 + 1, 3n/4)}^{p_3}\overbrace{(i = 3n/4 + 1, n)}^{p_4}).$$

In the above example, "parallel loop, block" refers to a schedule of this nature - i.e., the slices (or blocks) are contiguous, and within each block (i.e., on each processor) the iterations are executed in the original serial order.

This `doevery` section of the parallel loop is executed as a dependence-free *doall* loop [25]. Thus the associative transformation enables the execution of the bulk of the original serial loop consisting of an associative coalescing loop operator as a fully parallel doall loop.

After each processor has completed executing its slice of the iteration space in the `doevery` section, they each compute the `dofinal` once prior to loop exit. In the above case, this operation updates the shared variable *sum*. The update is atomic, and is done in the original order in which the slices were distributed. In other words, according to the schedule just presented, p_1 will update *sum* first, followed by p_2, etc. This is the meaning of the "dofinal, ordered lock" directive, and this schedule insures that the final cross-processor reduction does not interleave (i.e., commute) the slices of the iteration space.

Before concluding this section, we make the following definition:

Definition 3: Parallelizing Transformation of an Associative Coalescing Loop

Given a loop $\mathcal{L}$ with associative coalescing loop operator body α, we denote $\mathcal{L}$ transformed as above $\mathcal{L}^t$.

□

2.3.5 Theoretical Underpinnings

In the previous section we briefly demonstrated how the private copies of recurrent variable operands must be initialized (prior to loop execution) as part of the parallelizing transformation of an associative coalescing loop operator. In general, for operand sets which contain recurrent variables to be transformed as outlined in section 2.3.4 above, there must exist an *identity operand* set $\mathcal{I}$ which satisfies the following:

Definition 4: Identity Operand

For each recurrent variable in X_i, there must exist a corresponding *identity* in $\mathcal{I}$ such that

$\alpha(X_i, I) \equiv X_i$

□

In our example loop involving the sum reduction in the preceding section, the member of the identity operand $\mathcal{I}$ corresponding to the addition operation on the recurrent variable *sum* is the identity for addition, 0.

At this point it is possible to clarify a possible confusion as to how operands are actually reassociated. When transformation $\mathcal{L}^t$ is made, in effect private conglomerate operands are created locally on each processor (as noted in section 2.3.4). The corresponding general form of the law of associativity for coalescing loop operators takes the following form:

$$\alpha(\ \alpha(X_0\ \alpha(\mathcal{I}\ X_1))\ \alpha(\mathcal{I}\ X_2)) \equiv \alpha(X_0\ \alpha(\ \alpha(\mathcal{I}\ X_1)\ \alpha(\mathcal{I}\ X_2)))$$

An interesting point that we have not considered is the role the assignment operator plays in coalescing loop operators. Naturally, when a loop in an imperative language is executed, assignments are made. As will become clear in section 2.3.7, assignment is actually an associative binary coalescing operator. The *identity* for assignment is the rvalue of the variable being assigned. E.g., in terms of our example, $sum_p = sum_p$ leaves sum_p unchanged. In light of this fact, the initial value of a given private copy sum_p is the *coalesced identity* 0.

We summarize these results in the following lemma:

Lemma 1: Associative Coalescing Loop Operators

Given a loop $\mathcal{L}$ with coalescing loop operator α as defined in Definition 2 above, necessary and sufficient conditions for the `doevery` iterations of $\mathcal{L}^t$ to be executed as a doall loop are:

- $\exists$ *identity operand* set $\mathcal{I}$ as defined in Definition 4 above
- $\alpha(\alpha(X_i, X_j), X_k) \equiv \alpha(X_i, \alpha(X_j, X_k))$

$\square$

In the following sections we present several important cases in which we have identified coalescing loop operators which are associative in nature. A number of these cases involve loops previously considered difficult or impossible to parallelize; however, within the framework presented herein these loops can indeed be parallelized.

2.3.6 Loops that Perform Output Operations

In our research we have determined that output to a sequential file is a non-commutative, associative operation which coalesces output from the program to the file. To understand this point, consider a simple example involving the lisp **append** operator:

```
append(append([1] [2]) [3])
    ⇒ append([1 2] [3])
    ⇒ [1 2 3]
append([1] append([2] [3]))
    ⇒ append([1] [2 3])
    ⇒ [1 2 3]
```

Here we are making a simple list of the numbers 1, 2, and 3. The [ ] enclose lists. The append operator takes two operands which are lists and creates and returns a new list by appending the second operand to the first. In the first case above, the list [2] is first appended to the list [1], resulting in the list [1 2]. The list [3] is then appended to this list, resulting in the final list [1 2 3]. In the second case, the list [3] is first appended to the list [2], resulting in the list [2 3]. This list is then appended to the list [1], resulting in the same final list. The final result is identical in both cases even though the associative order of the operands differ.

However, if we now consider a case where we attempt to commute the operands, the results will differ:

```
append([1] [2])
    ⇒ [1 2]
append([2] [1])
    ⇒ [2 1]
```

Clearly the append operator is not commutative. This has implications for the parallelization of output operations in that loops containing sequential output operations must be parallelized based on associativity alone. In Chapter 3, section 3.4, we discuss how this is accomplished based on the transformation presented previously in section 2.3.4.

The specific techniques used to parallelize output operations of this nature are applicable generally in computer programs that perform output. These techniques can be applied to the automatic parallelization of computer programs in systems such as the Polaris restructurer [8].

2.3.7 Loops with Dynamic Last Values

When a loop writes to a variable which is “live-out”, the *last value* of the variable must be preserved across loop exit. For example, in the following code segment the variable *temp* is live-out:

```
do $i = 1, n$
        $\ldots$
        $temp = \ldots$
        $\ldots$
        $\ldots = temp$
        $\ldots$
enddo
$\ldots = temp$
```

After execution of the loop terminates, the rvalue of the variable *temp* is referenced in the statement following the loop. This is what is known as a last value.

Variables are often used as temporaries in loops, and when a loop is parallelized multiple writes to such shared temporaries create loop-carried output dependences. These are normally broken by privatizing the temporary. For example:

```
doall $i = 1, n$
        private $temp_p$
        $\ldots$
        $temp_p = \ldots$
        $\ldots$
        $\ldots = temp_p$
        $\ldots$
        if $(i.eq.n) temp = \ldots$
enddo
$\ldots = temp$
```

However, due to the reference after the loop, the last value must be written to the shared global variable *temp* during the last iteration. In this case identifying the last iteration is straightforward since *temp* is written every iteration. However, when a live-out shared variable is written conditionally, it is difficult to identify exactly when the last value is written. The following exemplifies this situation:

```
do $i = 1, n$
        $\ldots$
        if $(condition) temp = \ldots$
        $\ldots$
        if $(condition) \ldots = temp$
        $\ldots$
enddo
```

$$\ldots = temp$$

In this case the variable *temp* has a *dynamic last value*. As mentioned, this poses a difficulty for parallelizing compilers given that *condition* is loop variant. In the past this problem has been addressed using timestamps to identify each write. Writes by processors executing iterations later than the current timestamp are permitted. Processors executing iterations earlier than the current timestamp are not permitted to update the shared variable [60]. This solution incurs additional overhead in terms of both space to maintain timestamps and computational time to achieve synchronized access to shared variables.

Within the framework of coalescing loop operators however, we have determined that last value assignment is an associative operation which can be parallelized based on the transformation outlined in section 2.3.4.

To understand this point, consider the application of the *assign* operator, the functional equivalent of assignment:

```
assign((assign 1 2) 3)
    ⇒ (3)
assign(1 (assign 2 3))
    ⇒ (3)
```

The *assign* operator simply returns the argument on the right. This is assignment. As can be seen, *assign* is associative. However, if we now consider a case where we attempt to commute the operands of *assign*, the results differ:

```
(assign 1 2)
    ⇒ (2)
(assign 2 1)
    ⇒ (1)
```

Clearly assignment is not commutative[4]. Yet our example loop containing a dynamic last value can be readily parallelized based on associativity alone:

[4]Discounting the case $id = id$, self-assignment or the *identity operation*

```
parallel loop, block $i = 1, n$
    private $temp_p, written_p$
    dofirst
        $written_p = False$
    doevery
        $\ldots$
        if ($condition$) then
            $temp_p = \ldots$
            $written_p = True$
        endif
        $\ldots$
        if ($condition$) $\ldots = temp_p$
        $\ldots$
    enddo
    dofinal, ordered lock
        if ($written_p$) $temp = temp_p$
enddo
```

One noteworthy difference between this transformation and those encountered previously is the fact that $temp_p$ does not need to be initialized. Recall that the `dofirst` portion of the loop initializes private copies of shared variables. In this case, however, we need not be concerned with initialization of $temp_p$ because assignment (like output) does not involve an explicit recurrence relation and no initial value is needed. Note however that if $temp_p$ was read and written under differing conditions, the initial value of $temp$ would have to be copied to $temp_p$ in the `dofirst` section of code.

The above discussion has been based on the determination of dynamic last values of scalar variables. However, this technique can be easily extended to include the parallelization of loops with dynamic last values of entire arrays or array sections. Such a case occurs in the SPEC CFP95 benchmark *appsp* discussed in Chapter 4.

The fact that loops with dynamic last values can be transformed based on associativity alone accentuates once again that the fundamental property enabling parallelism in loops is associativity.

2.3.8 Operators Involving Arrays in Loop-carried Flow Dependences

In this section we consider the parallelization of linear recurrences in the framework of coalescing loop operators. We address for the first time a case in which operands are array elements with loop-carried flow dependences between individual elements.

It is well known that recurrences are parallelizable when based on associative binary operators [27]. By treating the array used to contain the result as a conglomerate operand, associative recurrence relations can be modeled as associative coalescing loop operators.

Consider the following functional representation of the non-homogeneous recurrence relation $a_i = a_{i-1} \oplus \beta_i$ where β_i is loop-variant[5]:

$$\mathit{op\&append}(\mathit{op\&append}(\ldots \mathit{op\&append}(\mathit{op\&append}(a_0\ \beta_1)\ \beta_2) \ldots \beta_{n-1})\ \beta_n)$$

op&append takes the left argument, performs the binary operation $\oplus$ with the right argument, and returns a list with this result appended to the left argument. The operand a_0 represents the initial value of the conglomerate operand. Similarly, $\beta_1 \ldots \beta_n$ are the operands which will be assimilated into the conglomerate. In effect, we have represented the array a as a conglomerate operand rather than as multiple individual elements.

As defined above, *op&append* is associative. This can be easily demonstrated in a way similar to append but with one important additional constraint:

$$\begin{array}{l}
\mathit{op\&append}(\ \mathit{op\&append}(\ [1]\ [2])\ [3]) \\
\quad \Rightarrow \mathit{op\&append}(\ [1\ (1 \oplus 2)]\ [3]) \\
\quad \Rightarrow [1\ (1 \oplus 2)\ ((1 \oplus 2) \oplus 3)] \\
\mathit{op\&append}(\ [1]\ \mathit{op\&append}(\ [2]\ [3])) \\
\quad \Rightarrow \mathit{op\&append}(\ [1]\ [2\ (2 \oplus 3)]) \\
\quad \Rightarrow [1\ (1 \oplus 2)\ (1 \oplus (2 \oplus 3))]
\end{array}$$

Here items in [ ] are lists, and ( ) are used to properly associate the operands of the binary operator $\oplus$. Note that if $\oplus$ is associative, then *op&append* is associative. This is a prototypical example of how two operators can be coalesced to form a single loop operator.

A straightforward transformation of $\mathcal{L}$ to $\mathcal{L}^t$ will result in inefficient code if the linear recurrence comprises the bulk of the computation in $\mathcal{L}$. In this case, the following well-known variation on $\mathcal{L}^t$ can be substituted in its place:

[5]In Chapter 4 section 4.3.1, we discuss the closed-form solution of such recurrences when β is loop invariant.

Transformation of *op&append*

Given a loop $\mathcal{L}$ with coalescing loop operator *op&append*, the algorithm to execute $\mathcal{L}$ on p processors involves the following four steps:

- Divide the iteration space of $\mathcal{L}$ into p contiguous slices
- Each processor computes the recurrence $a_i = a_{i-1} \oplus \beta_i$ for its given slice, using as an initial value the identity for the coalescing loop operator
- Serially compute the last values of each slice $[2, p-1]$
- Each processor $[2, p]$ combines the last value from the preceding slice with each element in its slice using $\oplus$

□

Here we have taken advantage of the indexable nature of the array a to treat each element as a scalar. This enables us to parallelize the loop without privatizing a.

The ability to optimize the parallelization of *op&append* in this way exemplifies the utility of the framework of coalescing loop operators: we get the best of both worlds in that we view a as a conglomerate for the purposes of identifying parallelism, but optimize the parallel performance by accessing a as multiple individual elements.

2.3.9 Loops with Conditional Exits

In our study of applications from several fields of science, we have determined another case of operations which are associative but conditionally non-commutative in nature. Let's return to the original example presented in the introduction, a loop with a conditional exit. Here we have duplicated the original parallel version of this loop:

```
doall 400 $j = search, neqns$
        private $node, ndeg$
        $node = perm(j)$
        if $(marker(node).lt.0)$ goto 400
        $ndeg = deg(node)$
        if $(ndeg.le.thresh)$ goto 500
        critical section
                if $(ndeg.lt.mindeg)$ $mindeg = ndeg$
        critical section
```

```
400 continue
500 ...
```

We determined in the introduction that the incorrect minimum will be returned when a processor p executes an iteration j which is not executed serially and at the same time $deg(perm(j))$ is a global minimum across the entire iteration space.

In the past, loops such as these were considered difficult to parallelize. However, within the framework of coalescing loop operators, this loop can be parallelized. The key lies in recognizing that the loop body, including the conditional exit, is a non-commutative, associative coalescing loop operator. The following portrays this same loop parallelized based on the transformation discussed in section 2.3.4:

```
parallel loop, block $j = search, neqns$
    private $node_p, ndeg_p, mindeg_p, exit_p$
    dofirst
        $mindeg_p = MAX$
        $exit_p = False$
    doevery
        $node_p = perm(j)$
        if ($marker(node_p).ge.0$) then
            $ndeg_p = deg(node_p)$
            if ($ndeg_p.le.thresh$) then
                $exit_p = True$
                goto 500
            endif
            if ($ndeg_p.lt.mindeg_p$) then
                $mindeg_p = ndeg_p$
            endif
        endif
    enddo
    500 continue
    dofinal, ordered lock
        $mindeg = MIN(mindeg, mindeg_p)$
        if ($exit_p$) then
            lock-exit
        endif
enddo
```

The coalescing loop operator in this example consists of a thresholded minimum operation performed across $deg(perm(j))$. Each processor computes the minimum accessed in its slice of the iteration space and stores the result in $mindeg_p$. When an early exit is taken, the condition is noted in the private variable $exit_p$ and the `doevery` is exited.

In the `dofinal` section of the parallel loop, the minimum of each slice is taken. However, if the processor corresponding to a given slice took an early exit, the `lock-exit` routine is called and any remaining processors waiting for entry into the `dofinal` critical section are discharged. This has the result that execution of the `dofinal` section of code ceases for all processors.

Several points of interest are noteworthy here. However, an overriding concern is that if the loop actually takes an early exit, it may exhibit little or no speedup. Why is this? The above transformation executes each slice regardless of whether the loop exited in an earlier iteration. This may result in a considerable amount of unnecessary computation.

What is needed is global communication that provides for the notification of an exit condition by processors executing slices which occur before other slices. This would enable all processors to cease execution when the condition is first true for the processor executing the "lowest order" slice (i.e., the slice occurring first in the iteration space). This optimization is implemented in a transformation discussed in Chapter 4, section 4.4.6.

A second concern has to do with the partitioning of the iteration space. Consider the case where an early exit is taken in the first slice. If the entire iteration space is divided evenly amongst the processors, even after the optimization just mentioned, the parallel execution time could be the same as the serial. However this problem can be alleviated by employing a schedule which distributes the slices in smaller pieces while still maintaining the original order of execution. An example of this optimization can be found in the HPF-2 benchmark *cholesky* in Appendix B, subroutine GENQMD, loop 400.

Not surprisingly, while and goto loops pose a similar problem. The following example demonstrates this point:

```
15 continue
        $sum = sum + a(i)$
        if ($condition$) goto 20
goto 15
20 continue
```

Control flow will exit this goto loop depending on the value of *condition*. Traditionally, loops of this nature have been considered difficult to parallelize due to the unknown size of the iteration space. In this sense the problem is similar to that discussed previously in that unwanted side-effects may cause the computation to be incorrect. However, unlike the case discussed above, goto and while loops will be non-commutative whenever the iteration space is "overshot" by any one processor participating in the computation. This follows from the same reasons as those discussed for the multiple-exit do loop.

The discussion to this point has been limited to conditionally exited loops in which the exit condition contains no references to conglomerate operands of coalescing operations which involve recurrent variables[6]. In the event that an exit condition references such an operand, transformation $\mathcal{L}^t$ may need to be modified in order to attain appreciable speedups.

Essentially there are two tractable cases: the first, in which a closed-form exists for the coalescing operation (as discussed in Chapter 4, section 4.3.1); or second, no closed-form exists but the coalescing operation is associative. In the former case parallelization is straightforward since the loop bounds are obtainable from the closed-form; i.e., the conditional exit has no effect on the parallelizability of $\mathcal{L}$ since $\mathcal{L}$ has effectively become a do loop with a known iteration space. If the loop operator is associative, the loop can be transformed using $\mathcal{L}^t$.

In the latter case, transformation $\mathcal{L}^t$ may still be applied with a slight modification. Following [49], the `doevery` loop of $\mathcal{L}^t$ can be strip-mined and executed for a pre-determined number of iterations after which the (accumulated) exit conditions for each slice can be checked. This can be repeated until the loop exits.

A final note on the application of $\mathcal{L}^t$ to conditionally exited loops relates to the privatization step of the transformation. When a loop is "overshot", non-privatized operands may be altered. This may be solved by privatizing all operands and employing copy-in and copy-out as discussed in Chapter 4.

[6]I.e., exit conditions do not reference variables with loop-carried flow dependences

2.3.10 Associativity in Pseudo Random Number Generation

As a final example of an associative coalescing loop operator, we consider operations which can be viewed as associative in an abstract sense. For example, many computer applications make use of pseudo-random number generators (RNGs) as part of algorithmic solutions to scientific problems. We have determined that RNGs occur frequently in computationally important loops in the sparse and irregular benchmark suite discussed in Chapter 5. Such generators commonly employ recurrence relations of various sorts including, for example, linear homogeneous relations. As noted, calls to generators of this nature often occur within computationally important loops. However, due to the recurrent nature of these relations, such calls can serialize a loop.

Although the generation of pseudo-random numbers is not an associative operation per se, it can be thought of as an associative operation if we consider the fact that the particular pseudo-random number returned by a generator is not important as long as it is truly "random". In other words, numbers are random only in association with other numbers. In this sense, the generation of a stream of random numbers can be considered a coalescing operation: each new random number generated represents a new operand to be assimilated into the conglomerate stream of random numbers.

The substitution of RNGs can thus be viewed as the replacement of a non-associative, single-threaded RNG with an associative RNG. In Chapter 4 we will further discuss this issue, and in Chapter 5 we present several examples of cases in which single-threaded RNGs can be successfully substituted by thread-parallel RNGs.

2.3.11 Commutativity vs. Associativity

In the introduction we highlighted the fact that a distinction must be made between associativity verses commutativity as the basis for parallelization. Simple operators such as $+$ and $*$ are both commutative and associative and expressions involving such operators can often be parallelized based either on associativity, commutativity, or some combination thereof [33]. In the preceding sections, we've seen several examples of coalescing loop operators which were non-commutative. As a result, the question remains open as to what role commutativity plays in parallelizing coalescing loop operators. In order to answer this question we must consider the parallel execution of a coalescing loop operator which is both commutative and associative.

In order for a coalescing loop operator to be commutative, the following must hold: $\alpha(X_i, X_j) \equiv \alpha(X_j, X_i)$. This is simply the definition of commutativity applied to α. Consider the the following loop:

```
sum = 0
do i = 1, 8
        sum = sum + i
enddo
```

The coalescing loop operator α is $+=$ as defined in section 2.3.3, and the reduction is on the variable *sum*. However, α is both an associative and commutative operator. By Lemma 1, this loop can be transformed into $\mathcal{L}^t$ and the resulting `doevery` doall loop executed on four processors as follows:

$$\overbrace{+\!=(\ +\!=(0\ 1)\ 2)}^{p_1}\ \overbrace{+\!=(\ +\!=(0\ 3)\ 4)}^{p_2}\ \overbrace{+\!=(\ +\!=(0\ 5)\ 6)}^{p_3}\ \overbrace{+\!=(\ +\!=(0\ 7)\ 8)}^{p_4}$$

It appears that commutativity cannot be applied to much effect in this loop. However, that is not quite true. In fact, if we wished to schedule iterations 7 and 8 on p_1 and iterations 1 and 2 on p_4, the commutativity of $+=$ would allow us to do so. Why? To understand this, consider the following reassociated form of $\mathcal{L}$:

$$+\!=(\ +\!=(\ +\!=(\ +\!=(\ +\!=(0\ 1)\ 2)\ \ +\!=(\ +\!=(0\ 3)\ 4))\ \ +\!=(\ +\!=(0\ 5)\ 6))\ \ +\!=(\ +\!=(0\ 7)\ 8))$$

This expression depicts the actual order of execution in the $\mathcal{L}^t$ `doevery` and `dofinal, ordered lock` sections. Commutativity clearly allows us to commute the two operands $+\!=(\ +\!=(\ +\!=(\ +\!=(0\ 1)\ 2)\ \ +\!=(\ +\!=(0\ 3)\ 4))\ \ +\!=(\ +\!=(0\ 5)\ 6))$ and $+\!=(\ +\!=(0\ 7)\ 8)$.

What this means in general is that commutativity enables the use of more flexible processor scheduling. The order in which the assimilated operands are reduced into the conglomerate whole does not affect the final result.

We thus conclude that two types of coalescing loop operators exist: operators which are both commutative and associative, and operators which are associative but not commutative. We term the latter *ordered associative coalescing loop operators*, and the former *unordered associative coalescing loop operators*. This choice of terminology reflects the fact that non-commutative, associative coalescing loop operators have an intrinsic order in which they must be computed.

2.4 Conclusions and Future Work

It has been an interesting task "chasing the tail" of the principle property determining the parallelizability of a given operation. However, it is our conclusion that associativity is fundamentally the only property necessary to parallelize loops with coalescing loop operators.

An open question is "Exactly what class of loops can be represented as coalescing loop operators?". This remains to be seen; however, based on our research to date, the framework is applicable to numerous loops. Further investigation is needed to determine how widely applicable the model is in theoretical terms, and this will necessarily involve a precise categorization of the types of operators which occur in loops.

In the introduction to this chapter we discussed the need for a recurrence recognition scheme more general than the current pattern-matching techniques implemented in the Polaris restructurer. We believe an approach which tests for associativity of coalescing loop operators may be a reasonable solution to the problem of recognizing parallelism in loops containing recurrences.

The following is a high-level view of one form an algorithm for determining associativity in loops could take. It is based on the Gated-SSA representation of a program [63]:

```
Given a loop $\mathcal{L}$

Transform $\mathcal{L}$ to GSA form
For each $\mu$ function variable $v$ at the header of $\mathcal{L}$
        back_substitute(variable $e$ from loop-carried edge)
Topologically sort $\mu$ functions into directed acyclic graph $\mathcal{G}$
        based on non-loop carried flow-dependences
```

```
// This DAG is the operator representing the loop body
if (is_associative_coalescing_op($\mathcal{G}$)) mark_parallel($\mathcal{L}$)

void back_substitute($expression$)
        if ($expression$ is one of $\mu$ function variables) return
        else if ($expression$ is invariant) return
        else if ($expression$ occurs on the lhs of assignment statement $S$)
                $expression$ = right_hand_side($S$)
        for each $subexpression$ in $expression$
                back_substitute($subexpression$)

boolean is_associative_coalescing_op(DAG $\mathcal{G}$)
        Bind initial values to $\mu$ function variables which are
                flow-predecessors of other $\mu$ function variables
        if ($(\mathcal{G}_{\mathcal{A}} \oplus \mathcal{G}_{\mathcal{B}}) \oplus \mathcal{G}_{\mathcal{C}} \equiv \mathcal{G}_{\mathcal{A}} \oplus (\mathcal{G}_{\mathcal{B}} \oplus \mathcal{G}_{\mathcal{C}})$)
                // $\mathcal{G}$ is associative
                return True
        else return False
```

There are many issues to address in the implementation of an algorithm such as the one sketched above. One interesting issue relates to the need to develop a representation capable of modeling loop bodies in a form that can be recognized as a coalescing loop operator. This presents a challenging problem in the field of parallelizing compilers.

In the following chapter we continue the discussion of associative coalescing loop operators with a case study of a modern C^{++} information retrieval application. We analyze this application based on the concepts presented in this chapter and demonstrate the applicability of the framework in the emerging Digital Library field.

CHAPTER 3

The Role of Coalescing Operations in Parallelizing Computer Programs: A Case Study

3.1 Introduction

This chapter treats the issue of determining parallelism in computer programs based on a general approach which models loops as *coalescing loop operators* as presented in Chapter 2. We perform a case study of the semantic retrieval application **cSpace** [45] in order to evaluate the effectiveness of this model. The performance of **cSpace** is characterized on a modern shared-memory multi-processor in order to demonstrate the the applicability of this approach.

In the following section we introduce the **cSpace** application and provide an overview of the algorithm to compute *Concept Spaces*. In section 3.3, we analyze the parallelism present in **cSpace** based on the model presented earlier in Chapter 2. In section 3.4, we outline the implementation of the parallelism present in **cSpace**, and in the final section we characterize the performance of **cSpace** on a shared-memory multi-processor.

3.2 The cSpace Application

The **cSpace** application benchmark represents an area of growing importance in the field of computational science. Developed as part of the federal NII Digital Library Initiative (DLI) at the University of Illinois [57], **cSpace** is a hybrid symbolic/numeric application which determines relationships between terms in a collection of documents. The resulting map between terms is designated a *Concept Space* and is useful in the refinement of queries presented to the collection. Concept Spaces are used, for example, in interactive query sessions as part of the DLI testbed at the University of Illinois, Urbana-Champaign [55]. Algorithms to perform iterative search refinement which incorporate the computation of *Concept Spaces* are also under development as part of the Digital Library Research Program (DLRP) at Illinois.

