The Role of Associativity and Commutativity in the Detection and Transformation of Loop-Level Parallelism*

William M. Pottenger
Department of Computer Science
University of Illinois at Urbana-Champaign
billp@uiuc.edu

Abstract

The study of theoretical and practical issues in automatic parallelization across application and language boundaries is an appropriate and timely task. In this paper, we discuss theory and techniques that we have determined useful in parallelizing recurrences and reductions in computer programs. We present a framework for understanding such parallelism based on an approach which models loop bodies as *coalescing loop operators*. Within this framework we distinguish between associative coalescing loop operators and associative and commutative coalescing loop operators. We present the result of the application of this theory in a case study of a modern C^{++} semantic retrieval application drawn from the digital library field.

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 certain operations.

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 [BDE$^+$96].

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 (naively) 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

*This work is supported in part by Army Contract DABT63-95-C-0097; Army Contract N66001-97-C-8532; NSF Contract MIP-9619351; and a Partnership Award from IBM. This work is not necessarily representative of the positions or policies of the Army or Government.

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

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 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 and reductions in Fortran, for example, has been a topic of study for several years.

2.1 Associative Operations

Associative operations have been the basis for parallelization of reduction operations in both hardware and software systems for many years [Kuc78, JD89, FG94, Pot94, PE95].

In most of these cases, the associativity is limited to a single binary operation involving the operator + (addition) or $*$ (multiplication). For example, in [PE95], 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 [JD89]. Similar to the techniques implemented in [FG94], 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 [Har86].

2.2 Commutative Operations

In [RD96], 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* [Kuc78].

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.

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 $\mathcal{L}$ 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 into 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.

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.

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*.
□

We will now move on to consider coalescing loop operators which are associative in nature.

3.2 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.

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

3.3 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: please refer to [Pot97] for additional detail.

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[Tu95]. 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 involving the non-commutative associative operator $\oplus$:

```
do i = 1, n
   var = var ⊕ 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 [IBM88] 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 var_p
   dofirst
      var_p = < id for ⊕ >
   doevery
      var_p = var_p ⊕ a(i)
   enddo
   dofinal, ordered lock
      var = var ⊕ var_p
enddo
```

The first two steps in the transformation are the privatization and initialization of the shared variable var. In the above code, var_p is the processor-private copy of var. If p processors participate in the computation, then p private copies of var 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, var_p is initialized on each processor. The initial value $< id\ for\ \oplus >$ is the identity for the operator $\oplus$. `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:

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

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 [GPHL90]. 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 var. 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 var 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 $\mathcal{L}^t$.
□

4 Abstracting Coalescing Operators from Loops

In this section 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, the framework presented herein provides necessary theoretical foundations for performing the analysis needed to prove these loops parallel.

4.1 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. 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 [BDE+96].

4.2 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. 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 $i = 1, n$
  ...
  if $(condition) temp = \ldots$
  ...
  if $(condition) \ldots = temp$
  ...
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 [TYZ90]. 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 3.3.

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[2]. Yet our example loop containing a dynamic last value can be readily parallelized based on associativity alone:

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

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 [Pot97].

4.3 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.

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

It is well known that recurrences are parallelizable when based on associative binary operators [Har86]. 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[3]:

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

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:

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

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, we can take advantage of the indexable nature of the array a to treat each element as a scalar. This transformation is discussed in detail in [Pot98].

The ability to optimize the parallelization of the operator *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.

4.4 Loops Performing Gather Operations

The following depicts a generic gather operation:

```
for (int i = 0, j = 0; i < n; ++i)
  if (is_true(a[i]))
    a[j++] = i;
```

Conceptually, this pattern is quite common in many sparse and symbolic codes such as those discussed in [Pot97].

The difficulty in parallelizing such a loop occurs because the initial value of j for a given iteration is dependent on how often a[i] was true in preceding iterations. However, in [Pot98] we show that gather is an associative coalescing operation, and thus this loop can be transformed into parallel form.

