Commutativity Analysis: A Technique for Automatically Parallelizing Pointer-Based Computations*

Martin Rinard (martin@cs.ucsb.edu) Pedro Diniz (pedro@cs.ucsb.edu)

Department of Computer Science, University of California at Santa Barbara
Santa Barbara, CA 93106-5110

Abstract

This paper introduces an analysis technique, commutativity analysis, for automatically parallelizing computations that manipulate dynamic, pointer-based data structures. Commutativity analysis views computations as composed of operations on objects. It then analyzes the program to discover when operations commute, i.e. leave the objects in the same state regardless of the order in which they execute. If all of the operations required to perform a given computation commute, the compiler can automatically generate parallel code. Commutativity analysis eliminates many of the limitations that have prevented existing compilers, which use data dependence analysis, from successfully parallelizing pointer-based applications. It enables compilers to parallelize computations that manipulate graphs and eliminates the need to analyze the data structure construction code to extract global properties of the data structure topology. This paper shows how to use symbolic execution and expression manipulation to statically determine that operations commute and how to exploit the extracted commutativity information to generate parallel code. It also presents performance results that demonstrate that commutativity analysis can be used to successfully parallelize the Barnes-Hut hierarchical N-body solver, an important scientific application that manipulates a complex pointer-based data structure.

1 Introduction

Current parallelizing compilers preserve the semantics of the original serial program by preserving the data dependences [1]. These compilers attempt to identify independent pieces of computation (two pieces of computation are independent if neither writes a piece of data that the other accesses), then generate code that executes independent pieces concurrently.

A significant limitation of applying data dependence analysis to computations that manipulate dynamic, pointer-based computations is the difficulty of performing analysis that is precise enough to expose the concurrency. To statically discover independent pieces of code, the compiler must recognize global topological properties of the manipulated data structures [7]. It must therefore analyze the code that builds the data structure and propagate the results of this analysis through the program to the sections that use the data. The difficulty of extracting and accurately maintaining this information often prevents the compiler from recognizing that independent pieces are, in fact, independent. An even more fundamental limitation is an inherent inability to parallelize computations that manipulate graphs. The aliases present in these data structures preclude the static discovery of independent pieces of code, forcing the compiler to generate serial code.

Experience with parallel applications provides additional evidence for the fundamental inadequacy of data dependence analysis. Programmers that parallelize serial applications by hand do not simply write code that executes independent pieces of code concurrently. For example, four (Water, MP3D, LocusRoute and Cholesky) of the six parallel applications in the SPLASH benchmark suite [13] and three of the four parallel applications described in [12] violate the data dependences of the original serial program. They instead rely on commuting operations (operations that generate the same result regardless of their execution order) to preserve the semantics of the original serial program. This experience strongly suggests that compilers will never be able to parallelize a substantial range of applications unless they recognize and exploit commuting operations.

This paper presents an analysis technique, commutativity analysis, that automatically recognizes and exploits commuting operations to generate parallel code. Commutativity analysis is designed for programs that use an object-based

*This research was supported in part by an Alfred P. Sloan Research Fellowship. The second author is sponsored by JNICT – Junta Nacional de Investigação Científica e Tecnológica and by the Fulbright Program.

programming paradigm. This paradigm aggregates individual words of memory into coarser grain units called objects and individual statements of the program into coarser grain units called operations. Commutativity analysis analyzes the program at this granularity to discover when operations on objects commute (i.e. generate the same result regardless of the order in which they execute). If all of the operations required to perform a given computation commute, the compiler can automatically generate parallel code. Commutativity analysis provides the following advantages over traditional approaches that use data dependence analysis:

- **Topology Independence:** The presence of aliases in the manipulated data structures does not impair the ability of commutativity analysis to parallelize the computation. Commutativity analysis therefore enables compilers to parallelize computations that manipulate graphs. It also eliminates the need to perform complex analysis to discover global properties of the data structure topology.

- **Focused Analysis:** Compilers that use commutativity analysis only need to analyze the code that they will parallelize. This property eliminates the need to extract information about the global data structure topology and propagate this information across large sections of the program. It may also enable the automatic parallelization of computations in areas of computer science that have traditionally been unable to benefit from parallelizing compilers. For example, commutativity analysis may be appropriate for parallelizing computations that manipulate the persistent data in object-oriented databases. In this context the code that originally created the data structure may be unavailable, negating any approach (including data-dependence based approaches) whose success depends on information about the data structure topology.