3.2.1 The Object Model

Figure 3.1 depicts the **cSpace** object model using the *Object Model Technique*, OMT [54]. Four object classes exist in the system: **Term**, **Document**, **Cooccurrence**, and **ConceptSpace**. The $\diamond$ symbol represents the *aggregation* operation. A ConceptSpace, for example, aggregates a group of Terms. The $\circ$ symbol on the link connecting **Term** to **ConceptSpace** indicates that a **ConceptSpace** aggregates 0 or more **Terms**. The link from **Term** to **Term** labeled *similarity* touches the **Cooccurrence** object with a partial loop. This link represents the *association* of **Terms** to other **Terms** in a similarity mapping. Similarity is a mapping from one term to another which indicates how similar they are semantically, based on their occurrence and co-occurrence across a document collection. When a **Term** occurs together with another **Term** in a document, the terms are said to co-occur, and a **Cooccurrence** object is formed. The use of a $\bullet$ indicates a mapping involving *many* objects. Many **Cooccurrence** objects, for example, are aggregated in **Term** objects. The small $\{o\}$ indicates that the collection of objects is ordered, and 1^+ qualifies the *aggregation* operation to mean that one or more objects are being aggregated.

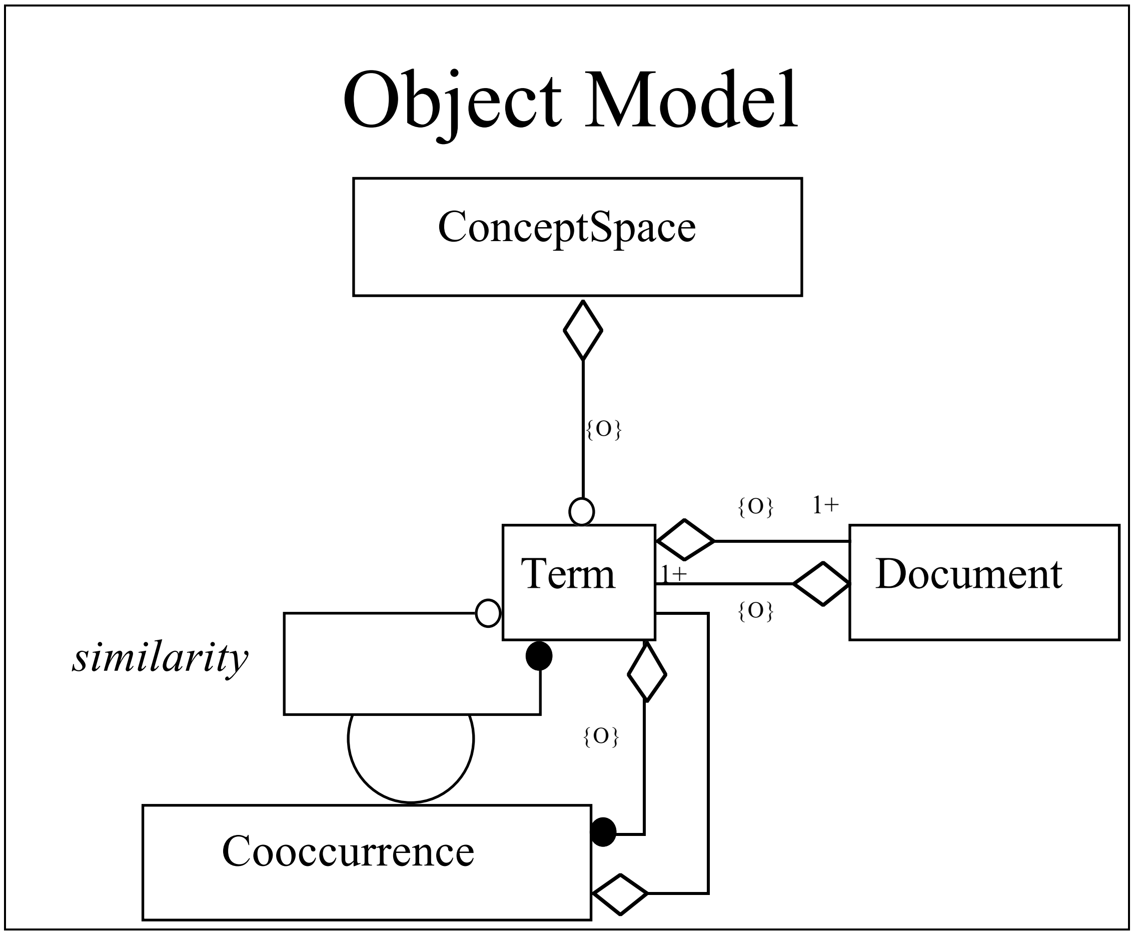

Figure 3.1: Object Model for **cSpace**

3.2.2 The Algorithm

Algorithms to compute *Concept Spaces* have been under development for several years. **cSpace** is a parallel C^{++} shared-memory implementation based in part on algorithms described in [53, 19, 11].

The computation proceeds in phases. The first phase is symbolic in nature and accounts for less than 5% of the sequential execution time. The second phase combines symbolic and numeric aspects, and accounts for the remaining 95% of the serial execution time.

3.2.2.1 Phase I

The following pseudo-code depicts the computation in Phase I of the algorithm.

```
For each input document
    Create Doc object doc and insert in global Docs collection
    Extract noun phrases from doc
    For each noun phrase in doc
        If phrase has not been entered in global Terms collection
            Create Term object term and insert in global Terms collection
        Else
            Return reference to term associated with phrase
        Increment term.sum_term_freq
        Increment term.term_freq for doc
        If this is the first occurrence of noun phrase in doc
            Increment term.doc_freq
            Insert reference to doc in term.docs
            Append reference to term to doc.terms_in_doc
```

In this phase documents are read from input, noun phrases are extracted, and occurrence frequencies for each noun phrase in each document, and across all documents in the collection, are recorded in a global database of terms. The complexity of this computation is $O(\mathcal{D}\mathcal{T})$ where $\mathcal{D}$ is the total number of documents and $\mathcal{T}$ is the total number of noun phrases across the collection.

3.2.2.2 Phase II

The second phase computes term co-occurrence and similarity. As mentioned in section 3.2.1, similarity is a mapping from one term to another which indicates how similar they are semantically. The following pseudo-code depicts the computation in Phase II.

```
For each $\mathbf{term}_a$ in Terms
    For each doc in $\mathbf{term}_a$.docs
        For each $\mathbf{term}_b$ in doc.terms_in_doc
            If $\mathbf{term}_b$ has not been entered in collection of co-occurrences associated with $\mathbf{term}_a$
                Create Cooccurrence object cooc and insert in $\mathbf{term}_a$.cooccurrences
            Else
                Return reference to cooc associated with $\mathbf{term}_b$
            Increment cooc.sum_intersections
            Increment cooc.sum_min_term_freqs
    For each cooc in $\mathbf{term}_a$.cooccurrences
        Compute cooc.similarity(term$_a$, term$_b$)
    Perform subset operation on $\mathbf{term}_a$.cooccurrences
    Output $\mathbf{term}_a$.similarities
```

The output consists of a one-to-many mapping where each term is associated with a list of related terms ranked by similarity. The complexity of this computation is $O(\mathcal{TDC})$ where $\mathcal{T}$ is the total number of terms, $\mathcal{D}$ is the number of documents in which $\mathcal{T}$ occurs, and $\mathcal{C}$ is the number of co-occurring terms in $\mathcal{D}$.

3.2.3 The Similarity Function

The similarity computation is based on an asymmetric "Cluster Function" developed by Chen and Lynch [10]. The authors show that the asymmetric cluster function represents term association better than the popular cosine function.

$$ClusterWeight(T_j, T_k) = \frac{\sum_{i=1}^{n} d_{jk}^{i}}{\sum_{i=1}^{n} d_{j}^{i}} \times WeightingFactor(T_k)$$

$$ClusterWeight(T_k, T_j) = \frac{\sum_{i=1}^{n} d_{kj}^i}{\sum_{i=1}^{n} d_k^i} \times WeightingFactor(T_j)$$

These two equations indicate the cluster weights, or similarity, from term T_j to term T_k (the first equation) and from term T_k to term T_j (the second equation). d_j^i and d_k^i are the product of term frequency and inverse document frequency and are defined in a similar manner. d_j^i, for example, is defined as

$$d_j^i = tf_j^i \times \log_{10}\left(\frac{N}{df_j} \times w_j\right)$$

where N represents the total number of documents in the collection, tf_j^i is the frequency of occurrence of term T_j in document i, df_j is the number of documents (across the entire collection of N documents) in which term T_j occurs, and w_j is the number of words in term T_j.

d_{jk}^i and d_{kj}^i represent the combined weights of both terms T_j and T_k in document i and are also defined in a similar manner. d_{jk}^i, for example, is defined as follows:

$$d_{jk}^i = tf_{jk}^i \times \log_{10}\left(\frac{N}{df_{jk}} \times w_j\right)$$

Here tf_{jk}^i represents the minimum number of occurrences of term T_j and term T_k in document i. df_{jk} represents the number of documents (in a collection of N documents) in which terms T_j and T_k occur together. The final expression, w_j, is the number of words in term T_j.

In order to penalize general terms which appear in many places in the co-occurrence analysis, the authors develop a weighting scheme similar to the inverse document frequency function. T_j, for example, has the following weighting factor:

$$WeightingFactor(T_j) = \frac{\log_{10} \frac{N}{df_j}}{\log_{10} N}$$

Terms with a higher value for df_j (i.e., more general terms) have a smaller weighting factor, which results in a lower similarity. Co-occurring terms are ranked in decreasing order of similarity, with the result that more general terms occur lower in the list of co-occurring terms.

3.2.4 The Implementation

cSpace is based upon a collection hierarchy derived from the Polaris Project, a research project investigating the automatic parallelization of Fortran codes [8]. The collection hierarchy provides an extensive set of templatized data structures including lists, trees, and sets. The data structures are templatized in the sense that they may contain many different types of objects. Any object derived from the base class *Listable* may be referred to by a collection. Key data structures employed by **cSpace** include red-black balanced binary trees which are used to alphabetically order both terms and co-occurring terms.

3.2.5 The Data Sets

cSpace currently has four input data sets. Each set consists of a list of noun phrases extracted from various sources, including an email-based discussion list on computer programming, abstracts from the Medline database, and abstracts from the Compendex science and engineering collection.

The first of these data sets, dubbed the "Small" set, consists of 300 documents each approximately the size of an abstract. The total size of this data set is 8,945 terms and the source file containing the documents is 639,483 bytes. This input was chosen specifically because this is the number of documents which is considered manageable in a response to a query to a bibliographic database [56]. The output size for this input data set is 2,665,339 bytes.

The second data set consists of 1505 documents and 53,776 terms and is dubbed the "Medium" data set. This data set is representative of a small *personal repository* [57]. The document source file is 3,756,384 bytes in size and the output produced is 15,590,146 in size.

The third data set consists of 8,787 documents and 710,431 terms. This set is dubbed the "Large" data set, and is representative of a collection of personal repositories. The document source file for this input set is 13,375,200 bytes in size and the output produced is 124,702,335 bytes in size.

The fourth and final data set consists of 66,344 documents and 6,082,070 terms. It is dubbed the "XLarge" data set, and is representative of a collection of documents from a particular discipline (e.g., information retrieval). The document source file for this data set is 160,906,227 bytes in size and the output produced is 1,235,206,929 bytes in size.

3.2.6 cSpace and the SPEChpc Suite

The process of establishing applicable benchmarks and characterizing their performance on supercomputer systems is an important step in evaluating language and machine environments which are being used to solve computationally challenging problems. The SPEChpc benchmark suite has been established under the auspices of the Standard Performance Evaluation Corporation (SPEC) in order to accomplish this goal [16].

SPEChpc is defined by a joint effort of industrial members, high-performance computer vendors, and academic institutions. The primary goal is to determine a set of industrially significant applications that can be used to characterize the performance of high-performance computers across a wide range of machine organizations. A secondary goal is to identify a representative workload for high-performance machines which will be made available for scientific study. SPEChpc includes multiple program versions for each benchmark, each targeted at a different class of machine architecture. Multiple data sets are also included for each benchmark. In a departure from other SPEC benchmark suites, SPEChpc specifically permits benchmark tuning within prescribed limits. This philosophy is designed to prevent algorithmic changes while at the same time permit each architecture's capabilities to be used to best advantage.

The **cSpace** application benchmark is currently undergoing evaluation for membership as part of an information retrieval benchmark in the SPEChpc benchmark suite. As such **cSpace** represents an important class of application which is not yet represented in the SPEChpc suite.

3.3 Coalescing Operations in cSpace

Coalescing operations in **cSpace** can be broken down into one of two general categories. The first category includes operations which involve a statistical reduction of select characteristics of the input set. For example, the number of occurrences of a given term in a given document may be tabulated and recorded. The second category includes operations used in reducing a collection of objects by grouping objects together which are related in some way. For example, multiple documents containing references to the same term may be collected into a single data structure associated with that term.

Both of these operations are iterative in nature in that they are executed in loops which perform the operation on numerous objects. Due to the fact that these coalescing loop operators are associative, the loops can be parallelized.

In the following discussion, we often must consider the individual operations which comprise the loop operator. Thus at times we refer to particular coalescing operations without specifically mentioning the loop operators to which these operations belong[1]. The following sections outline the parallelism present in the two phases of **cSpace** discussed in section 3.2.2.

3.3.1 Phase I

The first coalescing operation is a histogram reduction across terms in a document. The variable **term**.term_freq is unique for each term-document pair, and records the number of times **term** occurs in **doc**. The second operation is also a histogram reduction, but this time across all documents in the collection. This sum is recorded in the variable **term**.sum_term_freq, and is unique for each term.

[1]The actual loop operator can be determined from the context of the discussion by considering the pseudo-code of the algorithm.

The third operation is a histogram reduction which records the occurrence of each term once per document. The variable **term**.doc_freq stores this value, and is unique for each term. These three reductions correspond to the three increment operations in the pseudo-code for Phase I (repeated below). All three reductions are based on addition (+), and can be parallelized due to the fact that the increment operation is an example of an associative coalescing operator.

```
For each input document
    Create Doc object doc and insert in global Docs collection
    Extract noun phrases from doc
    For each noun phrase in doc
        If phrase has not been entered in global Terms collection
            Create Term object term and insert in global Terms collection
        Else
            Return reference to term associated with phrase
        Increment term.sum_term_freq
        Increment term.term_freq for doc
        If this is the first occurrence of noun phrase in doc
            Increment term.doc_freq
            Insert reference to doc in term.docs
            Append reference to term to doc.terms_in_doc
```

Several coalescing operations remain which have not been discussed. In all cases these operations involve the collection of objects into various data structures. One of the data structures employed by **cSpace** consists of a one-to-one mapping of keys to objects. When an object is inserted into such a structure, it will overwrite an existing object with the same key. Thus, if keys are not unique, the operation is not commutative. To understand this point, consider the case where a given term T_A occurs at least twice in a collection which will be indexed alphabetically by term. If the computation is executed in parallel, T_A will be inserted twice into the collection. However, the resulting race condition will leave the collection in an indeterminate state: one of the objects will be stored, and the other overwritten.

Despite this fact, in both cases involving insertion into an indexed data structure in Phase I, keys are guaranteed to be unique by the coalescing loop operator represented by the loop body. The If/Else conditional control flow construct in the above pseudo-code, for example, guarantees that keys are unique and thus that term insertion is an unordered associative coalescing operation. Similarly, the insertion of document references into term objects is an unordered associative coalescing operation guaranteed by the uniqueness of the address of the document object being referenced.

The final operation in Phase I involves a computation which, although associative, is not commutative. It is summarized by the pseudo-code "Append reference to **term** to **doc**.terms_in_doc" above. To understand this point, consider a simple example involving the lisp **append** operator drawn from Chapter 2:

```
append(append([1] [2]) [3])
    ⇒ append([1 2] [3])
    ⇒ [1 2 3]
append([1] append([2] [3]))
    ⇒ append([1] [2 3])
    ⇒ [1 2 3]
```

Here we are making a simple list of the numbers 1, 2, and 3. The append operator takes two operands which are lists and creates a new list by appending the second operand to the first. In the first case above, the list [2] is first appended to the list [1], resulting in the list [1 2]. The list [3] is then appended to this list, resulting in the final list [1 2 3]. In the second case, the list [3] is first appended to the list [2], resulting in the list [2 3]. This list is then appended to the list [1], resulting in the same final list. The final result is identical in both cases even though the associative order of the operands differ.

However, as was demonstrated in Chapter 2, if we now consider a case where we attempt to commute the terms, the results will differ:

```
append([1] [2])
    ⇒ [1 2]
append([2] [1])
    ⇒ [2 1]
```

Clearly the append operator is not commutative. As presented in Chapter 2, section 2.3.4, loops containing coalescing operations of this nature must be parallelized based on $\mathcal{L}^t$.

3.3.2 Phase II

Phase II of the computation initially involves two reductions across the data set, both of which occur in associative coalescing loops. The first is a histogram reduction across all terms which co-occur with a given term. In the pseudo-code below, the statement "Increment **cooc**.sum_intersections" records the number of times that $\mathbf{term}_a$ occurs together with $\mathbf{term}_b$. The second operation is likewise a histogram reduction which sums the minimum term frequencies of co-occurring terms across the entire collection. This reduction is summed in the variable **cooc**.sum_min_term_freqs, and is unique to each co-occurring pair of terms. Similar to the histogram reductions in Phase I, these reductions also have their basis in an associative coalescing operation based on the increment operator, and can be parallelized as a result.

```
For each $\mathbf{term}_a$ in Terms
    For each doc in $\mathbf{term}_a$.docs
        For each $\mathbf{term}_b$ in doc.terms_in_doc
            If $\mathbf{term}_b$ has not been entered in collection of co-occurrences associated with $\mathbf{term}_a$
                Create Cooccurrence object cooc and insert in $\mathbf{term}_a$.cooccurrences
            Else
                Return reference to cooc associated with $\mathbf{term}_b$
            Increment cooc.sum_intersections
            Increment cooc.sum_min_term_freqs
    For each cooc in $\mathbf{term}_a$.cooccurrences
        Compute cooc.similarity($\mathrm{term}_a$, $\mathrm{term}_b$)
    Perform subset operation on $\mathbf{term}_a$.cooccurrences
    Output $\mathbf{term}_a$.similarities
```

Several additional associative coalescing operations are performed in Phase II. For example, one such operation takes place in the subset computation during execution of the statement "Perform subset operation on $\mathbf{term}_a$.cooccurrences". This operation involves a simple heuristic designed to increase the precision of the resulting list of co-occurring terms. The computation involves the traversal of the list of co-occurring terms. Terms are selected for comparison based on the actual number of words which make up the term; if a term is made up of two words, for example, and occurs as an initial substring of a three-word term with a lower similarity, the two-word term will be removed from the list of co-occurring terms. This operation effectively reduces "noise" from the resulting list of co-occurring terms.

The process of traversing a list and conditionally removing entries is an operation which occurs in other computer programs as well. For example, in the irregular Fortran benchmark DSMC3D (Discrete Simulation Monte Carlo), a similar operation is performed [3]. Instead of terms, however, the list contains molecules. Both of these operations can be modeled as coalescing operations in which the list is being reduced in size. In [3], we reduce sections of the list in parallel, and the actual recombination of the reduced sections takes place in a `dofinal` section which follows the doall execution of the `doevery` portion of the loop. Thus, within the framework of coalescing loop operators, *list reduction* is associative and can be parallelized.

Similar to term insertion in Phase I, the insertion of co-occurrence objects into the co-occurrence data structure is an unordered associative coalescing operation. The final operation in Phase II that we will discuss is an append similar to that described previously in section 3.3.1. In this case, the operation takes place during disk file output. When the statement "Output $\mathbf{term}_a$.similarities" in the above pseudo-code is executed, terms and their co-occurring (related) terms are output to a sequential disk file. This operation is a list append, and the outermost loop in Phase II can thus be parallelized based on transformation $\mathcal{L}^t$ presented in Chapter 2.

3.4 Implementing the Parallelism in cSpace

In determining how to take best advantage of the loop-level parallelism in computer programs in light of the model we have presented in Chapter 2, two important factors must be considered:

- Coalescing Loop Operator Associativity
- Loop Execution Overhead

3.4.1 Coalescing Loop Operator Associativity

In sections 3.3.1 and 3.3.2, multiple operations were discussed without explicitly tying operations to specific loops. This is due to the fact that multiple levels of parallelism exist across loops in the program. Term insertion, for example, is an associative coalescing operation in both the outermost loop which iterates across the input document set and the inner loop which iterates across noun phrases in a given document. In fact, all of the coalescing operations discussed in section 3.3 are associative in their respective enclosing loops.

3.4.2 Loop Execution Overhead

As noted in section 3.2.2, Phase I accounts for less than 5% of the sequential execution time and Phase II for the remaining 95% of the execution time. When manually parallelizing a computer program, the overhead of implementing parallelism must be considered. Sources of overhead include the startup cost of spawning parallel processes and synchronization of access to global data structures. As a result, there is a trade-off between the available parallelism and a given architecture's ability to take advantage of that parallelism. For the experimental results reported in this chapter, Phase II was parallelized and Phase I executed serially. This choice was made purely as a practical matter based on their relative proportions of the sequential execution time.

3.4.3 Implementing the Parallelism in Phase II

At a high level Phase II of the **cSpace** application consists of the following loops:

```
For each $\mathbf{term}_a$ in Terms
    For each doc in $\mathbf{term}_a$.docs
        For each $\mathbf{term}_b$ in doc.terms_in_doc
            ...
```

As mentioned above, these loops perform associative coalescing operations and can be parallelized at any level. In fact, ample parallelism exists in this phase to support parallel execution of multiple loops simultaneously. However, our implementation was done on an architecture which supports only a single level of parallelism. As a result, we chose to parallelize the outermost loop.

Several techniques are needed in order to accomplish the parallelization of this loop. As an example of these techniques, in the following section we outline the steps necessary to perform output in parallel. The specific techniques used to parallelize output are applicable generally in computer programs that perform output. As noted in Chapter 2, these techniques can also be applied in the automatic parallelization of computer programs in systems such as the Polaris restructurer [8].

3.4.3.1 Parallel Output Operations

As discussed in section 3.3.2, the output operation is associative but not commutative. As a result, the parallelizing transformation must retain the original non-commuted order of execution. As outlined in Chapter 2, four steps are needed in order to accomplish this: privatization of shared variables; initialization of private variables; block scheduling; and cross-processor reduction.

Privatization refers to the creation of thread or process-private copies of global variables[2]. The privatization of the global output stream was accomplished by creating multiple thread-private output streams, one for each processor participating in the computation.

The second step in the transformation is the initialization of the newly created private variables. As discussed in Chapter 2 however, this was unnecessary due to the fact that the output stream pointer is not a recurrent variable.

The third step involves the determination of how iterations of the loop are to be distributed to the processors participating in the computation. Many possible processor schedules may be employed; however, the schedule must insure that the original, non-commuted order of execution is retained. In our case we employed a *block* schedule. In a block schedule, each processor is given a contiguous block of iterations which are executed in the original serial order. This insures that the execution of iterations of the loop are not interleaved (i.e., are not commuted). The block schedule was presented in detail in Chapter 2, section 2.3.4.

The final step is a cross-processor reduction. This involves the serial execution of the associative coalescing loop operator with each of the privatized variables in turn. In the case of the output operation,

[2]This was not necessary in our example of the list append operation in section 3.3.1 for the simple reason that the append example was written in the functional language lisp, and did not employ any global shared variables

this involved an append of the process-private output files to the final result file. This operation also preserved the original order of execution and thus insured that the operands were not commuted.

As discussed in Chapter 2, the first two steps take place in the `dofirst` section of $\mathcal{L}^t$. The body of the loop is executed in the third step as a *doall* loop in the `doevery` section of $\mathcal{L}^t$. The fourth step in which the private output files are serially appended takes place in the `dofinal` section of code. This was performed manually in our experiments.

The bulk of the source code for **cSpace** is contained in Appendix A.