5 Commutativity versus 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 [Kuc78]. 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 following loop:

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

The coalescing loop operator α is += with the semantics "add the assimilated argument on the right to the conglomerate argument on the left and return the resulting sum". In this case, α is both an associative and commutative operator with identity 0. This loop can thus be transformed into $\mathcal{L}^t$ and the resulting `doevery` doall loop executed on four processors as follows:

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

[3] In [Pot97], we discuss the closed-form solution of such recurrences when β is loop invariant.

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 summarize by distinguishing between two types of coalescing loop operators: 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.

6 Application of the Model

In this section we present the results of the application of the theory of associative coalescing loop operators in the parallelization of a modern C^{++} semantic information retrieval application drawn from the emerging Digital Library field [PS97]. In [Pot97] we perform a detailed analysis of this application, **cSpace**, and determine that the most time-consuming outermost loop can be modeled as an associative coalescing loop operator. The following depicts this loop transformed into parallel form based on $\mathcal{L}^t$:

```
parallel loop, block: For each term_a in Terms
  private ostream os_p, co-occurrences coocs_p
  doevery
    For each doc in term_a.docs
      For each term_b in doc.terms_in_doc
        ...
    For each cooc in coocs_p
      Compute cooc.similarity(term_a, term_b)
    Perform subset operation on coocs_p
    os_p << term_a.similarities
  enddo
  dofinal, ordered lock
    append(os,os_p)
enddo
```

Table 1 summarizes the performance and scalability of **cSpace** across three data sets. For each data set, the serial version of **cSpace** was executed on one processor, and the parallellized 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 (h:m:s) and in minutes and seconds (m:s). The S_p columns report speedup for the given execution.

	$no\ pr$	$PR\ m:s$	$PR\ S_p$	$CR\ h:m:s$	$CR\ S_p$	$DR\ h:m:s$	$DR\ S_p$
Serial	1	4:07	1	1:17:19	1	10:30:27	1
Parallel	2	1:50	2.25	0:27:16	2.84	3:32:49	2.96
	4	1:02	3.98	0:14:28	5.34	1:53:52	5.54
	8	0:40	6.18	0:08:41	8.90	1:08:20	9.23
	16	0:35	7.06	0:05:44	13.49	0:39:22	16.01

Table 1: Execution Times and Speedups for **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 paper 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.

Several interesting trends are revealed in Table 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 [PS97]. 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.

In summary, as can be determined from these results, **cSpace** achieved 100% efficiency for all runs made using the DR data set.

7 Conclusions

We draw the following conclusions:

- A wide variety of loops perform non-commutative, associative operations
- The theory of *coalescing loop operators* provides a general framework for understanding parallelism in loops of this nature

- Transformation $\mathcal{L}^t$ can be used as a general approach in implementing the parallelism in associative coalescing loop operators

In the introduction 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.

There are many issues to address in the implementation of an algorithm capable of detecting associative coalescing loop operations. An algorithm is sketched at a high level in [Pot98]. As such, this presents a challenging problem in the field of parallelizing compilers.

8 Acknowledgements

I would like to acknowledge with a grateful heart the assistance of my Lord and Saviour Jesus Christ in completing this work.

References

[BDE+96] 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.

[FG94] A. Fisher and A. Ghuloum. Parallelizing Complex Scans and Reductions. *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, June 1994.

[GPHL90] 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.

[Har86] 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.

[IBM88] IBM. Parallel FORTRAN Language and Library Reference, March 1988.

[JD89] 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.

[Kuc78] D. J. Kuck. *The Structure of Computers and Computations*, volume I. John Wiley & Sons, Inc., NY, 1978.

[PE95] 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.

[Pot94] William Morton Pottenger. Induction Variable Substitution and Reduction Recognition in the Polaris Parallelizing Compiler. Master's thesis, Univ of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., December 1994.

[Pot97] William Morton Pottenger. *Theory, Techniques, and Experiments in Solving Recurrences in Computer Programs.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., May 1997.

[Pot98] William M. Pottenger. The Role of Associativity and Commutativity in the Detection and Transformation of Loop-Level Parallelism. Technical Report 1532, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., April 1998.

[PS97] 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.

[RD96] 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.

[Tu95] 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.

[TYZ90] 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.