We believe that commutativity analysis will extend the range of applications that compilers can automatically parallelize. It is especially appropriate for computations that manipulate dynamic, pointer-based data structures because it simplifies the analysis required to parallelize such applications and extends the range of parallelizable applications to include graph computations. The difficulty of parallelizing dynamic, pointer-based applications by hand also makes commutativity analysis especially valuable in this context.

This paper makes the following contributions:

- It presents an analysis technique that uses symbolic execution, expression manipulation and expression comparison to statically recognize commuting operations.

- It shows how to exploit static knowledge of commuting operations to generate parallel code.

- It presents a case study of commutativity analysis applied to a complex application, the Barnes-Hut hierarchical N-body solver. This case study shows that commutativity analysis can successfully parallelize a computation that manipulates complex pointer-based data structures, and that the resulting parallel performance is comparable to the performance of a highly optimized version parallelized by hand.

The rest of the paper is structured as follows. In Section 2 we present an example that illustrates how commuting operations enable the concurrent execution of graph traversals. In Section 3 we describe the basic approach and state the conditions that a compiler can use to recognize commuting operations. In Section 4 we present an analysis algorithm that a compiler can use to determine if two operations commute. In Section 5 we present experimental results for a version of the Barnes-Hut hierarchical N-body solver parallelized using commutativity analysis. We survey related work in Section 6 and conclude in Section 7.

2 An Example

In this section we present a simple example that shows how recognizing commuting operations can enable the automatic generation of parallel code. The `visit` method in Figure 1 serially traverses a graph. When the traversal completes, each node's `sum` instance variable contains the sum of its original value and the values of the `val` instance variables in all of the nodes that directly point to that node. The example is written in C++.

```
class graph {
  boolean mrk;
  int val, sum;
  graph *left; graph *right;
};
graph::visit(int p) {
  sum = sum + p;
  if (!mrk) {
    mrk = TRUE;
    if (left != NULL) left->visit(val);
    if (right != NULL) right->visit(val);
  }
}
```

Figure 1: Serial Graph Traversal.

The traversal generates one invocation of the `visit` method for each edge in the graph. We call each method

```
class graph {
  mutex lock;
  boolean mrk;
  int val, sum;
  graph *left; graph *right;
};
graph::visit(int s){
  this->parallel_visit(s);
  wait();
}
graph::parallel_visit(int p){
  lock.acquire();
  sum = sum + p;
  if (!mrk) {
    mrk = TRUE;
    lock.release();
    if(left != NULL)
      spawn(left->parallel_visit(val));
    if(right != NULL)
      spawn(right->parallel_visit(val));
  } else {
    lock.release();
  }
}
```

Figure 2: Parallel Graph Traversal.

invocation an *operation*. The receiver of each `visit` operation is the node to traverse. Each `visit` operation takes as a parameter `p` the value of the instance variable `val` of the node that points to the receiver. The `visit` operation first adds `p` into the running sum stored in the receiver's `sum` instance variable. It then checks the receiver's `mrk` instance variable to see if the traversal has already visited the receiver. If not, the operation marks the receiver, then recursively invokes the `visit` method for all of the nodes that the receiver points to.

The way to parallelize the traversal is to execute the two recursive `visit` operations concurrently. But this parallelization may violate the data dependences. The serial computation executes all of the accesses generated by the left traversal before all of the accesses generated by the right traversal. If the two traversals visit the same node, in the parallel execution the right traversal may visit the node before the left traversal, changing the order of reads and writes to that node. This violation of the data dependences may generate cascading changes in the overall execution of the computation. Because of the marking algorithm, a node only executes the recursive calls the first time it is visited. If the right traversal reaches a node before the left traversal, the parallel execution may also change the order in which the overall traversal is generated.

In fact, none of these changes affects the overall result of the computation. It is possible to automatically parallelize the computation even though the resulting parallel program may generate computations that differ substantially from the original serial computation. The key property that enables the parallelization is that the parallel computation generates the same set of `visit` operations as the serial computation and the generated `visit` operations can execute in any order without affecting the overall behavior of the traversal.

Given this commutativity information, the compiler can automatically generate the parallel `visit` method in Figure 2. The top level `visit` method first invokes the `parallel_visit` method, then invokes the **wait** construct, which blocks until all parallel tasks created by the current task or its descendant tasks finishes. The `parallel_visit` method executes the recursive calls concurrently using the **spawn** construct, which creates a new task for the execution of each invocation. The compiler also augments the `graph` data structure with a mutual exclusion lock. This lock ensures that each invocation of `parallel_visit` executes atomically with respect to all other invocations with the same receiver. The `parallel_visit` method acquires the lock before accessing the receiver and releases the lock before invoking any methods. Application of lazy task creation techniques [9] can increase the granularity of the resulting parallel computation.