3.5 Characterizing the Performance of cSpace

The shared-memory multi-processor employed in our experiments is an SGI Power Challenge. The Power Challenge is a bus-based shared-memory cache-coherent NUMA (non-uniform-memory-access) multi-processor. The particular machine used in the experiments described in this chapter is a 16-processor model with 4GB of RAM. The 64-bit processors are based on the MIPS R10000 CPU and R10010 FPU clocked at 194MHz. Primary data and instruction caches are 32KB in size, with a 2MB unified secondary cache. The translation lookaside buffer (TLB) has a 128 entry capacity. Peak bandwidth on the 576-bit bus is 1.2GB/second.

	num *procs*	*Small* $m:s$	*Small* S_p	*Medium* $m:s$	*Medium* S_p	*Large* $h:m:s$	*Large* S_p	*XLarge* $h:m:s$	*XLarge* S_p
Serial	1	0:28	1	4:07	1	1:17:19	1	10:30:27	1
Parallel	2	0:15	1.87	1:50	2.25	0:27:16	2.84	3:32:49	2.96
	4	0:10	2.80	1:02	3.98	0:14:28	5.34	1:53:52	5.54
	8	0:07	4.00	0:40	6.18	0:08:41	8.90	1:08:20	9.23
	16	0:06	4.67	0:35	7.06	0:05:44	13.49	0:39:22	16.01

Table 3.1: Wall-clock Execution Times and Speedups for **cSpace**

Table 3.1 summarizes the performance and scalability of **cSpace** across all data sets. For each data set, the serial version of **cSpace** was executed on one processor, and the parallelized version on 2, 4, 8, and 16 processors in order to determine the scalability of the application. The reported execution times are elapsed (wall-clock) times in hours, minutes, and seconds. The S_p columns report speedup for

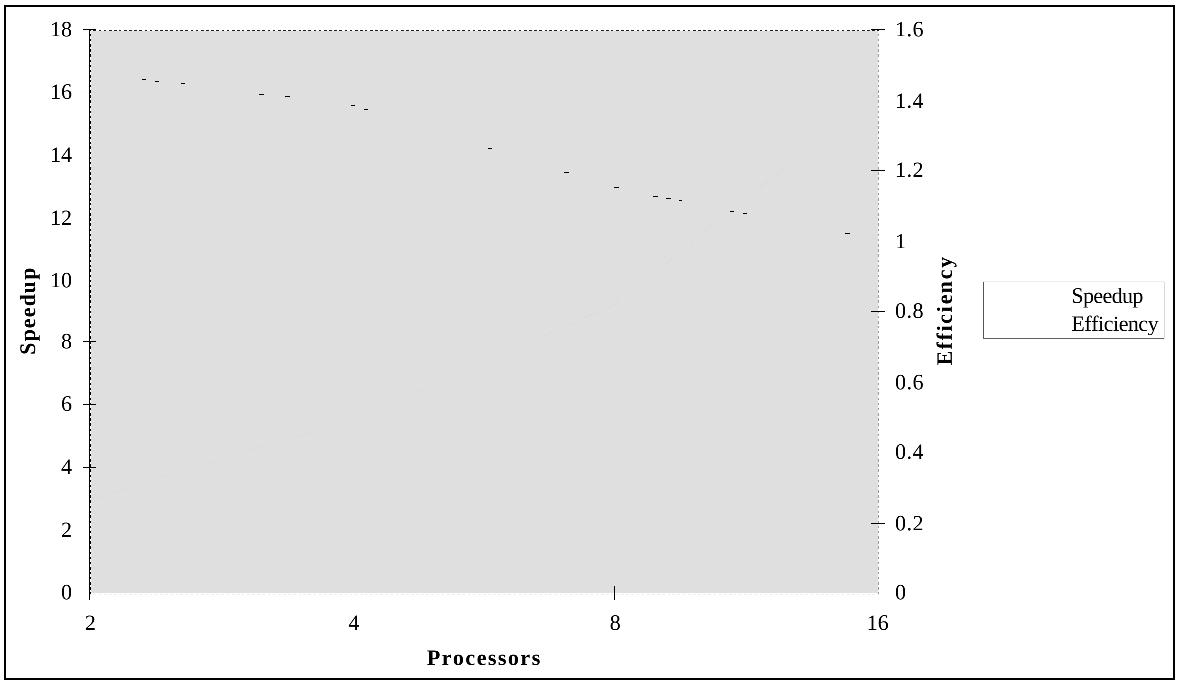

Figure 3.2: Speedup and Efficiency for **cSpace** XLarge Data Set

the given execution. Figure 3.2 portrays speedup and efficiency graphically for the largest data set. All experiments were conducted on a dedicated machine in single-user mode.

Several interesting trends are revealed in Table 3.1. First, several runs resulted in super-linear speedups. This is an indirect result of the poor performance of multi-threaded dynamic memory allocation in C^{++} on the SGI Power Challenge [46]. The parallel version of **cSpace** used in these experiments employs a customized memory manager which alleviates much of the overhead associated with multi-threaded dynamic memory allocation. However, this also provides an unexpected benefit in that the overhead of numerous calls to malloc (i.e., *operator new*) is entirely eliminated. As a result, for example, the parallel version of **cSpace** which employs the customized memory manager executes approximately 35% faster than the serial version when both are executed on a single processor using the Large input set.

Figure 3.2 confirms that **cSpace** scales well up to 16 processors with the given input sets. The efficiency of the computation is calculated as S_p/p where p is the number of processors participating in the computation. As can be seen from the figure, **cSpace** achieved 100% efficiency for all runs made using the XLarge data set.

3.6 Conclusion

In Chapter 2 we presented a model for determining parallelism in computer programs based on the concept of associativity in coalescing loop operators. In this chapter, we applied the model to a modern information retrieval application, **cSpace**, and demonstrated its validity by characterizing the performance of **cSpace** on a late-model, shared-memory multi-processor.

This work represents a melding of several fields in computational science: research in parallelizing compilation technology, research in high-performance benchmarking, and research into the parallelization of National Challenge digital library applications. As such, it represents a significant step towards the realization of the goals outlined in the National Research Council report *Computing The Future* [58].

In the following chapters we will turn our attention to the parallelization of scientific Fortran and hybrid Fortran/C codes. However, as will be seen, the concepts presented in Chapters 2 and 3 carry over into this realm as well.

CHAPTER 4

Techniques for Solving Recurrences

4.1 Introduction

Extensive analysis of applications from a number of benchmark suites has revealed the presence of many loops in which recurrences prevent *doall* [25] parallelization.

During a recent review of the Polaris restructurer [8] approximately 500 loops from programs in the aforementioned suites were identified as serial. Although the current Polaris parallelizer is able to solve a wide variety of recurrences involving induction variables and reductions, approximately 35% of these loops involve an explicitly coded recurrence, reduction, or induction for which Polaris is unable to determine a parallel form. Approximately 70% of the loops in this 35% subset were determined to be partially or fully parallelizable based on a manual inspection of the codes.

This chapter extends the solution techniques discussed in [47] to include additional compile-time and run-time techniques for solving recurrences in parallel. In the following sections we present an overview of the benchmark codes on which we have based our analysis, and then move on to detail the various techniques which have proven effective in parallelizing programs containing recurrences.

4.2 The Benchmark Suite

As mentioned in the Introduction to this chapter, the evaluation of the Polaris restructurer involved benchmarks from several suites. In this section we provide a brief overview of these suites which include Grand Challenge codes from the National Center for Supercomputing Applications (NCSA), codes from the Standard Performance Evaluation Corporation (SPEC CFP95), and codes from the Perfect Club [5]. Table 4.1 gives a brief synopsis of each code, including its origin, the number of lines of code, and the serial execution time on a R4400-based SGI Challenge. In the course of our investigation of various solutions to recurrences presented in this chapter, we have manually analyzed patterns in several of these benchmarks.

Program	Description	Origin	Lines	Serial exec (seconds)
APPLU	Parabolic/elliptic PDE solver.	SPEC	3870	1203
APPSP	Gaussian elimination system solver.	SPEC	4439	1241
ARC2D	Implicit finite-difference code for fluid flow.	PERFECT	4694	215
BDNA	Molecular dynamics simulation of biomolecules.	PERFECT	4887	56
CMHOG	3D ideal gas dynamics code.	NCSA	11826	2333
CLOUD3D	3D model for atmospheric convective applications.	NCSA	9813	20404
FLO52	2D analysis of transonic flow past an airfoil.	PERFECT	2370	38
HYDRO2D	Navier Stokes solver to calculate galactical jets.	SPEC	4292	1474
MDG	Molecular dynamics model for water molecules.	PERFECT	1430	178
OCEAN	2D solver for Boussinesq fluid layer.	PERFECT	3288	118
SU2COR	Quantum mechanics with Monte Carlo simulation.	SPEC	2332	779
SWIM	Finite difference solver of shallow water equations.	SPEC	429	1106
TFFT2	Collection of FFT routines from NASA codes.	SPEC	642	946
TOMCATV	Generates 2D meshes around geometric domains.	SPEC	190	1327
TRFD	Kernel for quantum mechanics calculations.	PERFECT	580	20
WAVE5	Solves particle and Maxwell's equations.	SPEC	7764	788

Table 4.1: Benchmark Codes

4.3 General Techniques for the Parallel Solution of Recurrences

This section presents an overview of parallelizing techniques for solving recurrences. Each technique in the following list will be briefly discussed and exemplified.

- Symbolic Computation of Closed-Forms

- Intrinsic Minimum & Maximum Reductions
- Semi-private Transformation
- Wrap-around Privatization
- Do-pipe Parallelization
- Multi-level Parallel Execution

4.3.1 Symbolic Computation of Closed-Forms

[47] discusses a variety of techniques employed in the solution of reductions and inductions. These techniques include, for example, the use of computer algebra in the determination of closed-forms for scalar induction variables. Techniques of this nature can be extended to the solution of linear recurrences.

Consider the following example:

```
$a(0) = 0$
do $i = 1, n$
        $a(i) = a(i-1) + 1$
enddo
```

Here we have a recurrence involving the array a. Techniques for solving linear non-homogeneous recurrences of this nature are well known [38]. The closed-form for this recurrence is $a(i) = i$, resulting in the following parallel form:

```
$a(0) = 0$
doall $i = 1, n$
        $a(i) = i$
enddo
```

This transformation has been found useful in the NCSA Grand Challenge code cmhog.

A second pattern determined to be of importance involves induction variables which have discontinuities in the sequence of values which they compute. For example:

```
$k = 0$
do $i = 1, n$
```

```
          $k = k + 1$
          $if(k.eq.m)$
                    $k = 0$
          use of $k$
enddo
```

Assuming $m < n$, the scalar induction variable k takes on the values $[1, m]$ n/m times, resulting in n/m discontinuities in its sequence of values. The solution of this pattern employs the *mod* operator in the closed-form:

```
$k = 0$
doall $i = 1, n$
          $k = mod(i, m)$
          use of $k$
enddo
```

A third pattern which has arisen in a suite of sparse and irregular codes discussed in Chapter 5 involves conditionally incremented induction variables. The following exemplifies this pattern:

```
$m = 0$
do $i = 1, n$
          if $(m.lt.maxm)$ then
                    $m = m + 1$
                    $pp(1, m) = \ldots$
          endif
enddo
```

This particular pattern can be easily transformed into a parallel form. The following steps are involved in the transformation:

- Determine the closed-form for the induction on m ignoring the conditional guard
- Privatize m and assign the closed-form to the private version at the loop header
- Substitute the privatized copy of m for all uses of m in the body of the loop
- Assign the last value $m = min(maxm, n)$ if m is live-out

The transformed code looks like this:

```
do $i = 1, n$
        $m_p = i - 1$
        if $(m_p.lt.maxm)$ then
                $pp(1, m_p) = \ldots$
        endif
enddo
$m = min(maxm, n)$
```

The closed-form of m is $i - 1$ at the header of the loop. The variable m_p is a loop-private copy of the induction variable m. The original statement $m = m + 1$ has been deadcoded and all remaining uses of m have been substituted by m_p. Finally, the last value of m is assigned at the loop exit. In order to simplify the presentation, the example assumes no zero-trip loops [44]. However, the technique is applicable in the presence of zero-trip loops with a proper guard on the last value assignment of m.

4.3.2 Intrinsic Minimum & Maximum Reductions

The recognition of reductions of the general form:

$$A(\alpha_1, \alpha_2, \ldots) = A(\alpha_1, \alpha_2, \ldots) + \beta$$

is discussed in [47]. Here β represents an arbitrary expression and A may be a multi-dimensional array with subscript vector $\{\alpha_1, \alpha_2, \ldots\}$ which may contain both loop-variant and invariant terms. Neither α_i nor β may contain a reference to A, and A must not be referenced elsewhere in the loop outside other reduction statements. Of course $\{\alpha_1, \alpha_2, \ldots\}$ may be null (i.e., A is a scalar variable).

These techniques have been implemented in a recognizer in the Polaris restructurer. However, these techniques can be readily extended to include the recognition of calls to intrinsic *min* and *max* functions. Once such intrinsics have been recognized, they can be automatically parallelized based on techniques described in [44]. In section 6.3 of Chapter 6 we discuss the parallelization of an intrinsic max reduction of this nature.

4.3.3 Semi-private Transformation

In the following example loop-carried dependences on the scalar a prevent doall parallelization, and the loop-variant expression $b(i)$ precludes a closed-form solution of the recurrence. In addition, the use of a in the loop outside the reduction statement prevents use of a parallelizing reduction transformation such as described in [44].

```
do $i = 1, n$
        $a = a + b(i)$
        $\ldots = \ldots a \ldots$
enddo
```

The traditional approach to solving this recurrence involves expansion and loop distribution as follows:

```
$a(0) = 0$
do $i = 1, n$
        $a(i) = a(i-1) + b(i)$
enddo
doall $i = 1, n$
        $\ldots = \ldots a(i) \ldots$
enddo
```

A second, run-time solution is the doacross loop [13]:

```
post(1)
doacross $i = 1, n$
P:      wait($i$)
Q:      $a = a + b(i)$
S:      $\ldots = \ldots a \ldots$
T:      post($i + 1$)
enddo
```

Access to the variable a in statement S creates a loop-carried anti-dependence which can be resolved by privatizing a as follows:

```
post(1)
doacross $i = 1, n$
P:      wait($i$)
Q:      $a = a + b(i)$
R:      $a_{private} = a$
T:      post($i + 1$)
S:      $\ldots = \ldots a_{private} \ldots$
enddo
```

The post operation has been floated up past the use of a in statement S, thereby allowing doacross computation to proceed with the next iteration. The privatization is from statement R onward, and a is termed a *semi-private* variable.

This transformation is generally applicable wherever expansion and loop distribution [68] are used. We have found this technique of importance in the CFP SPEC95 code su2cor. In the CFP SPEC95 code tomcatv a similar transformation was implemented to *partially privatize* arrays, thereby breaking loop-carried anti-dependences.

4.3.4 Wrap-around Privatization

Wrap-around privatization involves the privatization of variables through partial peeling of the first and final iterations of a loop. Consider the following example:

```
do $i = 1, n$
        $\cdots = \ldots a \ldots$
        *body*
        $a = \cdots$
enddo
```

The loop-carried flow dependence on a can be resolved as follows:

```
$\cdots = \ldots a \ldots$
*body*
doall $i = 1, n - 1$
        $a = \cdots$
        $\cdots = \ldots a \ldots$
        *body*
        enddo
        $a = \cdots$
```

In essence, one complete iteration is peeled, with part of the iteration peeled into the loop prologue, and the remainder peeled into the epilogue. The scalar a above is now privatizable, and the resulting loop executes the remaining $n-1$ iterations as a doall loop.

In the SPEC CFP95 benchmark applu several loop-carried dependences exist in the main time-stepping loop in subroutine *ssor*. However, an important dependence involving the array *rsd* can be broken through the application of *wrap-around privatization*. This same technique can be applied in the SPEC CFP95 benchmark turb3d.

4.3.5 Do-pipe Parallelization

In the following example, the assignment $a(0) = a(n)$ prevents doall parallelization of the outer i loop. However, a speedup can be achieved by *pipelining* the outer i loop [43]. The following exemplifies this transformation:

```
$a(0) = 0$
do $i = 1, n$
    do $j = 1, n$
        $a(j) = a(j-1) + 1$
    enddo
    $a(0) = a(n)$
    do $k = 1, n$
        $\cdots = \ldots a(k) \ldots$
    enddo
enddo
```

```
$process_1$
    $a(0) = 0$
    do $i = 1, n$
        do $j = 1, n$
            $a(j) = a(j-1) + 1$
        enddo
        $a(0) = a(n)$
```

```
                $a_{private} = a$
                post($i$)
        enddo
$process_2$
        do $i = 1, n$
                wait($i$)
                do $k = 1, n$
                        $\cdots = \ldots a_{private}(k) \ldots$
                enddo
        enddo
```

The outer loop has been broken into a 2-stage pipeline which is executed on two different processors. The synchronization on a enforces flow-dependences, thereby allowing the computation to proceed in parallel.

Several codes in our test suite are amenable to the *pipelining* transformation. In all cases these involve outermost time-stepping loops which cannot be solved as efficiently with other techniques. Although the pipelining technique has been known for some time, no empirical study has decisively demonstrated the effectiveness of this technique on actual application codes.

4.3.6 Multi-level Parallel Execution

Parallelism within loops can be loosely characterized as either doall [25], doacross [25], or functional in nature. Loop pipelining, as discussed in the previous section, is an example of functional parallelism.

Functional parallelism of various sorts has been a topic of study for some time ([21, 64, 23, 41]). However, to date no empirical study has been conducted with a set of real programs to determine if functional parallelism can be effectively combined with doall or doacross parallelism. In Chapter 6, we characterize the performance of multi-level parallelism by combining the doall, doacross, and do-pipe transformations.

4.4 Techniques for Recognition and Solution of Recurrences in Sparse & Irregular Codes

In this section we present an overview of parallelizing techniques for solving recurrences in sparse and irregular codes. Each technique in the following list will be discussed and exemplified.

- Histogram Reductions
- Random Number Generator Substitution
- Proving Monotonicity of Index Arrays
- Combined Static & Run-time Analysis of Induction Variable Ranges
- Copy-in and Copy-out
- Loops with Conditional Exits

4.4.1 Histogram Reductions

The following code portrays a reduction on the array A which involves a loop-variant subscript function $f(i, j)$.

```
 do i=1,n
   do j = 1, m
      k = f(i,j)
      a(k) = a(k) + expression
   enddo
enddo
```

Due to the loop-variant nature of the subscript function f, loop-carried dependences may be present at run-time. This pattern occurs commonly in many codes, both sparse and non-sparse, and is termed a *histogram* reduction [44, 31].

The parallelizing transformation takes one of three forms: *critical section*, *privatized*, or *expanded.* Each approach is discussed and exemplified below. As before, the language used in the examples is based on IBM's Parallel Fortran [29].

- Critical Section

The first approach involves the insertion of synchronization primitives around each reduction statement, making the sum operation atomic. In our example the reduction statement would be enclosed by a lock/unlock pair:

```
 parallel loop i=1,n
   do j = 1, m
      k = f(i,j)
      call lock e(k)
         a(k) = a(k) + expression
      call unlock e(k)
   enddo
 enddo
```

This is an elegant solution on architectures which provide fast synchronization primitives.

- Privatized

In *privatized* reductions, duplicates of the reduction variable that are private to each processor are created and used in the reduction statements in place of the original variable. The following code exemplifies this transformation:

```
 parallel loop i=1,n
   private a_p(sz)
   dofirst
     a_p(1:sz) = 0
   doevery
     do j = 1, m
        k = f(i,j)
        a_p(k) = a_p(k) + expression
     enddo
   dofinal lock
     a(1:sz) = a(1:sz) + a_p(1:sz)
enddo
```

Each processor executes the `dofirst` section of the parallel loop once at the beginning of its slice of the iteration space. The `doevery` section of the loop is executed every iteration. The `dofinal` section of the code is executed once by each processor after completion of its slice of the iteration space. The `lock` argument to `dofinal` indicates that the code be enclosed in a critical section.

- Expanded

The third approach employs expansion to accomplish the same functionality as the privatizing transformation. Rather than allocating loop-private copies of the reduction variable, the variable is expanded by the number of threads participating in the computation. All reduction variables are replaced by references to this new, global array, and the newly created dimension is indexed by the processor number executing the current iteration.

The initialization and cross-processor sums take place in separate loops preceding and following the original loop, respectively. In this approach there is no need for synchronization, and both loops can be executed in parallel:

```
global a_e(n,threads)
parallel loop j=1,threads
   do i=1,n
      a_e(i,j) = 0
   enddo
enddo
parallel loop i = 1, n
private tid = thread_id()
   do j = 1, m
      k = f(i,j)
      a_e(k,tid) = a_e(k,tid) + expression
   enddo
enddo
parallel loop i=1,n
   do j=1,threads
      a(i) = a(i) + a_e(i,j)
   enddo
enddo
```

In our study of the benchmark suite discussed in Chapter 5, we have found that histogram reductions occur in key computational loops in all four of the benchmarks derived from the HPF-2 motivating suite: NBFC, CHOLESKY, DSMC3D, and EULER. The parallelization of histogram reductions is based on a run-time technique which depends on the associativity of the operation being performed. The Polaris parallelizing restructurer recognizes histogram reductions based on the techniques discussed in [44]. A more detailed study which compares and contrasts the performance of these three transformations is contained in [31].

4.4.2 Random Number Generator Substitution

One approach to breaking dependences caused by calls to pseudo-random number generators is to substitute thread-parallel generators which produce a robust stream of random numbers. Recent work by Bailey and Mascagni involves the development and standardization of thread-parallel pseudo-random number generators based on lagged Fibonacci series [48, 39].

As discussed in Chapter 2, the substitution of robust thread-parallel pseudo-random number generators can be considered to enable the parallel execution of loops based on the associativity of the operation. In our study of the benchmark suite discussed in Chapter 5 as well as other benchmark suites, we have come across several cases involving RNGs:

- Calls to Single-threaded Library Routines
- Calls to Single-threaded User Routines
  - Linear Congruential Generators
  - Lagged Fibonacci Generators

4.4.2.1 Single-threaded Library Routines

We have determined that calls to RNG library routines occur in two computationally important loops in a test suite used in the evaluation of the FALCON MATLAB compiler [14]. In addition, random() is called in INITIA_DO2000 in the Perfect Club benchmark MDG [5]. The CHOLESKY benchmark in our sparse & irregular suite also calls the rand() library routine in an important loop. In three of these cases, the RNG call is the only factor preventing parallelization of the loop after application of the techniques implemented in the current Polaris restructurer. In CHOLESKY, compile-time analysis of the array access pattern reveals a straightforward test which is sufficient to prove independence if the random number generation can be parallelized (see section 4.4.3 below). In each of these cases, the loop-carried dependence can be broken by replacing these calls with a thread-parallel RNG.

4.4.2.2 User Implemented Single-threaded RNGs

Several codes in well-known test suites contain implementations of RNGs of various types. The Perfect Club benchmark QCD, for example, contains the routines PRANF, LADD, and LMULT which together implement a *linear congruential* pseudo-random number generator [36]. Similarly, the DSMC3D benchmark in our sparse & irregular suite implements a linear congruential generator based on work described in [37]. A third example occurs in the SPEC CFP95 benchmark su2cor which implements a lagged Fibonacci generator as described previously in this section.

Lagged Fibonacci generators such as that implemented in su2cor take the form of a homogeneous linear recurrence. Such relations can be automatically detected using pattern recognition techniques [1]. General techniques for solving linear recurrences of this type are well known [38], and closed-forms for such recurrences can be computed at compile-time as discussed in section 4.3.1 above. An example of the closed-form solution of such a generator from the SPEC CFP95 suite is discussed in Chapter 6, section 6.2.2.

4.4.3 Proving Monotonicity of Index Arrays

One of the major difficulties in automatically parallelizing sparse codes involves the analysis of subscripted array subscripts. The use of subscripted subscripts normally causes data dependence tests to draw overly conservative conclusions. The following portrays an example of such patterns:

```
do i = 1, n
 k = ia(i)
 a(k) = ...
end do
```

In the general case the subscript array ia must contain distinct indices if the outermost i loop is to be executed as a doall loop. Another pattern which occurs commonly in our suite involves the use of subscripted array subscripts in loop bounds:

```
do i = 1, n
 do j = ia(i), ia(i+1)-1
  a(j) = ...
 end do
end do
```

To parallelize the outermost loop in this case, the range $[ia(i), ia(i+1)-1]$ must be non-overlapping for all i. Although this condition may not hold generally, we have found that the index arrays in several of our sparse codes are monotonic in nature. This is due to the fact that matrices in sparse codes are often represented in a row-wise or column-wise format in which the values of non-zero elements of the matrix are stored in a one dimensional array and pointers to the first and last elements of each row or column are stored in an index array. When this representation is used, the index array is non-decreasing.

The analysis of such access patterns has been considered difficult to accomplish at compile-time. However, using a combination of static, compile-time analysis and simple run-time tests, it is possible to prove that these index arrays are non-decreasing. In the CHOLESKY benchmark discussed in Chapter 5, for example, it can be statically proven that the index array isu (initialized in SPARCHOL_GOTO290) satisfies the somewhat stronger strictly-increasing condition. This proof is possible due to the fact that the values assigned to isu are taken from a loop induction variable which is unconditionally incremented in the body of loop 290. This in turn is sufficient to prove that SPARCHOL_DO1017, a loop which uses isu in the bounds of an inner loop (as exemplified in the latter pattern above), can be executed as a doall loop.

In cases where the index array is read from input, the test for monotonicity can be done as the index array is initialized. If the test cannot be performed on input, the overhead is still quite small. This is due to the fact that the data representation employed in the codes studied in our suite guarantees that $n \equiv m + 1$, where n is the size of the index array and m is the number of columns or rows. In practice $n << \alpha$, where α is the number of non-zero entries in the matrix. As a result, the cost of testing the condition `ia(i+1).ge.ia(i)` for `i = 1,n` is insignificant.

When possible, this test should be inserted as part of the initial input operation. However, since this loop is essentially a reduction across ia, it can also be executed in parallel. Using techniques for handling loops with conditional exits discussed in section 4.4.6, the loop execution time may be decreased even further.

We have found this pattern occurs in key computational loops in codes in the test suite discussed in Chapter 5.

4.4.4 Combined Static & Run-time Analysis of Induction Variable Ranges

We have determined cases which require run-time assistance in the resolution of loop-carried dependences involving induction variables. The following exemplifies one such pattern:

```
$m = m_{iv}$
do $i = k(j), k(j+1)$
        $a(m) = a(i)$
        $m = m + 1$
enddo
```

This example is representative of copy operations within a workspace. In this particular case, a closed-form can be computed at compile time for the induction variable m. The resulting closed-form range of uses of m is $[m_{iv}, m_{iv} + (k(j+1) - k(j))]$ given that $k(j+1) \geq k(j) - 1$[1]. This loop can be executed as a doall parallel loop if one of two conditions is satisfied: either $m_{iv} > k(j+1)$ or $m_{iv} + (k(j+1) - k(j)) \leq k(j+1)$.

In the first case, the lower bound of the range of m is greater than the largest value taken by the loop index i, and the two ranges do not overlap. In the second case, if $m_{iv} + (k(j+1) - k(j))$ is also $< k(j)$, the upper bound of m is less than the lower bound of i and again the ranges do not overlap. If, on the other hand, $m_{iv} + (k(j+1) - k(j)) = k(j+1)$, then $m_{iv} = k(j)$. This in turn implies that $m = i$ is a loop-invariant condition and no loop-carried dependences hinder parallel execution. Finally, if $k(j) \leq m_{iv} + (k(j+1) - k(j)) < k(j+1)$, the loop can be parallelized by creating a read-only copy of the range $a(k(j) : m_{iv} + (k(j+1) - k(j)))$ and substituting reads of this copy for reads of the original array. This has the effect of removing anti-dependences via variable renaming. The remaining case in

[1]This is the *exact zero-trip* test discussed in [44]

which $m_{iv} + (k(j+1) - k(j)) > k(j+1)$ and $m_{iv} \le k(j+1)$ contains true loop-carried flow dependences and cannot be executed in a doall fashion.

A second example requiring similar analysis combined with the run-time estimation of the range of an induction variable is discussed in Chapter 5, section 5.3.8.

4.4.5 Copy-in and Copy-out

It is often necessary to break loop-carried anti and output dependences by *privatizing* both scalar and array variables which are defined and used within a single iteration [62]. However, many such variables have initial values which must be copied into each processor's local copy of the variable prior to the start of parallel execution. This process is known as "copy-in".

Corresponding to copy-in is an operation in which variables' values are copied out on the final iteration of each processor's slice of the iteration space. Termed "copy-out", this transformation is necessary whenever local variables have a *last value* which is used outside the parallel region.

The following example exhibits a case where copy-in is necessary:

```
do $i = 1, n$
        ...
        do $j = 2, m$
                $a(j) = \dots a(j-1) \dots$
                $\dots = \dots a(j) \dots$
        enddo
        ...
enddo
```

In this case, a(1) is read first and never written (although never being written is not a requirement). If there is nothing else which precludes parallelization of the outer loop, $a(1:m)$ can be privatized and the value of a(1) can be copied-in to each of the private copies. The following exemplifies this transformation:

```
doall $i = 1, n$
        private $a_p(1:m)$
```

```
          $a_p(1) = a(1)$
          ...
          do $j = 2, m$
              $a_p(j) = \dots a_p(j-1) \dots$
              $\dots = \dots a_p(j) \dots$
          enddo
          ...
      enddo
```

Although the Polaris restructurer implements copy-out operations, support for copy-in operations is lacking. An additional example of a case where both copy-in and copy-out are necessary for parallelization is discussed in Chapter 5, section 5.3.8.

4.4.6 Loops with Conditional Exits

In various cases in our test suite, while loops and loops with multiple exits are used to conditionally construct and manipulate data structures. As discussed at length in Chapter 2, such loops present difficulties in parallelization due to side-effects of iterations which are executed in parallel but would not be executed serially. Nonetheless, such loops can be parallelized if their bodies form associative coalescing loop operators.

One implementation difficulty which arises when parallelizing loops with conditional exits is the need for each processor to "flush" its remaining iterations once the exit has been taken. On the SGI Challenge no mechanism is provided to explicitly take an early exit from a parallel loop. In order to provide an early exit, we strip-mine the loop by creating a new, outer loop which executes one iteration on each processor. In the inner loop, iterations are logically interleaved so that processors execute relatively small slices of the iteration space. This enables exits to be detected with an efficiency proportional to the size of each slice. However, the transformation guarantees that iterations are not commuted[2]. A global, shared variable is used to store the minimum iteration in which the break condition is true. Any iteration greater than this minimum will take an early exit out of the loop. The following depicts this transformation at a high level:

[2]If this is confusing, please turn to Appendix B

```
          geti = n+1
          stagesize = blocksize * maxproc
          do 100 k = 1, maxproc
             do j = (k-1)*blocksize+1, n, stagesize
                do i = j, j+blocksize-1
                   if ( i > geti ) then goto 100
                   ...
                   if ( exit condition ) then
                      call lock
                         if ( i < geti ) then geti = i
                      call unlock
                      goto 100
                   end if
                   privatized reduction operation
                   ...
                end do
             end do
 100      end do
```

This transformation as shown involves the following five steps:

1. The iteration space is divided into stages.
2. Each stage is divided into blocks with one block assigned per processor. For simplicity, we assume here that the iteration space can be evenly divided by the blocksize.
3. Each processor goes through all stages; at each stage it executes the iterations in the block assigned to it.
4. Once a processor finds the exit condition true, it sets *geti* atomically to its current iteration. Note that once *geti* is set, it can only be reset to iterations less than *geti* due to the `if (i < geti)` statement.
5. If a processor finds that it is working on an iteration beyond *geti*, it will exit to 100, thereby flushing its remaining iterations.

This transformation has been somewhat simplified to illustrate the main idea. A detailed example of the parallelization of an associative, multiple-exit coalescing loop is contained in Appendix B. Chapter 5 further discusses the application of this technique in the sparse code cholesky.

In the following chapter we discuss a suite of sparse and irregular codes which we have developed in the course of our work. We then analyze this suite based on the techniques presented in this chapter.

CHAPTER 5

Automatic Parallelization of Sparse and Irregular Fortran Codes

5.1 Introduction

Irregular memory access patterns have traditionally caused difficulties in the automatic detection of parallelism, and in many cases parallelization is prevented. These problems are nonetheless important in that a significant fraction of current applications are irregular in nature.

In this chapter we present a benchmark suite representative of sparse and irregular codes which we have developed as part of this work. We consider how well the parallelization techniques presented in Chapter 4 apply to this collection of codes.

In conducting this work, we compare existing technology in the commercial parallelizer PFA from SGI with the Polaris restructurer [8]. In cases where performance is poor, we perform a manual analysis and determine the techniques necessary for automatic parallelization.

Benchmark	Description	Origin	# lines	Serial exec (seconds)
CHOLESKY	Sparse Cholesky Factorization	HPF-2	1284	265
DSMC3D	Direct Simulation Monte Carlo	HPF-2	1794	482
EULER	Euler equations on 3-D grid	HPF-2	1990	314
GCCG	Computational fluid dynamics	Vienna	407	739
LANCZOS	Eigenvalues of symmetric matrices	Malaga	269	868
MVPRODUCT	Basic matrix operations	Malaga	342	477
NBFC	Molecular dynamics kernel	HPF-2	206	375
SpLU	Sparse LU Factorization	HPF-2	363	471

Table 5.1: Benchmark Codes

5.2 The Benchmark Suite

Table 5.1 summarizes the eight codes in the benchmark suite employed in our experiments. The suite consists of a collection of sparse and irregular application programs as well as several kernels representing key computational elements present in sparse codes. Several of the benchmarks in our suite are derived from the set of motivating applications for the HPF-2 effort [18]. Exceptions include the kernels MVPRODUCT and LANCZOS which were developed as part of this project. The sparse CFD code GCCG was developed at the Institute for Software Technology and Parallel Systems at the University of Vienna, Austria.

5.2.1 CHOLESKY

The sparse cholesky factorization of a symmetric positive definite sparse matrix A produces a lower triangular matrix L such that $A = LL^T$. This factorization is used in direct methods to solve systems of linear equations. An example of the type of access pattern seen in CHOLESKY is depicted below:

```
do s = 1,nsu
   do j = isu(s),isu(s+1)-1
      snhead(j) = isu(s)
      nafter(j) = isu(s+1) - 1 - j
   enddo
 enddo
```

The indirectly referenced loop bounds of the inner `j` loop vary across iterations of the outer `i` loop. The Harwell-Boeing matrix BCSSTK30 was used as input for this benchmark [15].

5.2.2 DSMC3D

DSMC3D is a modification of the DSMC (Direct Simulation Monte Carlo) benchmark in 3 dimensions. DSMC implements a simulation of the behavior of particles of a gas in space using the Monte Carlo method [6]. An example of one of the access patterns occurring in this irregular application is abstracted below:

```
do i = 1, NM
   if (mcell(i)+1 .eq. ncell(i)) then
         cellx(mcell(i)) = cellx(mcell(i)) + 1
   endif
 enddo
```

In the above accumulation into `cellx`, subscripted subscripts occur on both the left and right-hand sides of assignment statements.

5.2.3 EULER

EULER is an application which solves the Euler equations on an irregular mesh. The computation is based on an indirectly referenced description of the grid. In addition, indirection is employed on both sides of assignment statements. The following code abstract exemplifies this two-level pattern of indirection:

```
 do  ng=1,ndegrp
  do  i=ndevec(ng,1),ndevec(ng,2)
  n1        = nde(i,1)
  n2        = nde(i,2)
  pw(n1,1)  = pw(n1,1) + qw(n2,1)*eps(i)
  pw(n2,1)  = pw(n2,1) + qw(n1,1)*eps(i)
 enddo
enddo
```

5.2.4 GCCG

GCCG is an example of a computational fluid dynamics solver. The access pattern is similar to that found in finite element methods where the value of an element is determined by the contribution of neighbors selected using subscripted subscripts. As a result, indirection occurs on the right-hand-side of the computed expressions.

```
      do  nc=nintci,nintcf
       direc2(nc)=bp(nc)*direc1(nc)
    *            -bs(nc)*direc1(lcc(nc,1))
    *            -bw(nc)*direc1(lcc(nc,4))
    *            -bl(nc)*direc1(lcc(nc,5))
      enddo
```

5.2.5 LANCZOS

The lanczos algorithm with full reorthogonalization determines the eigenvalues of a symmetric matrix [24]. LANCZOS is an implementation of the lanczos algorithm for sparse matrices. The key computational elements are the calculation of a sparse matrix-vector product and the reorthogonalization of a dense work matrix. Access patterns include subscripted subscripts on the right-hand-side of assignment statements as the following excerpt demonstrates:

```
      do  j=1,a_nr
       do  k=ar(j),ar(j+1)-1
        r(j)=r(j)+ad(k)*q(ac(k),i)
       enddo
      enddo
```

The matrix 1138_BUS of Harwell-Boeing collection was used as input for this benchmark.

5.2.6 MVPRODUCT

MVPRODUCT is a set of basic sparse matrix operations including sparse matrix-vector multiplication and the product and sum of two sparse matrices [4, 24]. The representation of the sparse matrices employs two different schemes: *compressed row storage* (CRS) and *compressed column storage* (CCS) [52]. The access pattern is demonstrated by the following code abstract:

```
      do  i=1,a_nr
       do  k=1,b_nc
        do  ja=ar(i),ar(i+1)-1
         do  jb=bc(k),bc(k+1)-1
           if (ac(ja).eq.br(jb)) THEN
            c(i,k)=c(i,k)
   &                + ad(ja)*bd(jb)
           endif
         enddo
        enddo
       enddo
      enddo
```

Here indirection occurs on the right-hand-side of the computed expressions. The matrix BCSSTK14 from the Harwell-Boeing collection has been used as input to this benchmark.

5.2.7 NBFC

The calculation of non-bonded forces forms a key element of many molecular dynamics computations [9]. NBFC computes an electro-static interaction between particles where the forces acting on an atom are calculated from a list of neighboring atoms. Similar to the DSMC3D benchmark, the data access pattern in this sparse code has indirection on both sides of the computed expressions:

```
do k = 1, ntimestep
 do i = 1, natom
  do  j = inblo(i),inblo(i+1)-1
   dx(jnb(j)) = dx(jnb(j)) - (x(i) - x(jnb(j)))
   dx(i)      = dx(i) + (x(i) - x(jnb(j)))
  enddo
 enddo
enddo
```

5.2.8 SpLU

SpLU computes the LU factorization of a sparse matrix. The LU factorization is used in several methods which solve sparse linear systems of equations. The factorization of a matrix A results in two matrices, L (lower triangular) and U (upper triangular), and two permutation vectors π and ρ such that: $A_{\pi_i \rho_j} = (LU)_{ij}$.

The pattern of access to arrays in SpLU includes indirectly referenced loop bounds across an iteration space traversed by a loop induction variable:

```
do i=cptr1(j),cptr2(j)
   a(shift)=a(i)
   r(shift)=r(i)
   shift = shift + 1
enddo
```

SpLU is a right-looking sparse LU factorization based on the CCS data structure. This algorithm is somewhat slower than the MA48 code from Harwell Subroutine Library [15], a left-looking standard benchmark for factorization. The motivation for developing a right-looking algorithm derived from the lack of significant parallelism in MA48. This led to the inclusion of the original C version of SpLU in the suite of HPF-2 motivating applications. The version of SpLU included in our benchmark suite is a Fortran implementation by the authors of the original HPF-2 version [2]. The sparse matrix lns_3937 from the Harwell-Boeing collection was used as input for the results reported in this chapter.

5.3 Analysis and Results

In Chapter 4 we discussed several techniques that we found important in parallelizing sparse and irregular Fortran codes. In this section, we categorize the transformations applicable to each benchmark.

		Polaris	PFA	Manual	Polaris	PFA	Manual
Benchmark	T_{seq}	T_{par}	T_{par}	T_{par}	Speedup	Speedup	Speedup
CHOLESKY	4:25	6:42	4:20	3:40	0.66	1.02	1.20
DSMC3D	8:02	6:35	7:53	1:45	1.22	1.02	4.95
EULER	5:03	2:23	4:56		2.20	1.02	
GCCG	12:19	1:27	1:57		8.49	6.32	
LANCZOS	14:28	2:01	1:58		7.17	7.36	
MVPRODUCT	7:57	1:42	1:11		4.68	6.72	
NBFC	6:15	1:15	6:20		5.00	0.99	
SpLU	3:54	15:25	3:44	1:01	0.25	1.04	3.84

Table 5.2: Speedups: PFA, Polaris, and Manual

Table 5.2 presents a comparison of the speedups obtained by Polaris with those of the commercial parallelizing compiler PFA, provided by SGI. With the exception of the Polaris-transformed version of EULER, the programs were executed in real-time mode on eight processors on a 12-processor SGI Challenge with 150 MHz R4400 processors. The Polaris-transformed version of EULER was executed in real-time mode on a 4-processor SGI Challenge with 200 MHz R4400 processors due to the unavailability of the 12 processor 150 MHz machine. The table shows that Polaris delivers, in several cases, substantially better speedups than PFA.

The table also portrays, in the manual column, additional speedups obtained using new techniques discussed in Chapter 4, section 4.4. In these cases the techniques were manually implemented and the resulting transformed program executed in parallel.

In general our results indicate that histogram reductions are one of the most important transformations applied in our suite. Other techniques which proved crucial to the process of parallelization include both sophisticated analysis of index array and induction variable access patterns, and the substitution of pseudo-random number generators.

In the following sections, techniques applied to each benchmark both manually and automatically will be outlined and compared to those applied by PFA.

5.3.1 NBFC

NBFC contains one computationally key loop which accounts for over 97% of the sequential execution time. Both histogram and single-address reductions occur in the loop. When a given array is involved in both types of reduction statement, it may be parallelized by applying the histogram reduction transformation at all reduction sites involving the array. The histogram reduction technique was sufficient to parallelize this loop, and excellent speedups were obtained. PFA, however, does not implement histogram reductions and therefore achieved no speedup on this benchmark.

5.3.2 CHOLESKY

The following techniques were applied to CHOLESKY:

- Histogram reductions

- Loops with conditional exits
- Proving monotonicity of index arrays
- Random number generator substitution

Histogram reductions are performed in the main update loop indexed by the variable kk in UPDATE_DO#3. This loop accounts for approximately 20% of the serial execution time. The transformation, however, did not yield significant speedups. This is due to the additional overhead incurred during the initialization and cross-processor reduction phases of the expanded transformation employed. We speculate that this may be due (at least in part) to the fact that little work other than the reduction is done in this loop. As a result, the initialization and final cross-processor reduction phases of the transformed loop essentially repeat the computation performed in the doall portion of the transformed loop.

Loops with conditional exits occur in GENQMD_DO400, UPDATE_DO#2, and UPDATE_DO#6. The transformation discussed in Chapter 4 section 4.4.6 was applied to GENQMD_DO400, a loop which performs a reduction across nodes to determine the minimum degree node. The reduction is terminated by a threshold condition which causes an early exit to be taken from the loop. The loop accounts for about 15% of the sequential execution time. Loop-level speedups of 1.6 were achieved on four processors.

The techniques outlined in section 4.4.3 of Chapter 4 apply in the SPARCHOL_DO1015 and SPARCHOL_DO1020 loops in CHOLESKY. Together these loops account for approximately 24% of the serial execution time. A loop-level speedup of 2.84 was obtained in SPARCHOL_DO1020 on four processors.

The final transformation involved the substitution of a parallelized pseudo-random number generator for the library call in SPARCHOL_DO1015. This loop accounts for approximately 3.6% of the serial execution time. The call to the random number generator is the primary work done in the loop, and a loop-level speedup of 2.5 was achieved on eight processors.

Although both Polaris and PFA find a large number of loops parallel in this code, little high-level parallelism is available due to the nature of the supernode algorithm employed. This is reflected in the results for all three versions of the benchmark: Polaris, PFA, and the manually transformed code.

5.3.3 DSMC3D

The following techniques were applied to DSMC3D:

- Histogram Reductions
- Random Number Generator Substitution
- Combined Static & Run-time Analysis of Induction Variable Ranges
- List Reduction in Coalescing Loop Operators

Histogram reductions, discussed in Chapter 4, section 4.4.1, are important in several loops: INDEXM_DO300, INDEXM_DO700, COLLMR_DO100, MOVE3_DO#3, and MOVE3_GOTO100. Together these five loops account for approximately 84% of the sequential execution time. Random number generator substitution, discussed in Chapter 2, section 2.3.10, is important in COLLMR_DO100, MOVE3_GOTO100, INIT3_DO605, and ENTER3_do4. Together these four loops account for almost 60% of the serial execution time. There are two other loops which contain conditionally incremented induction variables, ENTER3_DO4 and INIT3_DO605. Together these loops account for approximately 7.5% of the sequential execution time.

Both of the latter loops are parallelizable using techniques outlined in [44] for determining the closed-form of induction variables if the induction can be proven to be monotonically increasing. However, the conditional increment poses a problem in that monotonicity may not hold and the induction variable ranges may overlap as a result. Through static analysis of the pattern in these loops, a simple run-time test can be abstracted which determines that the induction variable ranges do not overlap and that the loops may be executed in parallel. This transformation was portrayed in Chapter 4, section 4.3.1.

5.3.3.1 List Reduction in Coalescing Loop Operators

DSMC3D contains a while loop in the MOVE3 subroutine which accounts for approximately 35% of the sequential execution time. This loop computes the movement phase, the first of three phases executed each iteration of the outermost time-stepping loop. Molecules involved in the movement phase are stored in lists comprised of two global arrays. These arrays are indexed almost without exception by the loop induction variable. However, when a molecule leaves the flow, it is deleted from the list and replaced by the last molecule in the list. This creates loop-carried dependences in the loop. However, the deletion of molecules can be deferred until after the entire list has been processed [66, 7]. Based on this fact, the following transformation can be made:

```
i = 1
j = n
while (i < j)                      parallel loop i = 1, n
    if (cond(a(i)) then               if (cond(a(i)) then
        a(i) = a(j)                       a(i) = mark
        j = j - 1           ⇒          endif
    else                           enddo
        i = i + 1                  call remove_marked(a)
    endif
endwhile
```

The current molecule in $a(i)$ is marked for later removal. After exit from the parallel loop, marked elements are removed and the array a is packed. Effectively, the operation of removing and replacing elements in a is an example of *list reduction* as discussed previously in Chapter 3 section 3.3.2, and can be modeled as a parallelizable associative coalescing loop operator.

The combination of these techniques in the loops mentioned above contributed to the overall program speedup of 4.95 in the manually parallelized version. The speedups reported for Polaris include the histogram reduction in MOVE3_DO#3. PFA, however, does not implement any of these techniques and therefore achieved less of a speedup than Polaris, although both were low.

5.3.4 EULER

EULER contains five time-consuming loops which are computationally important: DFLUX_DO100, DFLUX_DO200, EFLUX_DO100, EFLUX_DO200, and PSMOO_DO20. Together these loops account for over 70% of the serial execution time of the program. In all five loops, the histogram reduction transformation is the only transformation necessary to parallelize the loop.

The speedups for this benchmark are also fairly good, with an overall program speedup of 2.2 on 4 processors (based on the same efficiency (S_p/p), the speedup on 8 processors would be 4.4). No speedup was achieved with PFA due to the fact that PFA does not recognize and solve histogram reductions. In comparison, Polaris did considerably better.

5.3.5 GCCG

The primary access pattern in GCCG involves indirections which occur on the right-hand-sides of assignment statements, as discussed in section 5.2. These pose no particular dependence problem due to the fact that the array locations are read but not written. Many reductions occur in GCCG, but they are all scalar or *single-address* reductions in which the reduction variable is a single element of an array. Current parallelizing technology is capable of recognizing and transforming such reductions into parallel form. This fact is reflected in the speedup results for PFA as well as Polaris.

5.3.6 LANCZOS

LANCZOS presents a situation similar to that found in GCCG in that the primary access pattern involves indirection on assignment statements' right-hand-sides during a sparse matrix-vector product operation. The reorthogonalization is computed using dense matrices, and arrays are accessed via loop indices. As a result, no loop-carried dependences prevent parallelization. This fact is reflected in the good speedups achieved by both Polaris and PFA.

5.3.7 MVPRODUCT

MVPRODUCT has been implemented such that dense matrices result from the combination of sparse matrices. Due to this fact, indirection arises only on the right-hand-sides of assignment statements. This type of indirection poses no particular problem to parallelization, and the speedups achieved by both PFA and Polaris reflect this fact.

5.3.8 SpLU

The parallelization of SpLU involved the following techniques:

- Proving monotonicity of index arrays
- Combined Static & Run-time Analysis of Induction Variable Ranges
- Copy-in and Copy-out

Loop DPFAC_DO50 in SpLU accounts for almost 100% of the serial execution time of this benchmark. This loop involves access to arrays via an induction variable which is conditionally incremented. In this case, static analysis of the code reveals conditions which can be tested at run-time to prove that ranges are independent (non-overlapping). Consider the following example abstracted from SpLU DPFAC_DO50:

```
     do 20 i=1,n
        ia_2(i) = ia_1(i)
20   continue
     do 100 k=1,n
        shift=mod(k,2)*lfact+1
        do 50 j=k+1,n
           c1=shift
           do 60 i=ia_1(j),ia_2(j)
              a(shift)=a(i)
              shift=shift+1
60         continue
           c2=shift-1
           if (fill-in) then
              c2=c2+1
              a(shift:shift+positive_inc)= ...
              shift=shift+positive_inc
           endif
           do 95 i=ia_2(j)+1,ia_1(j+1)-1
              a(shift)=a(i)
              shift=shift+1
95         continue
           ia_1(j)=c1
           ia_2(j)=c2
50      continue
        ia(n+1)=shift
100  continue
```

Loop 100 is the outermost loop, and is executed for the n columns in array a. ia_1 and ia_2 are index arrays. $shift$ is an induction variable which is also used as an index into a. Two facts are sufficient to show that the do_50 loop may execute in parallel: one, for each j in do_50, the range of $shift$ must not overlap the range $ia\_1(j), ia\_1(j+1)-1$ for iterations of do_50 executed on other processors[1]; and two, the range of $shift$ must not overlap the same range of $shift$ for iterations executed on other processors[2].

In order to prove these two points we must first determine that ia_1 and ia_2 are non-decreasing. These index arrays are reassigned in each iteration of do_50, thereby complicating the analysis of the access pattern. However, it is possible to determine statically at compile time simple conditions under which these arrays will be non-decreasing. First, note that the induction variable $shift$ is never decremented in the loop. It is conditionally incremented under the "fill-in" condition by a positive amount. Likewise, the initial conditions (loop 20) guarantee that do_60 will execute at least one iteration. Thus, one of the invariant conditions of this loop is that the induction variable $shift$ is strictly increasing. If ia_1 and ia_2 are initially non-decreasing, by induction this invariant condition is sufficient to guarantee that they will remain non-decreasing across the entire execution of the outermost loop 100. Thus, our task of proving that these index arrays are non-decreasing has been reduced to the complexity of executing the test `ia(i+1).ge.ia(i)` for `i = 1,n` once at run-time.

As a side-effect of this analysis we have proven that no output dependences exist across iterations of the do_50 loop. This is a result of the fact that $shift$ is strictly increasing in do_50.

Given that ia_1 and ia_2 are non-decreasing, the next step is to show that for each j in do_50, the range of $shift$ does not overlap the range $[ia\_1(j), ia\_1(j+1)-1]$ for iterations of do_50 executed on other processors (in effect, we are proving that there are no flow or anti-dependences across iterations of do_50). This can be accomplished using a simple test which compares the upper bound of $shift$ to the lower bound of ia_1 and the lower bound of $shift$ to the upper bound of ia_1. As before, the presence of a conditional "fill-in" increment to $shift$ complicates the analysis. However, we can use an estimate of the maximum value of $shift$ by determining an upper bound across the entire iteration space of the do_50 loop. Based on the initial conditions and the strictly increasing nature of $shift$, $ia\_2(j) \geq ia\_1(j)$. Thus we know the tripcount of do_60, and we can conservatively assume that the "fill-in" is always true.

[1]I.e., no flow or anti-dependences are present
[2]I.e., no output dependences

Together these facts lead to the following run-time test to confirm the non-overlapping nature of reads and writes to a:

```
min_i=ia(k+1)
max_i=ia(n+1)-1
min_shift=shift
max_shift=shift+(max_i-min_i)+((n-k)*positive_inc)
if(min_shift.gt.max_i .or. max_shift.lt.min_i) then
       parallel=.true.
else
       parallel=.false.
end if
```

This test is placed outside the do_50 loop, and as a result incurs little overhead. When it is true, do_50 may be executed in parallel. When false, it must (conservatively) be executed serially. What this test actually proves is that writes to a are independent of reads to a across all iterations of do_50. This concludes the development of a test capable of proving that the iterations of do_50 are independent.

The final transformation applied in DPFAC_DO50 involved the copy-in and copy-out of the privatized arrays $a, r, cptr1$ and $cptr2$. Each of these variables have initial values which must be copied into each processor's privatized copy of the variable prior to the start of parallel execution.

Neither PFA nor Polaris implement the functionality present in these three techniques, and the corresponding speedups reflect this fact. Polaris, in particular, applied the histogram reduction transformation to an inner loop with a low tripcount. This resulted in a significant slowdown.

5.4 Conclusion

In our study of our sparse and irregular benchmark suite we have determined that indirection on the right-hand-sides of assignment statements is not a hindrance to automatic parallelization. We have also identified several new techniques which begin to point to the fact that, although much work remains to be done, automatic parallelization of sparse and irregular codes is feasible.

In the following chapter we consider the application of the techniques presented in Chapter 4 on Fortran application benchmarks containing outer time-stepping loops.

CHAPTER 6

Parallelism in Time-Stepping Loops

6.1 Introduction

The Polaris restructurer recognizes much doall parallelism at both the outer and inner loop level [8]. However, few experiments have been conducted on actual application codes to determine whether effective advantage can be taken of additional non-doall parallelism.

In this chapter we perform an analysis of two applications from the SPEC CFP95 benchmark suite in order to determine whether significant non-doall parallelism is present and whether this can be effectively combined with doall parallelism to achieve parallelism on multiple levels.