3 The Basic Approach

Commutativity analysis is designed for programs written using a pure object-based paradigm. Such programs structure the computation as a sequence of operations on objects. Each operation consists of a receiver object, an operation name and several parameters. Each operation name identifies a method that defines the behavior of the operation; when the operation executes it invokes that method. Each object implements its state using a set of instance variables. When a method executes it can recursively invoke other operations and/or use primitive operators (such as addition and multiplication) to perform computations involving the parameters and the instance variables of the receiver.

Commutativity analysis is designed to work with *separable* methods. A method is separable if its execution can be decomposed into an object section and an invocation section. The object section performs all accesses to the receiver. The invocation section invokes other operations and does not access the receiver. It is of course possible for local variables to carry values computed in the object section into the invocation section, and both sections can access the parameters. Separability imposes no expressibility limitations — it is possible to convert any method into a collection of separable methods via the introduction of auxiliary meth-

ods. In [11] we present a formal treatment of commutativity analysis as applied to separable operations.

The foundation of commutativity analysis is a set of conditions that a compiler can use to test if two operations A and B commute. The commutativity testing conditions must consider two execution orders: the execution order A;B in which A executes first then B executes, and the execution order B;A in which B executes first then A executes. Two operations commute if they meet the following conditions:

- **Instance Variables:** The new value of each instance variable of the receiver objects of A and B under the execution order A;B must be the same as the new value under the execution order B;A.

- **Invoked Operations:** The multiset of operations directly invoked by either A or B under the execution order A;B must be the same as the multiset of operations directly invoked by either A or B under the execution order B;A.

Both commutativity testing conditions are trivially satisfied if the two operations have different receivers — the executions of the two methods are independent because they access disjoint pieces of data. We therefore focus on the case when the two operations have the same receiver.

It is possible to determine if each of the receiver's instance variables has the same new value in both execution orders by analyzing the invoked methods to extract two symbolic expressions. One of the symbolic expressions denotes the new value of the instance variable under the execution order A;B. The other denotes the new value under the execution order B;A. Given these two expressions, a compiler may be able to use algebraic reasoning to discover that they denote the same value. The compiler uses a similar approach to determine if A and B together invoke the same multiset of operations in both execution orders.

We illustrate these concepts by applying them to the graph traversal example in Figure 1. We assume two invocations `r->visit(p1)` and `r->visit(p2)` of the `visit` method. `r->visit(p1)` has parameter `p1`, `r->visit(p2)` has parameter `p2` and both operations have the same receiver `r`.

We first consider the instance variable `sum`. Table 1 contains the two expressions denoting the new values of `sum` under the two execution orders. In these expressions `sum` represents the old value of the `sum` instance variable before either method executes. Given these two expressions, the compiler can use the fact that `+` is both commutative and associative to discover that the two expressions denote the same value.[1]

[1]We ignore here potential anomalies caused by the finite representation of numbers. A compiler switch that disables the exploitation of commutativity and associativity for operators such as `+` will allow the programmer to prevent the compiler from performing transformations that may change the order in which the parallel program combines the summands.

We next consider the new value of the `mrk` instance variable. Because its new value depends on the flow of control through the method, the expressions representing its new values contain conditionals. A conditional of the form $\texttt{if}(\texttt{cx}, \texttt{ex}_1, \texttt{ex}_2)$ denotes the expression $\texttt{ex}_1$ if `cx` is true and $\texttt{ex}_2$ if `cx` is false. Table 1 contains the new values of the `mrk` instance variable, which are identical under both execution orders.

Finally, because the `visit` method writes none of the other instance variables, their new values are the same as their values before the execution of the two operations. The two operations `r->visit(p1)` and `r->visit(p2)` therefore satisfy the first commutativity testing condition.

We next consider the multiset of invoked operations. Because an operation may be invoked along one control flow path but not another, the symbolic operation expressions used to denote invoked operations may contain conditionals. A conditional symbolic operation expression of the form `if(cx,o)` denotes an operation `o` that is invoked only if `cx` is true. Table 2 contains the symbolic operation expressions that denote the operations that `r->visit(p1)` and `r->visit(p2)` directly invoke under the two execution orders. The compiler checks that the two operations meet the second commutativity testing condition by comparing the multiset of the symbolic operation expressions under the execution order `r->visit(p1);r->visit(p2)` with the multiset of the symbolic operation expressions under the execution order `r->visit(p2);r->visit(p1)`. In this example `r->visit(p1)` and `r->visit(p2)` together invoke the same multiset of operations in both execution orders, and the two operations satisfy the second commutativity testing condition.