6.2 Transformations Employed in su2cor

In this section we discuss the application of several of the techniques presented in Chapter 4 on the SPEC CFP95 benchmark su2cor. The SPEC CFP95 Benchmark su2cor is a quantum physics program which computes the masses of elementary particles based on a monte carlo method. Transformations employed in parallelizing su2cor include the following:

- Discontinuous Inductions

- Symbolic Computation of Closed-forms
- Semi-private Variables
- Doacross
- Loop Pipelining
- Synchronization Schemes
- Multi-level Parallelism

Some of these transformations are not discussed in Chapter 4, and we assume the reader is familiar with these techniques (e.g., doacross)[1]. Examples of the application of each of these transformations follows.

6.2.1 Discontinuous Inductions

The example given in Chapter 4, section 4.3.1 typifies two induction variables found in SU2COR_do60. These induction variables, *ifreq* and *ipr*, were solved using the transformation depicted in the example in Chapter 4.

6.2.2 Symbolic Computation of Closed-Forms

The following example portrays a section of code from the routine TRNGV which implements a lagged Fibonacci pseudo-random number generator:

```
IFIRST=0
IMAX=2147483647
DO 100 N=1,N103
    DO 10 I=IFIRST,IFIRST+102
        IREG(I+250)=IREG(I+147)-IREG(I)
        IF (IREG(I+250).LE.0) IREG(I+250)=IREG(I+250)+IMAX
    10 CONTINUE
    IFIRST=IFIRST+103
100 CONTINUE
DO 20 I=IFIRST,IFIRST+LEFT-1
    IREG(I+250)=IREG(I+147)-IREG(I)
    IF (IREG(I+250).LE.0) IREG(I+250)=IREG(I+250)+IMAX
20 CONTINUE
RNORM=R1*RMAX
```

[1]Readers seeking background material may consult [68]

```
      DO 30 I=1,LVEC
         RANVEC(I)=RNORM*IREG(I+249)
   30 CONTINUE
      DO 40 I=0,249
         IREG(I)=IREG(I+LVEC)
   40 CONTINUE
```

TRNGV is called from within a nest of the major serial loops, including the outermost time-stepping loop SU2COR_do60 as well as SWEEP_do200. This section of code has an explicit recurrence relation of the form $a_i = a_{i-103} + a_{i-250}\ mod\ 2^{31} - 1$. It is an implementation of a lagged-Fibonacci pseudorandom number generator with recursion parameters of (103, 250) [48]. Note that the conditional subtraction performs a modulo operation which results in an integer in the range $[0, 2^{31}-2]$. In theoretical terms, this is the *representative residue class* of the ring which contains equivalence classes of the integers modulo $2^{31} - 1$.

The recurrence is computed in loops 100 and 20 in the above code. Loop 30 defines the storage variable *ranvec* which is returned to the caller. Loop 40 reinitializes the recurrence variable *ireg* with the final 250 values computed in loops 100 and 20. As a result, the sequence of values computed by the recurrence is mathematically continuous.

One approach to solving linear homogeneous recurrences of this nature is to symbolically calculate the closed-form of the recurrence during compilation as was discussed in the Chapter 4, section 4.3.1. Although a closed-form was determined for this recurrence, the actual computation of the sequence requires more precision than that available in the double precision floating-point representation. As a result, loop distribution was employed to break the dependence arcs in two enclosing loops in the calling subroutine SWEEP.

The loop distribution transformation required that *ranvec* be expanded by one dimension for each of the enclosing loops from within which the computation was hoisted. Choosing the level of loop distribution involved a tradeoff between the increase in space complexity due to expansion and the advantage of breaking dependence arcs at outer loop levels. Initially the recurrence was hoisted three levels to a point outside of loop SWEEP_do200/1, but this exceeded the available memory on the machine. As a result, a compromise was made and the recurrence was hoisted out only two levels to SWEEP_do200/2, a perfectly nested loop inside SWEEP_do200/1.

6.2.3 Semi-private Variables

Section 4.3.3 of Chapter 4 discussed semi-private variables in the general case. Such a pattern occurs in the main program unit SU2COR in the outermost time-stepping loop SU2COR_do60. In this case, the semi-private variable is the main data structure u, a double precision, 4-dimensional structure of size [4103,4,16,4]. u is semi-private from the call to CORR onwards in SU2COR_do60, allowing CORR to execute in tandem with SWEEP. Code depicting this transformation is outlined in section 6.2.5 below, and is based on the loop pipelining technique.

6.2.4 Doacross

The loop SWEEP_do200 in su2cor is a candidate for doacross parallelism for two reasons: one, there is work done at the head of the loop which does not involve accesses to variables with loop-carried dependences; and two, since the loop-carried dependences present in this loop are conditional in nature, it is possible to dynamically take advantage of the case(s) where no actual dependences exist.

A doacross loop was implemented for do200/1, the outermost of a perfect nest of two loops. This was accomplished using synchronization to enforce the dependence relationships on sections of the main data structure u, a double precision, 4-dimensional structure of size [4103,4,16,4]. The issue of granularity of synchronization arose, and it was determined empirically that synchronization should be done on “panels” of u. To understand this choice, consider Figure 6.1.

In this figure, u is divided up into 2^{ndim} lattices in the k dimension. Each lattice consists of $ndim$ “panels” in the j dimension and each panel consists of $lvec$ elements in the i dimension. Each element is a group of four double-precision floating-point numbers.

The computation proceeds as pictured below:

```
do su2cor_60 $it = 1, nmeas$
```

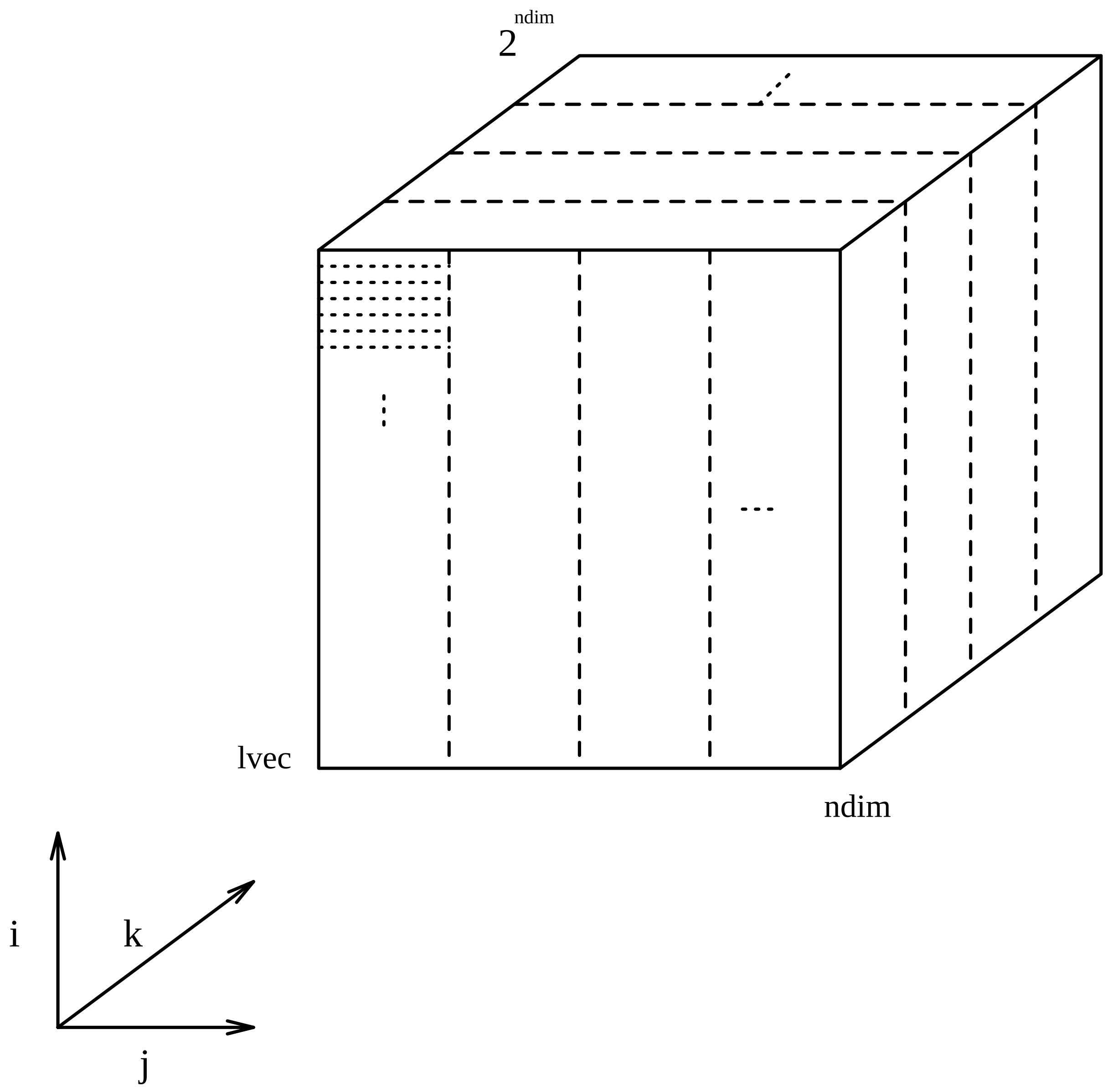

Figure 6.1: Data Access Pattern in SPEC CFP95 Benchmark su2cor

```
    doacross sweep_200/1 $ilat = 1, 2^{ndim}$
        do sweep_200/2 $i1 = 1, ndim$
            ...
            conditionally read multiple panels of $u$
            ...
            conditionally write multiple elements of panel $i1$, lattice $ilat$ of $u$
            ...
    200 continue
enddo 60
```

The loop SWEEP_do200/1 is the outermost loop in the SWEEP subroutine. The SWEEP subroutine is called from within SU2COR_do60 in the SU2COR main program unit (the call has been replaced by the body of SWEEP in order to simplify the example above). Each iteration of the main time-stepping loop SU2COR_do60 conditionally reads and writes panels and elements, respectively, in the 2^{ndim} lattices.

As mentioned earlier, su2cor is based on a monte carlo method. As a result, the reads of panels of u in SWEEP_do200/2 span the entire data structure - i.e., panels from several different lattices are read in each iteration of SWEEP_do200/2. Writes of u, on the other hand, are limited to elements in lattice $ilat$, panel $i1$ in loop 200/2. As a result, synchronization on u is done on a panel-by-panel basis. Section 6.2.6 provides additional detail.

In order to implement the doacross transformation, synchronization was required for the computation of pseudo-random numbers discussed in Section 6.2.2 above as well. This was a result of the choice to hoist the computation only two levels due to the complexity of space usage when three level hoisting was employed. An outline of the synchronization is given in the code example in the following section.

6.2.5 Loop Pipelining

In order to take advantage of the semi-private variable u in su2cor, the loop pipelining technique was employed. In this case, the outermost time-stepping loop SU2COR_do60 was coalesced with the outermost loop in the subroutine SWEEP, do200/1. The resulting loop was executed as a doacross as exemplified in section 6.2.4. However, SU2COR_do60 actually includes a call to the subroutine CORR which was not pictured in the previous example. As was discussed in section 6.2.3, the main data structure u is semi-private in the context of this call to CORR within SU2COR_do60. As a result, access to u was synchronized *and* u was partially privatized in the coalesced 200/1-60 loop. An outline of the code depicting these transformations follows:

```
post($ireg$,1)
post($u$,1)
doacross 200/1-60 $ki = 1, 2^{ndim} * nmeas$
    $i = mod(ki - 1, 2^{ndim}) + 1$
    initialize privatized reductions
    wait($ireg$,$ki$)
            precomputation of $ireg_{exp}$
    post($ireg$,$ki + 1$)
    do 200/2 $j = 1, ndim$
            ...
            wait($u_{panel_z}$,$ki$)
                    conditional reads of:
                            $u_{panel_z}$
                            $uprev_{panel_z}$
                            $uprevprev_{panel_z}$
                    ...
                    accumulate reductions
                    ...
                    $uprevprev_{panel_z} = uprev_{panel_z}$
                    $uprev_{panel_z} = u_{panel_z}$
                    conditional update of $u_{panel_z}$
            post($u_{panel_z}$,$ki + 1$)
    enddo
    if ($i.eq.n$) then
            reduce reductions across processors
            ...
            output intermediate results
            copy $u$ to $u_{private}$
            fork $corr$
    endif
enddo
```

The semantics of the *wait* and *post* operations are as follows: the first argument indicates the data structure for which access is being synchronized and the second is the timestep of the computation. Much detail in terms of synchronization has been abstracted to make the example understandable. However, section 6.2.6 provides additional detail.

In essence the transformed code combines the doacross parallelism in loop 200/1 with the do-pipe parallelism exposed by the semi-privatization of u in loop 60. Although we have not explicitly stated so, in fact the combination of the doacross loop with loop pipelining represents an example of multi-level parallelism. This issue is further explored in section 6.2.7.

6.2.6 Synchronization

The implementation of the four techniques discussed above in sections 6.2.2, 6.2.3, 6.2.4, and 6.2.5 required various synchronization schemes. The coalesced doacross loop 200/1-60, for example, employed a non-blocking barrier to improve the performance.

Due to the nature of the su2cor application, access to panels of the main data structure u involve potential loop-carried flow, anti, and output dependences. Flow dependences were resolved using explicit synchronization. Anti and output dependences were resolved using *variable renaming* [42]. The renamed variables in the transformed code depicted in section 6.2.5 are *uprev* and *uprevprev*, shorthand for the values of u in the preceding two timesteps. Here, a timestep is defined as one iteration of the original loop SU2COR_do60. Recall from section 6.2.4 that SU2COR_do60 encloses SWEEP_do200/1. Therefore, one timestep of the coalesced 200/1-60 loop is equivalent to the execution of the entire iteration space of the original SWEEP_do200/1. The choice of a two-timestep "buffer" was based on empirical observations of the amount of parallelism in the loop for the given input set.

Each of the 2^{ndim} lattices of u has an associated timestep. If a processor p wishes to update a given panel z in lattice l, it first ascertains the current minimum global timestep for all lattices (i.e., the minimum iteration of the original time-stepping loop SU2COR_do60 still being computed for any given lattice). If the minimum global timestep is more than two steps behind the timestep that processor p is computing, p must wait. Otherwise, p updates elements in panel z of u, *uprev*, and *uprevprev*. In this way, processor p_i is never allowed to get more than two timesteps ahead of processor p_j for all i, j of the processors participating in the computation.

Similarly, if processor p wishes to read a panel z, it first determines whether there is a potential flow or anti dependence. However, as a result of the write-access pattern in SWEEP_do200/2, certain dependences are automatically ruled out. This is a consequence of the fact that a given panel of u is conditionally updated only once each timestep. Suppose, for example, panel z is updated in iteration $ilat_i, i1_j$ by processor p_1. Furthermore, suppose that processor p_2 wishes to read panel z and p_2 is currently computing in panel $ilat_m, i1_n$ such that $ilat_m \leq ilat_i$ and $i1_n < i1_j$. If both p_1 and p_2 are computing in the same timestep, no flow-dependence can exist. As mentioned earlier, this is a direct result of the access pattern to u within SWEEP_do200/2.

In the event that a flow-dependence is possible[2] (e.g., $ilat_m \geq ilat_i$ and $i1_n > i1_j$), processor p_2 waits for panel z to be updated by p_1. As soon as panel z reaches p_2's timestep, p_2 reads u_{panel_z}[3]. On the other hand, if the potential exists for an anti-dependence, a test is made to determine if p_2's timestep is less than, greater than, or equal to z's timestep. This is done in order to determine how many times z has been updated. If p_2's timestep is equal to z's, u_{panel_z} is read. If z's timestep exceeds that of p_2 by one, $uprev_{panel_z}$ is read. Similarly, if z's timestep exceeds that of p_2 by two steps, $uprevprev_{panel_z}$ is read.

The synchronization implementation is based on *test&set* and *add_then_test* primitives provided by the architecture. Nine functions were written in C to provide support for the synchronization schemes discussed above. Primary support for the non-blocking barrier was provided by *tstepper*, a function which employs what is conceptually a monitor to provide access to the global minimum timestep. Likewise, *reader* and *writer*, which provide dynamic support for handling flow, anti, and output dependences, employ similar constructs.

The synchronization schemes employ busy-waiting. At a low level, all synchronization variables are mapped on cacheline boundaries. This provides clean busy-waiting for variables which are shared across processors (i.e., no false sharing of synchronization variables). As an optimization, exponential backoff is employed while processors spin-wait at synchronization points [40].

[2]Recall that updates to u are conditional

[3]If, while p_2 was waiting for z, z was updated twice (i.e., z was updated again by another processor p_3 before p_2 could read it), p_2 reads $uprev_{panel_z}$

6.2.7 Multi-level Parallelism

Section 6.2.5 depicted the pipelining and doacross transformations which were applied to the coalesced loops SWEEP_do200 and SU2COR_do60. This coalesced loop is the outermost loop in the su2cor benchmark, and these two types of parallelism have been combined to produce the speedups reported in section 6.4.2 below.

However, additional doall parallelism exists in subroutines called from within this loop. Specifically, INTACT and LOOPS are in the call tree rooted in SWEEP_do200, and both of these routines make multiple calls to compute-intensive doall parallel loops. Examples of such loops include MATMAT_do10, which performs matrix multiplication.

In order to take advantage of additional loop-level parallelism in su2cor, an architecture which supports such parallelism was required. This posed an implementation difficulty in that few architectures support more than a single level of parallelism for loops. For several of the results reported in sections 6.4.2 and 6.4.4, we have employed an SGI Challenge. The Challenge currently supports only a single level of loop parallelism[4]. In order to take advantage of doall loops inside the doacross loop, it was necessary to port su2cor to the Convex/HP Exemplar, the only architecture that we are aware of that provides support for more than one level of loop parallelism. As a result, the version of su2cor which was parallelized on the SGI Challenge was ported to the Convex/HP Exemplar.

The Exemplar currently supports two types of parallelism: *symmetric* and *asymmetric*. Symmetric parallelism includes two types of loop-level parallelism which correspond to execution of multiple threads either within a tightly-coupled cluster (hypernode) or across hypernodes. Threads executing under this model of computation all execute the same instruction stream. Asymmetric parallelism, on the other hand, is akin to the Unix fork command in that only a single thread is spawned.

Cross-cluster symmetric parallelism is termed *node-wise* parallelism, and intra-cluster symmetric parallelism is referred to as *thread-wise* parallelism. Compiler directives are available for identifying loops

[4]Note that the implementation of loop pipelining in a doacross loop made use of only a single-level of loop parallelism in the doacross loop itself

and tasks which are to execute with multiple threads at either of these levels. Asymmetric parallelism can be invoked for a single process from within either node or thread-parallel constructs. Likewise, symmetric parallelism can be invoked from within an asymmetric thread. Consider the following example:

```
c$dir loop_parallel ($nodes, ivar = i$)
do $i = 1, n$
          c$dir loop_parallel ($threads, ivar = j$)
          do $j = 1, n$
                    ...
          enddo
enddo
```

Here we have two levels of parallelism in the i and j loops. At runtime the iterations of the outer i loop will be distributed across the available hypernodes in a node-wise parallel fashion. The iteration space of each invocation of the the inner j loop, however, will be divided amongst the processors on a given hypernode in a thread-wise fashion[5].

Now we will add an asymmetric thread to the example:

```
c$dir loop_parallel ($nodes, ivar = i$)
do $i = 1, n$
          c$dir loop_parallel ($threads, ivar = j$)
          do $j = 1, n$
                    ...
          enddo
          if ($condition$) then
                    call cps_thread_create($sub$)
          endif
enddo
subroutine $sub$
          c$dir loop_parallel ($threads, ivar = k$)
          do $k = 1, n$
                    ...
          enddo
```

[5]Eight processors are clustered on each hypernode in the SPP-1200 employed in our experiments

We have now added a third dimension to the parallelism in that an asymmetric thread executing *sub* is conditionally spawned each iteration of the outer *i* loop. A final, fourth dimension of parallelism is created when *sub* encounters the thread-parallel *k* loop during execution.

This is, in essence, the transformation employed in the coalesced 200/60 loop in su2cor on the Exemplar: node-wise and thread-wise parallelism are combined in the doacross execution of loop 200/60, and asymmetric and thread-wise parallelism are combined in the pipelined execution of the same loop. Section 6.4.2 reports speedup results for these transformations on the Exemplar.

6.3 Transformations Employed in tomcatv

In this section we discuss the application of one of the techniques presented in Chapter 4 on the SPEC CFP95 benchmark tomcatv. In addition, we discuss the solution of min/max reductions in parallel in the context of this benchmark.

The SPEC CFP95 Benchmark tomcatv is a computational fluid dynamics program based on a solver employing LU decomposition. Transformations employed in parallelizing tomcatv include the following:

- Min/Max Reductions
- Loop Pipelining

Examples of the application of each of these transformations follows.

6.3.1 Min/Max Reductions

The current implementation in Polaris does not support the recognition of reduction operations which involve intrinsic min or max operators. Nonetheless, the parallelization of such reductions can be accomplished based on the transformation described in [44]. An example of such a reduction occurs in loop-nest MAIN_do80 in tomcatv. The following portrays this loop:

```
DO 80 J = 2,N-1
```

```
        DO 80 I = 2,N-1
            RXM(ITER) = MAX(RXM(ITER), ABS(RX(I,J)))
            RYM(ITER) = MAX(RYM(ITER), ABS(RY(I,J)))
    80 CONTINUE
```

As can be seen from this example, do80 is a max reduction on two variables, rx and ry. $ITER$ is the index of the outermost time-stepping loop (MAIN_do140).

```
do j = 1, procs
    rxm_e(j) = 0.0
    rym_e(j) = 0.0
enddo
doall k = 2, n-1
    do i = 2, n-1
        rxm_e(thread-num()) = max(rxm_e(thread-num()),abs(rx(i, k)))
        rym_e(thread-num()) = max(rym_e(thread-num()),abs(ry(i, k)))
    enddo
enddo
do m = 1, procs
    rxm(iter) = max(rxm(iter),rxm_e(m))
    rym(iter) = max(rym(iter),rym_e(m))
enddo
```

In the above, rxm_e and rym_e are expanded versions of the original *single-address* reduction variables rxm(iter) and rym(iter)[6]. They are expanded by the number of concurrent threads, initialized to zero, and indexed by the thread-id of each thread participating in the computation. Following the parallel reduction in the k loop, the partial results are summed across threads in the final m loop.

6.3.2 Do-pipe

The loop pipelining transformation discussed in section 6.2.5 has a corresponding application in tomcatv. Consider the following example abstracted from tomcatv loop 140:

```
DO 140 ITER=1,ITACT
    ...
    DO 60 J=2,N-1
```

[6]Note that the index iter is invariant in the do80 loop nest

```
        DO 50 I=2,N-1
            . . .
            RX(I,J) = A*PXX+B*PYY-C*PXY
            RY(I,J) = A*QXX+B*QYY-C*QXY
        50 CONTINUE
    60 CONTINUE
    DO 80 J=2,N-1
        DO 80 I=2,N-1
            RXM(ITER) = MAX(RXM(ITER), ABS(RX(I,J)))
            RYM(ITER) = MAX(RYM(ITER), ABS(RY(I,J)))
    80 CONTINUE
    . . .
140 CONTINUE
```

Both the do60 and do80 loops access rx and ry in their entirety during each step of the outermost loop 140 (index ITER). The dependences on rxm and rym in do140 are not loop-carried - i.e., rxm and rym are loop-private variables which are written in do80 and read prior to the exit of do140.

As a result, it is possible to overlap the access to rx and ry in do60 and do80 such that these two loops can be executed concurrently. This is accomplished by synchronizing on columns of rx and ry as pictured below:

```
DO 140 ITER=1,ITACT
    . . .
process_1
    DO parallel 60 J=2,N-1
        DO 50 I=2,N-1
            . . .
            RX(I,J) = A*PXX+B*PYY-C*PXY
            RX_p(I,J) = RX(I,J)
            RY(I,J) = A*QXX+B*QYY-C*QXY
            RY_p(I,J) = RY(I,J)
        50 CONTINUE
        post(j)
    60 CONTINUE
process_2
    DO 80 J=2,N-1
        wait(j)
        DO 80 I=2,N-1
            RXM(ITER) = MAX(RXM(ITER), ABS(RX_p(I,J)))
            RYM(ITER) = MAX(RYM(ITER), ABS(RY_p(I,J)))
    80 CONTINUE
    . . .
140 CONTINUE
```

In this example, loops do60 and do80 are executed as two separate processes which share the same address space. The term "loop pipelining" illustrates the nature of the access pattern of the two loops: do60 executes one iteration (writing one entire column of both RX_p and RY_p), then posts to notify do80 that these private variables are available. Loop do80 waits for the post then accesses the same columns of RX_p and RY_p in turn, with the result that access to the Jth columns of rx and ry are "pipelined". In much the same way that vector chaining enhances performance in a vector processor, loop pipelining hides the execution time of the do80 loop almost entirely.

The multi-level parallelism discussed in section 6.2 has also been implemented in tomcatv. When do60 is executed as a parallel, doall loop, the overlapped execution of do80 implies that two levels of parallelism exist: one level at the pipelined outermost loop do140, and the second level at the parallel loop do60. In addition, loop do80 may be executed either as a parallel reduction loop or as a serial loop depending on the ratio of the respective execution times for the loops. Results for the execution of tomcatv are discussed in section 6.4.4.

6.4 Results

In this section performance results will be discussed for the two benchmarks from the SPEC CFP95 suite which have been manually transformed using the techniques described in sections 6.2 and 6.3. The discussion will be organized on a loop-by-loop basis for the computationally important loops in these codes.

6.4.1 Loops in su2cor

The computationally key loops in su2cor are delineated in Table 6.1.

The S/P markings in Table 6.1 denote whether the given loop is parallel (P) or serial (S). The Depth column designates the (interprocedural) nesting level of the given loop, and % T_{seq} is the percentage of the sequential execution time for the loop (including nested loops). As these figures indicate, many time-consuming outer loops are serial in nature.

Subroutine	Loop	Depth	% T_{seq}	S/P
SU2COR	do60	1	99.9809	S
SWEEP	do200	2	63.1459	S
LOOPS	do900	2	36.8664	S
SWEEP	do220	4	34.3754	P
LOOPS	do400	3	29.6517	P
MATMAT	do10	5	18.3314	P
SWEEP	do310	4	16.1173	S
INT2V	do100	6	13.9536	S
SWEEP	do311	5	12.7783	P

Table 6.1: Loops > 10% of Sequential Time

6.4.2 Results for su2cor

Speedups for the loop SWEEP_do200 on the SGI Challenge are displayed in Table 6.2. These include two different transformations and their cumulative effect on loop-level speedups. Experiments were conducted on eight processors on an R4400-based Challenge in dedicated mode.

Transformation	speedup
No transformations	1.0
Doacross transformation	1.9
Doacross with non-blocking barrier	2.1

Table 6.2: Speedups of SWEEP_do200 on SGI Challenge

Table 6.3 outlines whole program speedups for su2cor on the SGI Challenge. As noted in section 6.2, these results are based on a combination of doacross and do-pipe parallelism in the coalesced sweep_do200/su2cor_do60 loop. Experiments were conducted in dedicated mode using 10 of the 12 processors available on the machine. Eight of the ten processors were assigned to the doacross loop pictured in the code example in section 6.2.5. In the same code example, the routine *corr* is conditionally forked when $i.eq.n$. When *corr* is forked, an additional processor is utilized to execute *corr*. Lastly, one final processor is utilized to enforce synchronization in the do-pipe.

Table 6.4 portrays whole program speedups for experiments with multi-level loop-based parallelism on the HP/Convex Exemplar. The experiments compare the performance of the various transformations

Transformations	speedup
No transformations	1.0
Doacross + Pipelining	3.63

Table 6.3: Su2cor Program Speedups on SGI Challenge

discussed in section 6.2. Experiments were conducted on a two-hypernode, 16 processor subcomplex in dedicated mode.

Transformations	$p = 3$	$p = 6$	$p = 9$	$p = 12$	$p = 15$
No transformations	1.0	1.0	1.0	1.0	1.0
Doall Only	1.88	2.19	1.85	1.83	1.77
Multi-level Doacross/Doall + Pipelining	1.48	1.99	2.05	2.23	2.23

Table 6.4: Su2cor Program Speedups on HP/Convex Exemplar

The columns in Table 6.4 are labeled with the number of processors used in each experiment. They are integer multiples of three (i.e., 3, 6, 9, 12, 15) because a minimum of three processors were needed to execute su2cor: two processors (one on each hypernode) for the doacross loop and one additional processor to execute *corr* as an asymmetric thread in the do-pipe. For the experimental results reported here, an equal number of doall sub-threads were assigned to each of these three "main" threads. The results recorded in each column are speedups as compared with the serial execution of the benchmark.

In the "Doall Only" experiment neither doacross nor pipelined parallelism were enabled. This gives a basis for comparison in that single-level doall parallelism is the standard on most parallel architectures today. The second experiment employed doacross and pipelined parallelism in conjunction with doall parallelism. In this experiment, the set of doall loops executed was the same as the set of doall loops used in the "Doall Only" experiment.

The su2cor benchmark does not perform as well on the Exemplar as it does on the Challenge. This is primarily due to the distributed, shared-memory nature of the Exemplar architecture. For applications such as su2cor with an irregular access pattern, cross-hypernode cache coherence has an impact on performance. This fact is reflected in the results for the "Doall Only" experiments in Table 6.4. As can be seen from these results, the doall version of su2cor does not scale well. In particular, when executed on more than eight processors, there is a drop-off in performance due to the increased coherence traffic between hypernodes (as noted earlier, the SPP-1200 employed in these experiments has eight processors per hypernode).

This reveals one of the strengths of loop-based multi-level parallelism as it was applied in su2cor. The inter-hypernode schedule that we employed allocated a single node-wise thread to each hypernode. Due to the write-access pattern in su2cor (discussed earlier in section 6.2.6), it was possible to create duplicates of the main data structure u which were local to each hypernode. In effect, write accesses to u by a given thread were local to the hypernode on which the thread was executing. Reads, on the other hand, were either local or remote depending on the location accessed. This "privatizing" transformation was made possible by the fact that the Exemplar provides for the allocation of different classes of memory. Among the classes available are thread-private, node-private, far-shared, near-shared, etc. The near-shared memory class allows for the allocation of globally shared, hypernode-local memory, and it was this class of memory that was used in duplicating u.

For the the multi-level experiments conducted on 12 and 15 processors, a similar trend to the doall experiments is exhibited in that no additional speedup was obtained. We speculate that the cause of this lack of speedup is a result of the schedule employed. With 12 processors, the two, node-wise threads each executed four thread-wise threads on each of the two hypernodes. This left four free processors on each hypernode. When *corr* is spawned in the do-pipe, the asymmetric thread also goes thread-wise parallel (as explained in section 6.2.7), and spawns four additional threads on one of the hypernodes. This hypernode then has a total of eight threads active. In our experiments, we allocated at most one thread per processor. As a result, when doacross, do-pipe, asymmetric, and doall parallelism were all active, no more then 12 processors could participate in the computation and still maintain local write access to the "privatized" versions of u. Consequently, with 15 processors at least two of the threads had to be scheduled on a hypernode which did not write to the local copy of u.

As reflected by the results in Table 6.4, the combination of loop-based multi-level parallelism with functional do-pipe parallelism resulted in better speedups and scaling than the "Doall Only" version.

6.4.3 Loops in tomcatv

The computationally key loops in tomcatv are delineated in Table 6.5.

Subroutine	Loop	Depth	% T_{seq}	S/P
MAIN	do140	1	98.4012	S
MAIN	do60	2	64.5203	P
MAIN	do100	2	12.6117	S
MAIN	do120	2	8.67711	S
MAIN	do130	2	7.24051	P
MAIN	do80	2	5.32165	S

Table 6.5: Loops > 5% of Sequential Time

The S/P markings in Table 6.5 denote whether the given loop is parallel (P) or serial (S). The Depth column designates the (interprocedural) nesting level of the given loop, and % T_{seq} is the percentage of the sequential execution time for the loop (including nested loops). As these figures indicate, many of the more time-consuming loops in tomcatv are serial in nature.

6.4.4 Results for tomcatv

The performance of the initial implementation of parallel reductions on the SGI Challenge was very poor due to false-sharing of reduction variables between threads running on different processors. A second version solved this problem by making each element in the expanded reduction variables rxm_e and rym_e the size of a cacheline[7]. The performance of this version was quite good, with a speedup of over 7.2 for the loop, and an overall program speedup of 4.29. Table 6.6 summarizes these results.

The second transformation involved pipelining the outermost time-stepping loop into two stages. In conjunction with the partial privatization of two arrays, the granularity of synchronization was decreased in order to enable the execution of the two stages to be overlapped [64]. Performance results for do-pipe vs. privatized reductions also appear in Table 6.6. The speedup for the do-pipe technique exceeds that achieved for the parallel reductions. This result is partially due to better cache utilization in the do-pipe version.

[7]128 bytes on the Challenge

Transformation	speedup
Polaris	3.49
Polaris (do80 reduction in parallel)	4.29
Polaris (loop do-pipe transformation)	4.96

Table 6.6: Tomcatv speedups on 8 processors

6.5 Conclusion

We have identified several new techniques which result in a significant increase in performance when applied to outer, time-stepping loops in the benchmark codes we have studied. In addition, we have demonstrated that functional, doacross, and doall parallelism can be effectively combined in real scientific applications. As a whole, the results indicate that significant additional parallelism, both beyond and in conjunction with doall parallelism, is available in real application codes.

CHAPTER 7

Conclusion

In the introduction to this thesis we highlighted the importance of parallelizing compiler technology for both present and coming multi-processor systems. In the course of our study of automatic parallelization, we have determined several techniques which can be integrated into a parallelizing compiler. We have demonstrated the applicability and relevance of these techniques in experiments based on the transformation of programs from a cross-section of scientific fields. We have also established the fact that many of these techniques cross computer language boundaries.

In the opening chapter of this work we presented a formalism based on the concept of an associative *coalescing loop operator*. A central result of this research is the development of a general framework in which a wide variety of loop-based computations can be successfully modeled. A second result is the discovery of a number of loops which compute functions which although associative are not commutative.

Several possibilities exist for extending the theory of coalescing operators. As discussed in the Chapter 2, one such extension involves the development and implementation of algorithms which automatically recognize associative coalescing loop operators in a parallelizing compiler such as Polaris. This holds promise of enabling the identification of parallelism based on the symbolic execution of as few as three iterations of a loop.

In Chapter 4 we presented a virtual smorgasbord of parallelizing techniques which significantly improve the performance of representative benchmarks from several fields of science. These techniques were developed with automatability in mind, and one clear research area which lies ahead is the development and implementation of additional algorithms for those techniques which have not yet been automated.

A final note worth mentioning in terms of future research has to do with the computation of closed-forms. As discussed in Chapter 4, the symbolic computation of closed-forms of recurrences applies in more than one application area. In studying this particular method of solving recurrences, it has become clear that a recurrence relation expressed in a program has a fundamental mathematical identity which transcends the particulars of a given implementation expressed using a given language. In other words, a recurrence in a computer program is in fact just an instance of a mathematical formulation for the solution of a problem in science[1].

This observation opens up a very interesting area of research. Think for a moment what a computer program really is... a computer program takes certain inputs and computes outputs which are a function of those inputs. It may involve multiple loops within various subroutines computed over a given period of time-steps, but for a large class of scientific applications fundamentally the output is simply a function of the input.

But wait a moment... *This is the definition of a recurrence!!!* What are the implications of this realization? Only this: many computer programs can be understood as complex recurrence relations packaged in a deterministic context-free language wrapper, and that *just like linear recurrences with constant coefficients*, it may be possible to compute closed-forms for entire programs.

[1]Modulo numerical considerations of the finite representation of course.

APPENDIX A

Coalescing Loop Operators in cSpace

In this appendix we include the C^{++} source code (minus the header files) of the **cSpace** semantic indexing application. This code ©Copyright William Morton Pottenger.

```
#include "ConceptSpace.h"
#include "String.h"

extern int dbx_warn_overflow;

static String version("ConceptSpace1.0");

int
main(int argc, char *argv[])
{
    // HeapStats::start();

    if (argc != 1 && argc != 5 && argc != 6) {
        cerr << "Usage: cSpace [ name num_procs input_file output_file";
        cerr << " [ idx_file ] ]\n";
        exit(1);
    }

    dbx_warn_overflow = 1;

    if (argc == 1) {

ConceptSpace cs;
        String o_rname = "cout";
        cs.compute(o_rname);
    }
    else if (argc == 5) {

        String name = argv[1];
        int threads = atoi(argv[2]);
        String ifile = argv[3];
        String ofile = argv[4];
        String idx_ofile = "";

        ConceptSpace cs(name, ifile, ofile, idx_ofile);

        String o_rname = ofile;

        cs.compute(o_rname, threads);
    }
    else if (argc == 6) {

        String name = argv[1];
        int threads = atoi(argv[2]);
        String ifile = argv[3];
        String ofile = argv[4];
        String idx_ofile = argv[5];

        ConceptSpace cs(name, ifile, ofile, idx_ofile,
                        sO_NOSTORE, dF_RAW, dF_INDEXED);

        String o_rname = ofile;

        cs.compute(o_rname, threads);
    }

    // HeapStats::stop();
    // HeapStats::report(cout);
    // HeapStats::print_memory_leaks(cout);
}

// module ConceptSpace

#include "ConceptSpace.h"

#include <strstream.h>

#include "Collection/BaseMapIter.h"
#include "Collection/Iterator.h"
#include "Collection/KeyDatabase.h"

// KeyDatabase of Term objects

KeyDatabase<String, Term> Terms;

// List of Doc objects

List<Doc> Docs;
```

```
#ifndef __SUNPRO_CC

// Template instantiations for KeyDatabase<String, Term>

template class KeyDatabase<String, Term>;
template class ProtoDatabase<String, Term>;
template class TypedBaseMap<String, Term>;
template class KeyIterator<String, Term>;

#endif

// Implementation

// Constructors

#ifdef __SUNPRO_CC

// Sun-specific iostreams code

// Create a concept space from cin, output to cout, nothing saved

ConceptSpace::ConceptSpace()
{
    _istream = cin.rdbuf();
    _ostream = cout.rdbuf();
    _iformat = _oformat = dF_RAW;
    _max_term_length = 0;
    _sopt = sO_NOSTORE;
}

// Create a named concept space from input in ifile using
// iformat, output to ofile with oformat, and
// save as directed by the argument sopt

ConceptSpace::ConceptSpace(String &name, String &ifile, String &ofile,
                           String &idx_ofile, saveOpt sopt GIV(sO_NOSTORE),
                           docFormat iformat GIV(dF_RAW),
                           docFormat oformat GIV(dF_RAW))
{
    // assign input & output streams
    _ifile.open(ifile,ios::in);
    _istream = _ifile.rdbuf();
    _ofile.open(ofile,ios::out);
    _ostream = _ofile.rdbuf();
    _iformat = iformat;
    _oformat = oformat;
    _name = name;
    _max_term_length = 0;
    _sopt = sopt;

    if (_oformat == dF_INDEXED){
        _idx_ofile.open(idx_ofile,ios::out);
        _idx_ostream = _idx_ofile.rdbuf();
    }
}

#else

// Proposed standard iostreams code

// Create a concept space from cin, output to cout, nothing saved

ConceptSpace::ConceptSpace() : _istream(cin.rdbuf()), _ostream(cout.rdbuf())
{
    _iformat = _oformat = dF_RAW;
    _max_term_length = 0;
    _sopt = sO_NOSTORE;
}

// Create a named concept space from input in ifile using
// iformat, output to ofile with oformat, and
// save as directed by the argument sopt

ConceptSpace::ConceptSpace(String &name, String &ifile, String &ofile,
                           String &idx_ofile, saveOpt sopt GIV(sO_NOSTORE),
                           docFormat iformat GIV(dF_RAW),
                           docFormat oformat GIV(dF_RAW))
{
    // assign input & output streams
    _ifile.open(ifile,ios::in);
    _istream.rdbuf(_ifile.rdbuf());
    _ofile.open(ofile,ios::out);
    _ostream.rdbuf(_ofile.rdbuf());
    _iformat = iformat;
    _oformat = oformat;
    _name = name;
    _max_term_length = 0;
    _sopt = sopt;

    if (_oformat == dF_INDEXED){

        _idx_ofile.open(idx_ofile,ios::out);
        _idx_ostream.rdbuf(_idx_ofile.rdbuf());
    }
}

#endif

ConceptSpace::~ConceptSpace()
{
    if (_ifile.rdbuf()->is_open())
        _ifile.close();
    if (_ofile.rdbuf()->is_open())
        _ofile.close();
    if (_idx_ofile.rdbuf()->is_open())
        _idx_ofile.close();
}

// Return the name of the concept space
String &
ConceptSpace::name()
{
    return _name;
}

extern "C" int m_lock(void);
extern "C" int m_unlock(void);
extern "C" void m_sync(void);
extern "C" int m_set_procs(int);
extern "C" int m_get_myid(void);
extern "C" int m_get_numprocs(void);
extern "C" int m_fork(void (*)(void *, void *, void *, void *),...);

extern "C" int usconfig(int, int);
extern "C" int schedctl (int cmd, int arg1);
extern "C" void perror (const char *s);

void *pmem(size_t, void *, int, size_t, int);

// Compute and output concept space
void
ConceptSpace::compute(String &o_rname, int threads GIV(1))
{
    // cout << "Single threaded malloc status was ";
    // int status = usconfig(14,0);
    // if (status == 14) cout << "on.\n";
    // else cout << "off.\n";

    // Input documents, extract & count terms

    int doc_count = 0;
    int term_count = 0;

    while (!_istream.eof()) {

        List<StringElem> nps_in_doc;

        Doc *doc = new Doc(_iformat,_sopt);
        Docs.ins_last(doc);
        doc->read(_istream, nps_in_doc, term_count);
        doc->index(doc_count, nps_in_doc, _max_term_length);
    }

    // Initialize custom memory manager
    pmem(0, (void *) NULL, -1, 0, 0);

    // Set the number of threads

    if (m_set_procs (threads) == -1)
        perror("ConceptSpace::compute: failed m_set_procs");

    // Fork the similarity computation

    void simComp(void *, void *, void *, void *);

    if (m_fork(simComp, (void *) &doc_count, (void *) &o_rname,
               (void *) &_oformat, (void *) &_max_term_length) == -1)
        perror("ConceptSpace::compute:simComp: failed m_fork");
}

#include <stdio.h>
#include <fcntl.h>
#define MMAP_FILE       "/dev/zero"
#define MMAP_PERM       (O_RDWR)

// Compute the similarities
void
simComp(void *doc_count, void *r_oname, void *oformat, void *max_term_length)
{
    // Code to set the schedule to SGS_GANG on the SGI PC
    // if (schedctl(13,4) == -1) perror("simComp::schedctl failed");

    // Allocate slices of the term-space to each thread

    int myid = m_get_myid();
    int threads = m_get_numprocs();
    int entries = Terms.entries();
    int slice = entries / threads;

    int terms=slice*myid;

    // Assign the 'leftover' iterations to the last thread

    if (myid+1 == threads)
        slice += entries % threads;

    // Open thread-private output stream

    strstream s;
    s << myid << '\0';
    char *buf = s.str();
    String p_oname = *(String *) r_oname + "_" + buf;
    String p_idx_oname = *(String *) r_oname + "_idx_" + buf;
    String p_coocs_oname = *(String *) r_oname + "_coocs_" + buf;
    delete buf;
```

```
FILE *p_ofile = fopen((const char *) p_oname,"w");
FILE *p_idx_ofile = fopen((const char *) p_idx_oname,"w");
FILE *p_coocs_ofile = fopen((const char *) p_coocs_oname,"w");

// ofstream p_ofile;
// p_ofile.open((const char *) p_oname,ios::out);
// ostream p_ostream = p_ofile.rdbuf();

BaseMapIter titer(Terms);

// Each thread iterates to the starting point of its slice

for (int term_cnt=0; titer.valid() && term_cnt<terms; ++titer,++term_cnt) ;

// BMNode &start = titer.current_node();
// Obtain an upper bound on the total number of cooccurrences
// int cooc_count = 0;
// for (term_cnt=0; titer.valid() && term_cnt < slice; ++titer,++term_cnt) {
    // Term &term = *(Term *) titer.current_data();
    // cooc_count += term.cooc_count();
// }
// for (titer.reset(start),term_cnt=0;

m_lock();
    cerr << "Similarity start: " << "tid: " << myid << " slice: " << slice;
    cerr << " terms: " << terms << " num procs: " << threads << "\n";
    // cerr << " cooc count: " << cooc_count << "\n";
m_unlock();

// Open /dev/zero for custom memory allocation
int memfd = open(MMAP_FILE,MMAP_PERM);
if (memfd < 0) {
    m_lock();
        cerr << "Error opening /dev/zero on thread " << myid << ".\n";
        perror("/dev/zero");
        exit(1);
    m_unlock();
}

int totcoocs = 0;

// Each thread computes the similarity for the terms in its slice
size_t alloc=0;
int term_id = terms;
for (term_cnt=0; titer.valid() && term_cnt < slice;
     ++titer,++term_cnt,++term_id) {

    Term &termA = *(Term *) titer.current_data();

    // Set up a private memory pool for cooccurrence and similarity data

    alloc = 384*(termA.cooc_count()+250);
```

```
pmem(0, (void *) NULL, 3, alloc, memfd);

{

Database<String, Cooccurrence> cooccurrences;

BaseMapIter diter(termA.docs());

for ( ; diter.valid(); ++diter) {

    Doc &doc = *(Doc *) diter.current_key()._keyptr;
    int termA_freq = *(IntElem *) diter.current_data();
    Iterator<Term> citer(doc.terms_in_doc());

    for ( ; citer.valid(); ++citer) {

        Term &termB = citer.current();

        if (&termA != &termB) {

            int termB_freq = *termB.docs().find_ref(doc);
            int min_term_freq = termA_freq < termB_freq ?
                                termA_freq : termB_freq;
            Cooccurrence *cooc_ref =
                        cooccurrences.find_ref(termB.key_phrase());
            if (cooc_ref)
                cooc_ref->update(min_term_freq);
            else {

                 ++totcoocs;

                // enclose in transaction which allocates
                // loop-private memory
                cooccurrences.ins(termB.key_phrase(),
                              new Cooccurrence(termB,min_term_freq));
            }
        }
    }
}

fprintf(p_coocs_ofile,"%s: %d\n", termA.phrase_ref(),
        cooccurrences.entries());

BaseMapIter citer(cooccurrences);

for ( ; citer.valid(); ++citer) {

    Cooccurrence &coocAB = *(Cooccurrence *) citer.current_data();
    Term &termB = coocAB.termB();
    // enclose in transaction which allocates shared memory
    coocAB.similarity(termA, termB, *(int *) doc_count);
}
```

```
        // enclose in transaction which allocates loop-private memory
        termA.subset(*(int *) max_term_length);
        termA.output(p_ofile, p_idx_ofile, term_id, *(docFormat *) oformat);
        // cooccurrences.clear();
        // termA.similarities().clear();

        }

        // Release private memory
        pmem(0, (void *) NULL, 4, alloc, 0);
    }

    fprintf(p_coocs_ofile,"\nTerms: %d Total Coocs: %d\n", terms, totcoocs);

    // if (p_ofile.rdbuf()->is_open())
        // p_ofile.close();

    fclose(p_ofile);
    fclose(p_idx_ofile);
    fclose(p_coocs_ofile);

    m_lock();
        cerr << "Similarity finish: " << "tid: " << myid;
        cerr << " output file: " << p_oname << "\n";
    m_unlock();
}

    // Example exception:
    // if (...) throw returnCode(returnCode::FAIL);

    // Code to set the schedule to SGS_GANG on the SGI PC
    // extern "C" int schedctl (int cmd, int arg1);
    // if (schedctl(13,4) == -1) perror("simComp::schedctl failed");;

    // Code to assign each thread to a specific processor
    // extern "C" int sysmp(int, int);
    // int proc = m_get_myid();
    // int status = sysmp(19,proc);
    // if (status == -1) perror("simComp::sysmp failed");;

// module Term

#include <ctype.h>
#include <strstream.h>
#include "Collection/BaseMapIter.h"
#include "Collection/BaseMapNode.h"
#include "Collection/Iterator.h"
#include "Collection/KeyDatabase.h"
#include "Collection/RefDatabase.h"
#include "Collection/RefMap.h"
#include "Term.h"
```

```
// Global KeyDatabase of Term objects (defined in ConceptSpace.cc)

extern KeyDatabase<String, Term> Terms;

// Implementation

Term::Term(char *phrase, int words_in_phrase, int doc_count, Doc &doc)
{
    _phrase = new String;
    _phrase->absorb(phrase);
    _words_in_phrase = words_in_phrase;
    _current_doc = doc_count;
    _id = 0;
    _doc_freq = 1;
    _sum_term_freq = 1;
    // _lock = 0;
    _term_freq = new IntElem(1);
    _docs.ins(doc, _term_freq);
    _w_factor = _weight = -1.0;
}

// Term::Term(const Term &term)
// {
    // _words_in_phrase = term._words_in_phrase;
    // _current_doc = term._current_doc;
    // _id = term._id;
    // _doc_freq = term._doc_freq;
    // _sum_term_freq = term._sum_term_freq;
    // _lock = 0;
    // _term_freq = term._term_freq;
    // _w_factor = term._w_factor;
    // _weight = term._weight;
// }

Term::~Term()
{
    if (_docs.entries()) ;
        // _docs.clear();
    // if (_cooccurrences.entries()) ;
        // _cooccurrences.clear();
    if (_similarities.entries()) ;
        // _similarities.clear();
}

// Lock *this in preparation for synchronized access
// void
// Term::lock()
// {
    // while (test_and_set(&_lock, (unsigned long) 1) != 0) ;
// }
// Unlock *this
// void
// Term::unlock()
// {
    // test_and_set(&_lock, (unsigned long) 0);
// }

// Return reference to cooccurrences
// Database<String, Cooccurrence> &
// Term::cooccurrences()
// {
    // return _cooccurrences;
// }

// Return an upper bound on the number of cooccurring terms
int
Term::cooc_count()
{
    int cooc_count = 0;
    BaseMapIter diter(_docs);

    for ( ; diter.valid(); ++diter) {

        Doc &doc = *(Doc *) diter.current_key()._keyptr;

        cooc_count += doc.terms_in_doc().entries();
    }

    return cooc_count;
}

// Return reference do docs
Map<Doc, IntElem> &
Term::docs()
{
    return _docs;
}

// Return reference to similarity data structure
Database<int, RefList<Term> > &
Term::similarities()
{
    return _similarities;
}

Term *
Term::clone() const
{
    return new Term(*this);
}

Listable *
```

```
Term::listable_clone() const
{
    return new Term(*this);
}

// Term::output(ostream & o, ostream &idx_o, docFormat oformat GIV(df_RAW))

// Output co-occurrences of *this
void
Term::output(FILE *o, FILE *idx_o, int term_id GIV(0),
             docFormat oformat GIV(df_RAW))
{
    // static int term_id = sid;

    switch (oformat) {

      case dF_INDEXED:

          // p_assert(term_id_ref,"Term::output(): NULL term_id_ref.\n");

          if (!_id) {

              _id = ++term_id;
              // idx_o << phrase_ref() << "\n";
              fprintf(idx_o,"%s\n", phrase_ref());
          }
          // o << '#' << _id-1 << "\n";
          fprintf(o,"#%d\n",_id-1);
          break;

      default:

          // o << phrase_ref() << "\n";
          fprintf(o,"%s\n", phrase_ref());
    }

    int cooc_count = 0;
    BaseMapIter simi(_similarities);
    simi.reset_last();

    for ( ; simi.valid() && cooc_count < MAXCOOCOUT; --simi) {

        RefList<Term> &sterms = *(RefList<Term> *) simi.current_data();

        if (sterms.entries()) {

            BMKey &similarity_key = simi.current_key();

            switch (oformat) {

              case dF_INDEXED:
                  break;

                default:

                  // o << "  " << *((int *) similarity_key._keyptr) << "\n";
                  fprintf(o,"  %d\n",*((int *) similarity_key._keyptr));
                  break;
            }

            Iterator<Term> ctermi(sterms);

            for ( ; ctermi.valid() && cooc_count < MAXCOOCOUT; ++ctermi) {

                cooc_count++;

                Term &termB = ctermi.current();

                switch (oformat) {

                  case dF_INDEXED:

                      if (!termB._id) {

                          termB._id = ++term_id;
                          // idx_o << termB.phrase_ref() << "\n";
                          fprintf(idx_o,"%s\n", termB.phrase_ref());
                      }
                      // o << termB._id-1 << " ";
                      // o << *((int *) similarity_key._keyptr) << "\n";
                      fprintf(o,"%d %d\n",  termB._id-1,
                              *((int *) similarity_key._keyptr));
                      break;

                  default:

                      // o << "    " << termB.phrase_ref() << "\n";
                      fprintf(o,"    %s\n", termB.phrase_ref());
                      break;
                }
            }
        }
    }
}

// Return reference to original (mixed-case) phrase
const String &
Term::phrase()
{
    return *_phrase;
}

// Return char * reference to phrase which makes up *this
```

```
const char *
Term::phrase_ref()
{
    return (const char *) phrase();
}

// Return reference to String which makes up *this (the phrase is used
// as the key for the node holding *this in the global Terms database)

const String &
Term::key_phrase()
{
    return *((String *) Terms.key_ref(*this));
}

// Return pointer to phrase which makes up *this
// String *
// Term::grab_phrase()
// {
    // if (_phrase) {

        // String *phrase = _phrase;
        // _phrase = NULL;
        // return phrase;
    // }
    // else
        // return NULL;
// }

void
Term::print(ostream & o) const
{
    // o << "Term: " << phrase() << "\n";
}

// As indicated by sopt, store the concept spaces & documents/ids
// void
// Term::save_docinfo(Doc *doc, saveOpt sopt, int doc_count)
// {
//    if (sopt & sO_STOREOSMASK) {
//
//        // Store in object store
//
//        switch (sopt) {
//
//            case sO_STOREALLOS: sO_STOREDOCOS:
//
//                // allocate with ostore new to save Doc objects
//                break;
//
//            case sO_STORECSDOCIDOS: sO_STOREDOCIDOS:
//
//                // allocate with ostore new to save Doc ids only
//                break;
//
//     default:
//
// // do nothing
// break;
//        }
//    }
//
//
//    if (sopt & sO_STORESTMASK) {
//
//        // Store for later output to ostream
//
//        String id;
//        strstream s;
//
//        switch (sopt) {
//
//            case sO_STOREALLST: sO_STOREDOCST:
//
//                id.absorb(doc->id().grab());
//
//                if (!id.defined()) {
//
//                    s << doc_count;
//                    id = s.str();
//                }
//                _docs.ins(&id, new Doc(doc));
//                break;
//
//            case sO_STORECSDOCIDST: sO_STOREDOCIDST:
//
//                id.absorb(doc->id().grab());
//
//                if (!id.defined()) {
//
//                    s << doc_count;
//                    id = s.str();
//                }
//                _docs.ins(&id, (Doc *) NULL);
//                break;
//
//            default:
//
// // do nothing
// break;
//        }
//    }
// }
// Perform subset operation on *this term
void
Term::subset(int max_term_length)
{
    int terms_out = 0;
    BaseMapIter simiA(_similarities);
    BaseMapIter simiB(_similarities);
    RefMap<Term, RefList<Term> > pruned_terms;

    for (simiA.reset_last(); simiA.valid() && terms_out < MAXCOOCOUT; --simiA) {

        RefList<Term> &termsA = *(RefList<Term>*) simiA.current_data();
        Iterator<Term> termAi(termsA);

        for ( ; termAi.valid() && terms_out < MAXCOOCOUT; ++termAi) {

            Term &termA = termAi.current();
            const String &phraseA = termA.key_phrase();
            Boolean pruned = False;
            int terms_compared = 0;

            for (simiB.reset_last();
                 simiB.valid() && terms_compared < max_term_length*MAXCOOCOUT;
                 --simiB) {

                RefList<Term> &termsB = *(RefList<Term>*) simiB.current_data();
                Iterator<Term> termBi(termsB);

                for ( ; termBi.valid() &&
                        terms_compared++ < max_term_length*MAXCOOCOUT;
                      ++termBi) {

                    Term &termB = termBi.current();
                    const String &phraseB = termB.key_phrase();

                    if (termA._words_in_phrase < termB._words_in_phrase) {

                        if (!strncmp(phraseA, phraseB, phraseA.len())) {

                            pruned_terms.ins(termA,termsA);
                            pruned = True;
                            break;
                        }
                    }
                }

                if (pruned) break;
            }

            if (!pruned) ++terms_out;
        }
    }
    BaseMapIter piter(pruned_terms);

    for ( ; piter.valid(); ++piter) {

        Term &term = *(Term *) piter.current_key()._keyptr;
        RefList<Term> &terms = *(RefList<Term> *) piter.current_data();
        terms.del(term);
    }
}

// Return term frequency
int
Term::term_freq()
{
    return *_term_freq;
}

// Update term & doc frequencies
void
Term::update(char *phrase, int doc_count, Doc &doc)
{
    if (doc_count != _current_doc) {

        _doc_freq++;
        _current_doc = doc_count;
        _term_freq = new IntElem(0);
        _docs.ins(doc, _term_freq);
    }

    _term_freq->value(*_term_freq+1);
    _sum_term_freq++;

    // Simple heuristic to choose the phrase which starts
    // with a capital and maximizes lower case letters

    if (isupper(phrase[0]))

        if (!isupper((*_phrase)[0]) || *_phrase < phrase) {

            delete _phrase;
            _phrase = new String;
            _phrase->absorb(phrase);
        }
        else
            delete phrase;
    else
        delete phrase;
}


// module Doc
```

```
#include <ctype.h>
#include "Doc.h"
#include "Term.h"
#include "Collection/Database.h"
#include "Collection/RefDatabase.h"
#include "Collection/BaseMapIter.h"
#include "Collection/Iterator.h"
#include "Collection/Mutator.h"

// Global KeyDatabase of Doc objects (defined in ConceptSpace.cc)

extern KeyDatabase<String, Term> Terms;

#ifndef __SUNPRO_CC

#endif

// Implementation

Doc::Doc(docFormat iformat, saveOpt sopt)
{
    _iformat = iformat;
    _sopt = sopt;
}

Doc::Doc(Doc *doc)
{
    _iformat = doc->_iformat;
    _sopt = doc->_sopt;
    _title.absorb(doc->_title.grab());
    _body.absorb(doc->_body.grab());

    Iterator<StringElem> aui(doc->_authors);

    for ( ; aui.valid(); ++aui) {

        StringElem *se = new StringElem;
        se->absorb(aui.current().grab());
        _authors.ins_last(se);
    }
}

Doc::~Doc()
{
    if (_authors.entries()) ;
        _authors.clear();
    if (_terms_in_doc.entries()) ;
        _terms_in_doc.clear();
}

// Return reference to id of *this
```

```
String &
Doc::id()
{
    return _doc_id;
}

// Return reference to _terms_in_doc
RefList<Term> &
Doc::terms_in_doc()
{
    return _terms_in_doc;
}

// Perform indexing
void
Doc::index(int &doc_count, List<StringElem> &nps_in_doc, int &max_term_length)
{
    ++doc_count;

    Mutator<StringElem> termi(nps_in_doc);

    for ( ; termi.valid(); ++termi) {

        char *phrase = termi.current().grab();
        String *term_key = new String(upcase_ch(phrase));
        Term *term_ref = Terms.find_ref(*term_key);

        if (term_ref) {

            term_ref->update(phrase, doc_count, *this);
            if (!_terms_in_doc.member(*term_ref))
                _terms_in_doc.ins_last(*term_ref);
            delete term_key;
        }
        else {

            int num_words = 1; char *ptr = phrase;
            while (*ptr)
                if (isspace(*ptr++)) ++num_words;
            if (num_words > max_term_length) max_term_length = num_words;
            num_words *= num_words;
            Term &term = Terms.ins(term_key,
                                   new Term(phrase, num_words, doc_count, *this));
            _terms_in_doc.ins_last(term);
        }
    }
}

// Required clone() method
Listable *
Doc::listable_clone() const
{
    // stub
    return NULL;
}

// Required print() method
void
Doc::print(ostream & o) const
{
    o << "Doc: " << _doc_id << "\n";
}

// At this point in time, the input is expected to be a list of terms,
// one per line, where each group of terms (from a single document)
// is separated from the next group by a single (blank) line.

// Input one document
void
Doc::read(istream &i, List<StringElem> &nps_in_doc, int &term_count)
{
    char buf[MAXLINE];

    buf[0] = '\0';

    i.getline(buf,MAXLINE);

    while (buf[0]) {

        ++term_count;
StringElem *np = new StringElem(buf);
        nps_in_doc.ins_last(np);
        buf[0] = '\0';
        i.getline(buf,MAXLINE);
    }
}

// module Cooccurrence

#include <math.h>
#include "Collection/RefList.h"
#include "Cooccurrence.h"
#include "Term.h"

// Global KeyDatabase of Term objects (defined in ConceptSpace.cc)

extern KeyDatabase<String, Term> Terms;

#ifndef __SUNPRO_CC

#endif

// Implementation
```

```
Cooccurrence::Cooccurrence(Term &termB, int min_term_freq)
{

    _termB_ref = &termB;
    _intersect_count = 1;
    _sum_min_term_freq = min_term_freq;
    _invalid = False;
}

Cooccurrence::~Cooccurrence()
{
    // nothing to do
}

// Invalidate *this
void
Cooccurrence::invalidate()
{
    _invalid = True;
}

// Required clone() method
Listable *
Cooccurrence::listable_clone() const
{
    // stub
    return NULL;
}

// Required print() method
void
Cooccurrence::print(ostream & o) const
{
    cout << "Cooccurrence: " << *this << "\n";
}

// Compute similarity
void
Cooccurrence::similarity(Term &termA, Term &termB, int doc_count)
{
    if (termA._weight < 0 && termA._w_factor < 0) {

        // calculate termA's weight and weighting factor

        double oratio = (double) doc_count / (double) termA._doc_freq;
        termA._weight = termA._sum_term_freq *
                        log10(oratio * (double) termA._words_in_phrase);
        termA._w_factor = log10(oratio) / log10((double) doc_count);
    }

    double termB_wf = termB._w_factor;
    if (termB_wf < 0) {

        // calculate a private copy of termB's weighting factor

        double oratio = (double) doc_count / (double) termB._doc_freq;
        termB_wf = log10(oratio) / log10((double) doc_count);
    }

    double iratio = (double) doc_count / (double) _intersect_count;

    double similarityAB = (
                           (
                             ( (double) _sum_min_term_freq *
                               log10(iratio * (double) termA._words_in_phrase)
                             )
                             / termA._weight
                           )
                           * termB_wf
                         );

    if (similarityAB > SIMTHRESH) {

        int _similarity = (int) ( (double) SIMULT * similarityAB );

        RefList<Term> *siml = termA._similarities.find_ref(_similarity);
        if (!siml) {

            siml= new RefList<Term>;
            termA._similarities.ins(_similarity, siml);
        }

        siml->ins_last(termB);
    }
}

// Return a reference to co-occurring term
// Term &
// Cooccurrence::termB(const String &termB_phrase)
// {
    // if (!_termB_ref)
        // _termB_ref = Terms.find_ref(termB_phrase);

    // return *_termB_ref;
// }

// Return a reference to co-occurring term
Term &
Cooccurrence::termB()
{
    return *_termB_ref;
}
```

```
// Update _intersect_count & _sum_min_term_freq
void
Cooccurrence::update(int min_term_freq)
{
    _intersect_count++;
    _sum_min_term_freq += min_term_freq;
}
```

APPENDIX B

Coalescing Loop Operators in CHOLESKY

In this appendix we include a partial listing of the sparse HPF-2 benchmark CHOLESKY. The purpose of including this code is to demonstrate the parallelization of an associative coalescing loop operator in a multiple-exit loop. In the listing below, loop 400 in subroutine GENQMD has been transformed based on the algorithms described in Chapter 2 section 2.3.4 and Chapter 4 section 4.4.6.

```
* This is a modification of cholesky.f by Bill Pottenger and Yuan Lin
c
c  Finds the Cholesky factor of a sparse symmetric matrix
      program sparchol
c

**bp Doubled sizes of MAXN, MAXNROWI, MAXNADJ, and MAXNZ
**bp See http://math.nist.gov/MatrixMarket for additional input files

      integer
     .     MAXN, MAXNROWI, MAXNADJ, MAXNSUB, MAXNZ
      parameter
     .     (
     .     MAXN=40000, MAXNROWI=1050000, MAXNADJ=2020000,
     .     MAXNSUB=10000000, MAXNZ=6004000
     .     )
c
```

```
      character
     .     title*72, key*8, mxtype*3, ptrfmt*16, indfmt*16,
     .     valfmt*20, rhsfmt*20
      integer
     .     totcrd, ptrcrd, indcrd, valcrd, rhscrd, nrow, ncol,
     .     nnzero, neltlvl
      integer
     .     colptr(MAXN+1), rowind(MAXNROWI), nu(MAXN), lp(MAXN),
     .     neqns, xadj(MAXN+1), xadjc(MAXN+1), adjncy(MAXNADJ),
     .     adjncyc(MAXNADJ), xlnz1(MAXN+1), xnzsub1(MAXN),
     .     nzsub1(MAXNSUB), xlnz2(MAXN+1), xnzsub2(MAXN),
     .     nzsub2(MAXNSUB), fstdau(MAXN), ndaugh(MAXN), sister(MAXN),
     .     mother(MAXN), stk(MAXN), post(MAXN), postinv(MAXN),
     .     nsu, isu(MAXN), snhead(MAXN), nafter(MAXN), xmylnz(MAXN+1)
      integer
     .     deg(MAXN), qsize(MAXN), marker(MAXN), nbrhd(MAXN),
     .     perm(MAXN), invp(MAXN), rchset(MAXN), nofsub,
     .     rchlnk(MAXN), mrglnk(MAXN), qlink(MAXN), flag
      real
     .     mylnz(MAXNZ), denupd(MAXN)
      integer
     .     i, j, k, adjp, nsub, nlnz, pp, node, sp, ii, jj, iii,
     .     s, inzsub, l, tgtnum, jp, ksize, tmp
      real
     .     rand
c
      real tstart, tend

ccc used by the runtime test
ccc
      pointer (ptrposmylnz,posmylnz)
      integer*4 posmylnz(256,*)

ccc  1: read min  2: read max   3: write min    4: write max

      integer threadid
      integer numthreads
      integer failed
      logical boo1, boo2


ccc
c  Read in symmetric structure from Boeing-Harwell file
      read (*,'(a72,a8/5i14/a3,11x,4i14/2a16,2a20)')
     .     title, key, totcrd, ptrcrd, indcrd, valcrd, rhscrd,
     .     mxtype, nrow, ncol, nnzero, neltlvl, ptrfmt, indfmt, valfmt,
     .     rhsfmt


      if (mxtype(2:3) .ne. 'SA') then
         write(15,*) 'Bad matrix type:',mxtype
         stop
      end if
      if (ncol .gt. MAXN) then
         write(15,*) 'ncol too big:',ncol
         stop
      end if
      if (nnzero .gt. MAXNROWI) then
         write(15,*) 'nnzero too big:',nnzero
         stop
      end if
      read (*,ptrfmt) (colptr(i), i=1,ncol+1)
      read (*,indfmt) (rowind(i), i=1,nnzero)
c
cTTT
      call my_time(tstart)
cTTT

c  Find adjacency structure
      neqns = ncol
      do 1001 j = 1,neqns
         nu(j) = 0
 1001 continue
      do 1002 j = 1,neqns
         call SORTS1(colptr(j+1)-colptr(j),rowind(colptr(j)))
         if (rowind(colptr(j)) .ne. j) then
            write(15,*) 'Unable to find diagonal in col',j
            stop
         end if
         do 1003 i = colptr(j)+1,colptr(j+1)-1
            k = rowind(i)
            nu(k) = nu(k) + 1
 1003    continue
 1002 continue
      xadj(1) = 1
      do 1004 j = 1,neqns
         xadj(j+1) = xadj(j) + colptr(j+1) - colptr(j) - 1 + nu(j)
         lp(j) = xadj(j) + nu(j)
         nu(j) = 0
 1004 continue
      if (xadj(neqns+1)-1 .gt. MAXNADJ) then
         write(15,*) 'Not enough room in adjncy; need:',xadj(neqns+1)-1
         stop
      end if
      do 1005 j = 1,neqns
         adjp = lp(j)
         do 1006 i = colptr(j)+1,colptr(j+1)-1
            k = rowind(i)
            adjncy(xadj(k)+nu(k)) = j
            nu(k) = nu(k) + 1
            adjncy(adjp) = k
            adjp = adjp + 1
 1006    continue
 1005 continue
```

```
      do 1007 j = 1,neqns+1
         xadjc(j) = xadj(j)
 1007 continue
      do 1008 j = 1,xadj(neqns+1)-1
         adjncyc(j) = adjncy(j)
 1008 continue
c
c  Find new order
      call GENQMD(neqns,xadj,adjncy,perm,invp,deg,
     .     marker,rchset,nbrhd,qsize,qlink,nofsub)
c
c  Determine fill-in
      nsub = MAXNSUB
      call SMBFCT(neqns,xadjc,adjncyc,perm,invp,
     .     xlnz1,nlnz,xnzsub1,nzsub1,nsub,
     .     rchlnk,mrglnk,marker,flag)
      if (flag .gt. 0) then
         write(15,*) 'Not enough room in nzsub:',nsub
         stop
      end if
      if (nlnz .gt. MAXNZ) then
         write(15,*) 'Not enough room in lnz:',nlnz
         stop
      end if
c
c  Find e-tree
      do 1009 j = 1,neqns
         fstdau(j) = 0
         ndaugh(j) = 0
 1009 continue
      do 1010 j = 1,neqns-1
         mother(j) = nzsub1(xnzsub1(j))
         if (ndaugh(mother(j)) .ne. 0) then
            sister(j) = fstdau(mother(j))
         else
            sister(j) = 0
         end if
         fstdau(mother(j)) = j
         ndaugh(mother(j)) = ndaugh(mother(j)) + 1
 1010 continue
      mother(neqns) = 0
      sister(neqns) = 0
c
c  Find postorder
      pp = 0
      node = neqns
      sp = 0
 130  if (ndaugh(node) .ne. 0) go to 170
      go to 150
 140  if (sister(node) .ne. 0) go to 160
      if (sp .eq. 0) go to 180
      node = stk(sp)
      sp = sp - 1
 150  pp = pp + 1
      post(pp) = node
      go to 140
 160  node = sister(node)
      go to 130
 170  sp = sp + 1
      stk(sp) = node
      node = fstdau(node)
      go to 130
 180  continue
c
c  Find supernodes
      nsu = 1
      ii = 1
c
c  Consider a supernode with leading node post(ii)
 290  isu(nsu) = ii
      jj = ii + 1
      iii = ii
c
c  Try to add j to the supernode
 260  if (jj .gt. neqns) go to 230
      j = post(jj)
      i = post(iii)
      if (j .ne. mother(i) .or. i .ne. fstdau(j) .or. sister(i) .ne. 0)
     .     go to 230
      if (xlnz1(i+1)-xlnz1(i) .ne. xlnz1(j+1)-xlnz1(j)+1) go to 230
      l = xnzsub1(j)
      do 240 k = xnzsub1(i)+1,xnzsub1(i)+xlnz1(i+1)-xlnz1(i)-1
         if (nzsub1(k) .ne. nzsub1(l)) go to 230
         l = l + 1
 240  continue
c
c  Everything succeeded: we can add to the supernode
      jj = jj + 1
      iii = iii + 1
      go to 260
c
c  We can't add any more to the supernode
 230  if (jj .gt. neqns) go to 280
      ii = jj
      nsu = nsu + 1
      go to 290
c
c  Done: wrap up
 280  isu(nsu+1) = neqns+1
c
c  Find postinv
      do 1011 j = 1,neqns
         postinv(post(j)) = j
 1011 continue
```

```
c
c  Determine structure with new postorder numbering
      xlnz2(1) = 1
      inzsub = 1
      xmylnz(1) = 1
c
c  Consider each supernode
      do 1012 s = 1,nsu
         j = isu(s)
         xnzsub2(j) = inzsub
         jj = post(j)
         do 1013 k = xnzsub1(jj),xnzsub1(jj)+xlnz1(jj+1)-xlnz1(jj)-1
            nzsub2(inzsub) = postinv(nzsub1(k))
            inzsub = inzsub + 1
 1013    continue
c
c  Consider each column in the supernode
         do 1014 j = isu(s),isu(s+1)-1
            jj = post(j)
            xlnz2(j+1) = xlnz2(j) + xlnz1(jj+1) - xlnz1(jj)
            xmylnz(j+1) = xmylnz(j) + xlnz1(jj+1) - xlnz1(jj) + 1
            xnzsub2(j) = xnzsub2(isu(s)) + j - isu(s)
 1014    continue
 1012 continue
c
c  Initialize matrix with random values
      do 1015 j = 1,neqns
         do 1016 i = xmylnz(j)+1,xmylnz(j+1)-1
*e
            if (i.le.0) then
               print *,'i=',i,'<0',' in j=',j
               stop
            endif
*e
            mylnz(i) = 2.0 * rand(0) - 1.0
 1016    continue
*e
            if (xmylnz(j).le.0) then
               print *,'i=',i,'<0',' in j=',j
               stop
            endif
*e
         mylnz(xmylnz(j)) = 8.0 * float(neqns+j)
 1015 continue
c
c  Write out matrix
**bp      do j = 1,neqns
**bp         write(*,'(4e20.10)') (mylnz(i),i=xmylnz(j),xmylnz(j+1)-1)
**bp      end do
**bp      write(*,*) 'BETWEEN'
* skip deadcode
          print *, '1:mylnz:',mylnz(1)
```

```
*
c
c  Determine head of each supernode and number after each node
      do 1017 s = 1,nsu
         do 1018 j = isu(s),isu(s+1)-1
            snhead(j) = isu(s)
            nafter(j) = isu(s+1) - 1 - j
 1018    continue
 1017 continue
c
c  Perform Cholesky factorization
      do 1022 i = 1,neqns
         denupd(i) = 0.0
 1022 continue


ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
ccc runtime test version
ccc
ccc
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

      numthreads = mp_numthreads()

c      print *, numthreads, " threads avaiable"

ccc
      ptrposmylnz = malloc(numthreads*4*256)


      do 1019 k = 1,neqns
ccc          print *, k
         call normalize(k,xmylnz,mylnz)

         if (nafter(k) .ne. 0) then

        ksize = xlnz2(k+1) - xlnz2(k) + 1


c         print *, "getsize"
         do i = 1, numthreads
          posmylnz(1,i) = MAXNZ+1
          posmylnz(3,i) = MAXNZ+1
          posmylnz(2,i) = 1-1
          posmylnz(4,i) = 1-1
         enddo
c         print *, "getsize 1"
C$DOACROSS
 do j = k+1, k+nafter(k)
    call sizeupdd(j-k+1,ksize,
     .        xmylnz(k),xmylnz(j),
     .        posmylnz(1,mp_my_threadnum()+1))
```

```
         enddo


       failed = 0
       do i = 1, numthreads
c          print *,"============"
c          print *,minposmylnz(1,i),maxposmylnz(1,i)
c          print *,minposmylnz(2,i),maxposmylnz(2,i)
c          print *,"============"
          do j = 1, numthreads
cc i write -> j read
             if ( i .ne. j ) then
                boo1= posmylnz(4,i).lt.posmylnz(1,j)
                boo2= posmylnz(3,i).gt.posmylnz(2,j)
                if ( boo1 .or. boo2 ) then
                else
                     failed = 1
                     goto 2511
                endif
             endif
cc i write -> j write
             if ( j .gt. i ) then
                boo1= posmylnz(4,i).lt.posmylnz(3,j)
                boo2= posmylnz(3,i).gt.posmylnz(4,j)
                if ( boo1 .or. boo2 ) then
                else
                     failed = 1
                     goto 2511
                endif
             endif
          enddo
        enddo

 2511   continue



CC  print *, "failed:", failed

C$DOACROSS IF (failed .eq. 0), local(j)
            do 1020 j = k+1,k+nafter(k)
               call upddense(mylnz(xmylnz(k)),mylnz(xmylnz(j)),
     .                       j-k+1,ksize)
 1020       continue

        else

         do i = 1, numthreads
          posmylnz(1,i) = MAXNZ+1
          posmylnz(3,i) = MAXNZ+1
          posmylnz(2,i) = 1-1
          posmylnz(4,i) = 1-1
         enddo

            tmp = xnzsub2(k)
C$DOACROSS local(jp,j,tgtnum)
            do jp = xnzsub2(k),xnzsub2(k)+xlnz2(k+1)-xlnz2(k)-1
               tgtnum = jp-tmp+1
               j = nzsub2(jp)
               call szupdate(k,j,xmylnz(snhead(k)),
     .                     xmylnz(j),tgtnum,snhead,xlnz2,
     .                     xnzsub2,nzsub2,nafter,
     .        posmylnz(1,mp_my_threadnum()+1))
            enddo


       failed = 0
       do i = 1, numthreads
          do j = 1, numthreads
cc i write -> j read
              if ( i .ne. j ) then
                 boo1= posmylnz(4,i).lt.posmylnz(1,j)
                 boo2= posmylnz(3,i).gt.posmylnz(2,j)
                 if ( boo1 .or. boo2 ) then
                 else
                      failed = 1
c       print *,"w>r",i,j,minposmylnz(2,i),maxposmylnz(2,i)
c       print *, "     ", minposmylnz(1,j),maxposmylnz(1,j)
                      goto 2521
                 endif
              endif
cc i write -> j write
              if ( j .gt. i ) then
                 boo1= posmylnz(4,i).lt.posmylnz(3,j)
                 boo2= posmylnz(3,i).gt.posmylnz(4,j)
                 if ( boo1 .or. boo2 ) then
                 else
                      failed = 1
c              print *,"w>w",i,j,minposmylnz(2,i),maxposmylnz(2,i)
c              print *,"    ", minposmylnz(2,j),maxposmylnz(2,j)
                      goto 2521
                 endif
              endif
          enddo
        enddo

 2521   continue



cc  print *, "failed 1021:", failed

         tmp = xnzsub2(k)
C$DOACROSS if ( failed .eq. 0 ), local(jp,j,tgtnum,denupd)
```

```
            do 1021 jp = xnzsub2(k),xnzsub2(k)+xlnz2(k+1)-xlnz2(k)-1
               tgtnum = jp - tmp +1
               j = nzsub2(jp)
               call update(k,j,mylnz(xmylnz(snhead(k))),
     .                     mylnz(xmylnz(j)),tgtnum,snhead,xlnz2,
     .                     xnzsub2,nzsub2,nafter,denupd)
 1021       continue

         end if

 1019 continue

ccc Free CParray & SHarray
ccc
ccc
      call free(ptrposmylnz)
ccc wcnt should not be freed until program ends




cTTT
       call my_time(tend)
       print *,'Time:',tend-tstart,' sec.'
cTTT
c
c  Write out factor
c      do j = 1,500
c         write(*,'(4e20.10)') (mylnz(i),i=xmylnz(j),xmylnz(j+1)-1)
c      end do
c
* skip deadcode
          print *,'1:mylnz:',mylnz(1)
*
      end


C----- SUBROUTINE GENQMD
C****************************************************************
C****************************************************************
C**********    GENQMD ..... QUOT MIN DEGREE ORDERING    *********
C****************************************************************
C****************************************************************
C
C     PURPOSE - THIS ROUTINE IMPLEMENTS THE MINIMUM DEGREE
C        ALGORITHM.  IT MAKES USE OF THE IMPLICIT REPRESENT-
C        ATION OF THE ELIMINATION GRAPHS BY QUOTIENT GRAPHS,
C        AND THE NOTION OF INDISTINGUISHABLE NODES.
C        CAUTION - THE ADJACENCY VECTOR ADJNCY WILL BE
C        DESTROYED.
C
C     INPUT PARAMETERS -
```

```
C        NEQNS - NUMBER OF EQUATIONS.
C        (XADJ, ADJNCY) - THE ADJACENCY STRUCTURE.
C
C     OUTPUT PARAMETERS -
C        PERM - THE MINIMUM DEGREE ORDERING.
C        INVP - THE INVERSE OF PERM.
C
C     WORKING PARAMETERS -
C        DEG - THE DEGREE VECTOR. DEG(I) IS NEGATIVE MEANS
C               NODE I HAS BEEN NUMBERED.
C        MARKER - A MARKER VECTOR, WHERE MARKER(I) IS
C               NEGATIVE MEANS NODE I HAS BEEN MERGED WITH
C               ANOTHER NODE AND THUS CAN BE IGNORED.
C        RCHSET - VECTOR USED FOR THE REACHABLE SET.
C        NBRHD - VECTOR USED FOR THE NEIGHBORHOOD SET.
C        QSIZE - VECTOR USED TO STORE THE SIZE OF
C               INDISTINGUISHABLE SUPERNODES.
C        QLINK - VECTOR TO STORE INDISTINGUISHABLE NODES,
C               I, QLINK(I), QLINK(QLINK(I)) ... ARE THE
C               MEMBERS OF THE SUPERNODE REPRESENTED BY I.
C
C     PROGRAM SUBROUTINES -
C        QMDRCH, QMDQT, QMDUPD.
C
C*****************************************************************
C
      SUBROUTINE  GENQMD ( NEQNS, XADJ, ADJNCY, PERM, INVP, DEG,
     1                     MARKER, RCHSET, NBRHD, QSIZE, QLINK,
     1                     NOFSUB )
C
C*****************************************************************
C
         INTEGER ADJNCY(1), PERM(1), INVP(1), DEG(1), MARKER(1),
     1           RCHSET(1), NBRHD(1), QSIZE(1), QLINK(1)
         INTEGER XADJ(1), INODE, IP, IRCH, J, MINDEG, NDEG,
     1           NEQNS, NHDSZE, NODE, NOFSUB, NP, NUM, NUMP1,
     1           NXNODE, RCHSZE, SEARCH, THRESH
         integer cpunum, loopsize, upper, lower
         integer subsize, subnum, blocksize, blocknum
         integer start, last
         integer ith,jth,kth, lastith, lastjth
         integer geti
         volatile geti, node, lastith, lastjth
 integer mindeg0(5,1000)
         integer nnode, nndeg

C
C*****************************************************************
C
C        ----------------------------------------------------
C        INITIALIZE DEGREE VECTOR AND OTHER WORKING VARIABLES.
C        ----------------------------------------------------
```

```
         MINDEG = NEQNS
         NOFSUB = 0
         DO 100 NODE = 1, NEQNS
            PERM(NODE) = NODE
            INVP(NODE) = NODE
            MARKER(NODE) = 0
            QSIZE(NODE)  = 1
            QLINK(NODE)  = 0
            NDEG = XADJ(NODE+1) - XADJ(NODE)
            DEG(NODE) = NDEG
            IF ( NDEG .LT. MINDEG )  MINDEG = NDEG
  100    CONTINUE
         NUM = 0
C        -----------------------------------------------------
C        PERFORM THRESHOLD SEARCH TO GET A NODE OF MIN DEGREE.
C        VARIABLE SEARCH POINTS TO WHERE SEARCH SHOULD START.
C        -----------------------------------------------------
  200    SEARCH = 1
            THRESH = MINDEG
            MINDEG = NEQNS
  300       NUMP1 = NUM + 1
               IF ( NUMP1 .GT. SEARCH )  SEARCH = NUMP1

cly               DO 400 J = SEARCH, NEQNS
cly                  NODE = PERM(J)
cly                  IF ( MARKER(NODE) .LT. 0 )  GOTO 400
cly                     NDEG = DEG(NODE)
cly                     IF ( NDEG .LE. THRESH )  GO TO 500
cly                     IF ( NDEG .LT. MINDEG )  MINDEG =  NDEG
cly  400          CONTINUE


 lower = search
 upper = neqns
         cpunum = mp_numthreads()
c  cpunum = 4
         loopsize = upper - lower + 1
         subsize = 64
         blocksize = subsize * cpunum
         subnum = ( loopsize + subsize - 1 ) / subsize
         last = mod( subnum, cpunum )
         if ( last .eq. 0 ) last = cpunum

         geti = upper + 1
 lastith = last
 lastjth = (subnum + cpunum - 1)/cpunum
         if ( lastjth .gt. 1000 ) stop
C$DOACROSS LOCAL(ith,jth,kth,start,nnode,nndeg)
         do ith = 1, cpunum
            start = subsize * (ith-1) + lower
            if ( ith .ne. last ) then
               do jth = 1, (subnum+cpunum-ith)/cpunum
```

```
                  mindeg0(ith,jth) = mindeg
                  do kth = start, start + subsize - 1
                     if ( kth .ge. geti ) goto 999
     nnode = perm(kth)
     if ( marker(nnode) .lt. 0 ) goto 401
     nndeg = deg(nnode)
     if ( nndeg .le. thresh ) then
                        call mp_setlock()
                        if ( kth .lt. geti) then
                                geti = kth
lastith = ith
lastjth = jth
node = nnode
                        end if
                        call mp_unsetlock()
                        goto 999
                     end if
     if ( nndeg .lt. mindeg0(ith,jth) ) then
mindeg0(ith,jth) = nndeg
c if (thresh .eq. 52) print *,ith,jth,kth,nndeg
     end if
 401              continue
          end do
                  start = start + blocksize
               end do
            else
               do jth = 1, (subnum+cpunum-ith)/cpunum-1
                  mindeg0(ith,jth) = mindeg
                  do kth = start, start + subsize - 1
                     if ( kth .ge. geti ) goto 999
     nnode = perm(kth)
     if ( marker(nnode) .lt. 0 ) goto 402
     nndeg = deg(nnode)
     if ( nndeg .le. thresh ) then
                        call mp_setlock()
                        if ( kth .lt. geti) then
                                geti = kth
lastith = ith
lastjth = jth
node = nnode
                        end if
                        call mp_unsetlock()
                        goto 999
                     end if
     if ( nndeg .lt. mindeg0(ith,jth) ) then
mindeg0(ith,jth) = nndeg
c if (thresh .eq. 52) print *,ith,jth,kth,nndeg
     end if
 402              continue
  end do
                  start = start + blocksize
               end do
               mindeg0(ith,jth) = mindeg
               do kth = start, upper
                     if ( kth .ge. geti ) goto 999
     nnode = perm(kth)
     if ( marker(nnode) .lt. 0 ) goto 403
     nndeg = deg(nnode)
     if ( nndeg .le. thresh ) then
                        call mp_setlock()
                        if ( kth .lt. geti) then
                                geti = kth
lastith = ith
lastjth = jth
node = nnode
                        end if
                        call mp_unsetlock()
                        goto 999
                     end if
     if ( nndeg .lt. mindeg0(ith,jth) ) then
mindeg0(ith,jth) = nndeg
c if (thresh .eq. 52) print *,ith,jth,kth,nndeg
     end if
 403 continue
end do
             end if
 999  continue
         end do

 j = geti

 mindeg = mindeg0(1,1)
 do jth = 1, lastjth - 1
   do ith = 1, cpunum
     if ( mindeg0(ith,jth) .lt. mindeg ) then
mindeg = mindeg0(ith,jth)
c  if ( thresh .eq. 52 ) print *,'i,j',ith,jth,mindeg
     end if
           end do
         end do

 do ith = 1, lastith
     if ( mindeg0(ith,jth) .lt. mindeg ) then
mindeg = mindeg0(ith,jth)
c  if ( thresh .eq. 52 ) print *,'i,j',ith,jth,mindeg
     end if
         end do

c  print *, '-------------'
c         print *, j, mindeg, thresh,nndeg
c  print *, lower, upper, last, subnum


 if ( geti .le. neqns ) then
```

```
c          print *, j, mindeg, node
  goto 500
 end if

            GO TO 200
C           ------------------------------------------
C           NODE HAS MINIMUM DEGREE. FIND ITS REACHABLE SETS BY
C           CALLING QMDRCH.
C           ------------------------------------------
  500       SEARCH = J
c            print *, j, mindeg, node
c     print *, '^^^^^^^^^'

            NOFSUB = NOFSUB + DEG(NODE)
            MARKER(NODE) = 1
            CALL QMDRCH (NODE, XADJ, ADJNCY, DEG, MARKER,
     1                   RCHSZE, RCHSET, NHDSZE, NBRHD )
C           ----------------------------------------
C           ELIMINATE ALL NODES INDISTINGUISHABLE FROM NODE.
C           THEY ARE GIVEN BY NODE, QLINK(NODE), ....
C           ----------------------------------------
            NXNODE = NODE
  600       NUM = NUM + 1
               NP  = INVP(NXNODE)
               IP  = PERM(NUM)
               PERM(NP) = IP
               INVP(IP) = NP
               PERM(NUM) = NXNODE
               INVP(NXNODE) = NUM
               DEG(NXNODE) = - 1
               NXNODE = QLINK(NXNODE)
            IF (NXNODE .GT. 0) GOTO 600
C
            IF ( RCHSZE .LE. 0 )  GO TO 800
C              ----------------------------------------
C              UPDATE THE DEGREES OF THE NODES IN THE REACHABLE
C              SET AND IDENTIFY INDISTINGUISHABLE NODES.
C              ----------------------------------------
               CALL  QMDUPD ( XADJ, ADJNCY, RCHSZE, RCHSET, DEG,
     1                        QSIZE, QLINK, MARKER, RCHSET(RCHSZE+1),
     1                        NBRHD(NHDSZE+1) )
C              ------------------------------------
C              RESET MARKER VALUE OF NODES IN REACH SET.
C              UPDATE THRESHOLD VALUE FOR CYCLIC SEARCH.
C              ALSO CALL QMDQT TO FORM NEW QUOTIENT GRAPH.
C              ------------------------------------
               MARKER(NODE) = 0
               DO 700 IRCH = 1, RCHSZE
                  INODE = RCHSET(IRCH)
                  IF ( MARKER(INODE) .LT. 0 )  GOTO 700
                     MARKER(INODE) = 0
                     NDEG = DEG(INODE)
```

```
                   IF ( NDEG .LT. MINDEG )  MINDEG = NDEG
                   IF ( NDEG .GT. THRESH )  GOTO 700
                      MINDEG = THRESH
                      THRESH = NDEG
                      SEARCH = INVP(INODE)
700          CONTINUE
             IF ( NHDSZE .GT. 0 )  CALL  QMDQT ( NODE, XADJ,
   1            ADJNCY, MARKER, RCHSZE, RCHSET, NBRHD )
800    IF ( NUM .LT. NEQNS )  GO TO 300
       RETURN
    END
```

BIBLIOGRAPHY

[1] Zahira Ammarguellat and Luddy Harrison. Automatic Recognition of Induction & Recurrence Relations by Abstract Interpretation. *Proceedings of Sigplan 1990, Yorktown Heights*, 25(6):283–295, June 1990.

[2] R. Asenjo, M. Ujaldón, and E. L. Zapata. *SpLU – Sparse LU Factorization. HPF-2. Scope of Activities and Motivating Applications.* High Performance Fortran Forum, version 0.8 edition, November 1994.

[3] Rafael Asenjo, Eladio Gutierrez, Yuan Lin, David Padua, Bill Pottenger, and Emilio Zapata. On the Automatic Parallelization of Sparse and Irregular Fortran Codes. Technical Report 1512, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., December 1996.

[4] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, Philadelphia, PA, 1994.

[5] M. Berry, D. Chen, P. Koss, D. Kuck, L. Pointer, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *Int'l. Journal of Supercomputer Applications, Fall 1989*, 3(3):5–40, Fall 1989.

[6] G.A. Bird. *Molecular Gas Dynamics and the Direct Simulation of Gas Flows.* Oxford University Press, Oxford, England, 1994.

[7] Graeme Bird. Personal communication with author, 1996.

[8] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel Programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[9] B.R. Brooks, R.E. Bruccoleri, B.D. Olafson, D.J. States, S. Swaminathan, and M. Karplus. CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations. *J. Comp. Chem.*, 4:187–217, 1983.

[10] H. Chen and K. J. Lynch. Automatic Construction of Networks of Concepts Characterizing Document Databases. *IEEE Transactions on Systems, Man and Cybernetics*, 22(5):885–902, September/October 1992.

[11] Hsinchun Chen, Bruce Schatz, Tobun Ng, Joanne Martinez, Amy Kirchhoff, and Chienting Lin. A Parallel Computing Approach to Creating Engineering Concept Spaces for Semantic Retrieval: The Illinois Digital Library Initiative Project. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1996.

[12] S. C. Chen, D. J. Kuck, and A. H. Sameh. Practical Parallel Band Triangular System Solvers. *ACM Trans. on Mathematical Software*, 4(3):270–277, Sept., 1978.

[13] Ronald Gary Cytron. *Compile-Time Scheduling and Optimization for Asynchronous Machines.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Sci., Oct., 1984.

[14] Luiz DeRose, Kyle Gallivan, Bret Marsolf, David Padua, and Stratis Gallopoulos. FALCON: A MATLAB Interactive Restructuring Compiler. *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing, Columbus, OH*, pages 18.1–18.18, August 1995.

[15] Iain Duff, Nick Gould, John Reid, Jennifer Scott, and Linda Miles. Harwell Subroutine Library. Technical Report http://www.rl.ac.uk/departments/ccd/numerical/hsl/hsl.html, Council for the Central Laboratory of the Research Councils, Department for Computation and Information, Advanced Research Computing Division.

[16] Rudolf Eigenmann and Siamak Hassanzadeh. Evaluating High-Performance Computer Technology through Industrially Significant Applications. *IEEE Computational Science & Engineering*, Spring 1996.

[17] A. Fisher and A. Ghuloum. Parallelizing Complex Scans and Reductions. *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, June 1994.

[18] Ian Foster, Rob Schreiber, and Paul Havlak. HPF-2 Scope of Activities and Motivating Applications. Technical Report CRPC-TR94492, Rice University, November 1994.

[19] William B. Frakes and Ricardo Baeza-Yates. *Information Retrieval Data Structures & Algorithms*. Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[20] Dr. Sam Fuller. Seminar Presented at UIUC, October 1996.

[21] D. D. Gajski, D. J. Kuck, and D. A. Padua. Dependence Driven Computation. *Proceedings of the COMPCON 81 Spring Computer Conf.*, pages 168–172, Feb., 1981.

[22] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond Induction Variables: Detecting and Classifying Sequences Using a Demand-driven SSA Form. *To appear in TOPLAS*.

[23] Milind Baburao Girkar. *Functional Parallelism Theoretical Foundations and Implementation*. PhD thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., December 1991.

[24] G.H. Golub and C.F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1993.

[25] Mark D. Guzzi, David A. Padua, Jay P. Hoeflinger, and Duncan H. Lawrie. Cedar Fortran and Other Vector and Parallel Fortran Dialects. *Journal of Supercomputing*, 4(1):37–62, March 1990.

[26] Mohammad R. Haghighat and Constantine D. Polychronopoulos. Symbolic Program Analysis and Optimization for Parallelizing Compilers. *Presented at the 5th Annual Workshop on Languages and Compilers for Parallel Computing, New Haven, CT*, August 3-5, 1992.

[27] Luddy Harrison. Compiling Lisp for Evaluation on a Tightly Coupled Multiprocessor. Technical Report 565, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., Mar. 20, 1986.

[28] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 1979.

[29] IBM. Parallel FORTRAN Language and Library Reference, March 1988.

[30] P. Jouvelot and B. Dehbonei. A Unified Semantic Approach for the Vectorization and Parallelization of Generalized Reductions. In *Proceedings of the 1989 International Conference on Supercomputing*, Crete, Greece, June 5-9, 1989. ACM.

[31] Jee Myeong Ku. The Design of an Efficient and Portable Interface Between a Parallelizing Compiler and its Target Machine. Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., 1995.

[32] D. Kuck, P. Budnik, S-C. Chen, Jr. E. Davis, J. Han, P. Kraska, D. Lawrie, Y. Muraoka, R. Strebendt, and R. Towle. Measurements of Parallelism in Ordinary FORTRAN Programs. *Computer*, 7(1):37–46, Jan., 1974.

[33] D. J. Kuck. *The Structure of Computers and Computations,*, volume I. John Wiley & Sons, Inc., NY, 1978.

[34] David Kuck and Yoichi Muraoka. Bounds on the Parallel Evaluation of Arithmetic Expressions Using Associativity and Commutativity. *Acta Informatica*, 3, Fasc. 3:203–216, 1974.

[35] David J. Kuck and Richard A. Stokes. The Burroughs Scientific Processor (BSP). *Special Issue on Supersystems, IEEE Trans. on Computers*, C-31(5):363–376, May, 1982.

[36] D.H. Lehmer. Mathematical Methods in Large-scale Computing Units. In *2nd Symposium on Large-Scale Digital Calculating Machinery*, pages 141–146, Cambridge, MA, 1951. Harvard University Press.

[37] P.A.W. Lewis, A.S. Goodman, and J.M. Miller. A Pseudo-Random Number Generator for the System/360. *IBM Systems Journal*, 8(2):136–146, May 1969.

[38] G. Lueker. Some Techniques for Solving Recurrences. *Computing Surveys, Vol. 12, No. 4*, December 1980.

[39] Michael Mascagni and David Bailey. Requirements for a Parallel Pseudorandom Number Generator. Technical Report http://olympic.jpl.nasa.gov/SSTWG/lolevel.msgs.html, Center for Computing Sciences, I.D.A. and NASA Ames Research Center, June 1995.

[40] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems, Vol 9, No 1*, pages 21–65, February 1991.

[41] Jose Eduardo Moreira. *On the Implementation and Effectiveness of Autoscheduling for Shared-Memory Multiprocessors.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., February 1995.

[42] D. Padua and M. Wolfe. Advanced Compiler Optimization for Supercomputers. *CACM*, 29(12):1184–1201, December, 1986.

[43] D. A. Padua, D. J. Kuck, and D. H. Lawrie. High-Speed Multiprocessors and Compilation Techniques. *Special Issue on Parallel Processing, IEEE Trans. on Computers*, C-29(9):763–776, Sept., 1980.

[44] Bill Pottenger and Rudolf Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. *Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona, Spain*, pages 444–448, July 1995.

[45] Bill Pottenger and Bruce Schatz. **cSpace**: A Parallel C^{++} Information Retrieval Benchmark. Technical Report 1511, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., January 1997.

[46] Bill Pottenger and Bruce Schatz. On the Evaluation of C^{++} in a Parallel Programming Environment. Technical Report 1506, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., November 1996.

[47] William Morton Pottenger. Induction Variable Substitution and Reduction Recognition in the Polaris Parallelizing Compiler. Master's thesis, Univ of Illinois at Urbana-Champaign, Cntr for Supercomputing Res & Dev, December 1994.

[48] Daniel V. Pryor, Steven A. Cuccaro, Michael Mascagni, and M. L. Robinson. Implementation of a Portable and Reproducible Parallel Pseudorandom Number Generator. *Proceedings of Supercomputing '94*, Nov. 1994.

[49] Lawrence Rauchwerger. *Run-Time Parallelization: A Framework for Parallel Computation.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., August 1995.

[50] Lawrence Rauchwerger and David Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *Proceedings of the SIGPLAN'95 Conference on Programming Language Design and Implementation*, June 1995.

[51] Martin C. Rinard and Pedro C. Diniz. Commutativity Analysis: A New Analysis Framework for Parallelizing Compilers. In *Programming Language Implementation and Design (PLDI)*, pages 54–67. ACM, 1996.

[52] L.F. Romero and E.L. Zapata. Data Distributions for Sparse Matrix Vector Multiplication. *J. Parallel Computing*, 21(4):583–605, April 1995.

[53] Dmitri Roussinov. Personal communication with author, 1996.

[54] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[55] B. Schatz, E. Johnson, P. Cochrane, and H. Chen. Interactive Term Suggestion for Users of Digital Libraries: Using Subject Thesauri and Co-occurrence Lists for Information Retrieval. In *1st International ACM Conference on Digital Libraries*, pages 126–133, Bethesda, MD, 1996. ACM.

[56] Bruce Schatz. *Interactive Retrieval in Information Spaces Distributed across a Wide-Area Network*. PhD thesis, University of Arizona Computer Science Department, December 1990.

[57] Bruce Schatz and Hsinchun Chen. Building Large-Scale Digital Libraries. *IEEE Computer*, May 1996.

[58] NRC Computer Science and Telecommunications Board. *Computing The Future*. National Academy Press, Washington, D.C., 1992.

[59] T. Suganuma, H. Komatsu, and T. Nakatani. Detection and Global Optimization of Reduction Operations. *Proceedings of ICS'96, Philadelphia, PA, USA*, July 1996.

[60] Peiyi Tang, Pen-Chung Yew, and Chuan-Qi Zhu. Compiler Techniques for Data Synchronization in Nested Parallel Loops. *Proceedings of ICS'90, Amsterdam, Holland*, 1:177–186, May 1990.

[61] Peng Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis*. PhD thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., May 1995.

[62] Peng Tu and David Padua. Automatic Array Privatization. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, OR. Lecture Notes in Computer Science.*, volume 768, pages 500–521, August 12-14, 1993.

[63] Peng Tu and David Padua. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. *Proceedings of the 9th ACM International Conference on Supercomputing, Barcelona, Spain*, pages 414–423, July 1995.

[64] Alexander Veidenbaum. *Compiler Optimizations and Architecture Design Issues for Multiprocessors*. PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Comput. Sci., May 1985.

[65] Guhan Viswanathan and James R. Larus. User-defined Reductions for Efficient Communication in Data-Parallel Languages. Technical Report 1293, Univ. of Wisconsin-Madison, Computer Sciences Department, Aug. 1996.

[66] Dick Wilmoth. Personal communication with author, 1996.

[67] Dr. Richard Wirt. Seminar Presented at UIUC, September 1996.

[68] Hans Zima with Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, NY, 1990.

VITA

William Morton Pottenger
www.ncsa.uiuc.edu/People/billp

William Morton (Bill) Pottenger was born in February of 1957 in Chicago Illinois. Bill completed a Bachelor of Arts in Religion Studies at Lehigh University in 1980. After working in the computer field for several years, Bill returned to the University of Alaska to obtain a Secondary Teaching Certificate as well as a Bachelor of Science in Computer Science. While at the University of Alaska Bill was inducted into the Phi Kappa Honor Society, appeared on the Chancellor's List multiple times, and in 1989 was distinguished Outstanding Student of the Year in Computer Science.

Following matriculation from the University of Alaska, Bill continued his education at the University of Illinois at Urbana-Champaign. In 1995 Bill completed a Master of Science in Computer Science, and in May of 1997 completed a Doctor of Philosophy in Computer Science.

Bill's research interests lie in the fields of semantic information retrieval, high-performance benchmarking, and automatic parallelization of computer programs. Publications and technical reports which Bill has jointly authored include the following:

1. Polaris: Improving the Effectiveness of Parallelizing Compilers, *Lecture Notes in Computer Science 892*, Springer-Verlag, pages 141–154, August 1994; also in *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, pages 10.1 – 10.18, August 1994 *[joint author]*
2. Idiom Recognition in the Polaris Parallelizing Compiler, *Proceedings of the 9th ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995 *[primary co-author]*
3. Effective Automatic Parallelization with Polaris, *International Journal of Parallel Programming*, May 1995 *[joint author]*

4. Targeting a Shared Address Space version of the Seismic Benchmark Seis1.1, *www.specbench.org /hpg/sas.ps*, January 1996 *[primary co-author]*
5. Parallel Programming with Polaris, *IEEE Computer*, December 1996 *[joint author]*
6. Real-time Semantic Retrieval on Parallel Computers, Center for Supercomputing Research and Development, Technical Report 1516, January 1997 *[primary co-author]*
7. **Seismic**: A Hybrid C/Fortran Seismic Processing Benchmark, Center for Supercomputing Research and Development, Technical Report 1510, January 1997 *[primary co-author]*
8. **cSpace**: A Parallel C^{++} Information Retrieval Benchmark, Center for Supercomputing Research and Development, Technical Report 1511, January 1997 *[primary co-author]*
9. On the Automatic Parallelization of Sparse and Irregular Fortran Codes, Center for Supercomputing Research and Development, Technical Report 1512, January 1997 *[primary joint author]*

Bill currently holds a position as an Adjunct Assistant Professor in the Computer Science Department at the University of Illinois in Urbana-Champaign. He is also a Research Scientist at the National Center for Supercomputing Applications (NCSA), as well as holding a research appointment in the Digital Library Research Program in the University Library at Illinois.