A compiler can use the commutativity testing conditions described above to determine if it can legally generate parallel code for a given method. The compiler first computes a conservative approximation to the set of methods invoked as a result of invoking the given method. It then applies the commutativity testing conditions described above to all pairs of potentially invoked methods that may have the same receiver. If all of the pairs commute, the compiler can legally generate parallel code.

4 Analysis

Programmers define operations by writing methods. Each operation corresponds to a method invocation: to execute an operation, the machine executes the code in the corresponding method. The commutativity analysis algorithm determines if operations commute by analyzing the corresponding methods.

Execution Order	New Value of sum	New Value of mrk
r->visit(p1); r->visit(p2)	(sum + p1) + p2	if(!if(!mrk,TRUE,mrk),TRUE,if(!mrk,TRUE,mrk))
r->visit(p2); r->visit(p1)	(sum + p2) + p1	if(!if(!mrk,TRUE,mrk),TRUE,if(!mrk,TRUE,mrk))

Table 1: New Values of sum and mrk Under Different Execution Orders

Execution Order	Symbolic Operations Invoked By r->visit(p1) and r->visit(p2)

r->visit(p1); r->visit(p2)	if(!mrk,if(left!=NULL,left->visit(val))),
	if(!mrk,if(right!=NULL,right->visit(val))),
	if(!if(!mrk,TRUE,mrk),if(left!=NULL,left->visit(val))),
	if(!if(!mrk,TRUE,mrk),if(right!=NULL,right->visit(val)))

r->visit(p2); r->visit(p1)	if(!mrk,if(left!=NULL,left->visit(val))),
	if(!mrk,if(right!=NULL,right->visit(val))),
	if(!if(!mrk,TRUE,mrk),if(left!=NULL,left->visit(val))),
	if(!if(!mrk,TRUE,mrk),if(right!=NULL,right->visit(val)))

Table 2: Symbolic Operations Invoked by r->visit(p1) and r->visit(p2) Under Different Execution Orders

4.1 Overview

Given a piece of code to parallelize, the compiler first traverses the static call graph to compute a conservative approximation to the set of methods invoked as a result of executing a given method. To apply the commutativity testing conditions the compiler must represent and reason about both the new values of the receiver's instance variables and the multiset of methods invoked when two operations execute. The compiler represents these new values and invoked methods using symbolic expressions. To check if two method invocations commute, the compiler first uses symbolic execution to extract the relevant expressions under the two execution orders. It then applies the commutativity testing conditions by simplifying the expressions and comparing corresponding expressions for equality. If all pairs of invoked methods commute the compiler can generate parallel code.

Figure 3 contains the commutativity analysis algorithm, which determines if it is possible to generate parallel code for a method. The algorithm performs $O(n^2)$ commutativity testing operations where n is the number of methods potentially invoked by the method either directly or indirectly. Figure 3 also contains the commutativity testing algorithm. This algorithm performs v instance variable expression comparisons, where v is the number of instance variables in the class of the receiver object. The algorithm also compares $\mathtt{s}_1$ and $\mathtt{s}_2$, which are two multisets of expressions denoting potentially invoked methods. $\mathtt{s}_1$ and $\mathtt{s}_2$ both contain $m_1 + m_2$ expressions denoting potentially invoked methods, where m_1 and m_2 are the number of methods directly invoked by $\mathtt{op}_1$ and $\mathtt{op}_2$, respectively. The comparison of $\mathtt{s}_1$ with $\mathtt{s}_2$ performs $m_1 + m_2$ comparisons of these expressions.

```
can_parallelize(op)
 // invoked_by(op) traverses the static call graph rooted at op
 // to compute a conservative approximation to the set of methods
 // directly or indirectly invoked as a result of executing op.
 forall ⟨op1, op2⟩ ∈ invoked_by(op)×invoked_by(op)
  // recv(op) is the class of the receiver of op
  if recv(op1)=recv(op2)
   // commute(op1,op2) returns TRUE if all invocations
   // of op1 and op2 commute (see algorithm below).
   if (not(commute(op1,op2)))
    return(FALSE)
 return(TRUE)

commute(op1, op2)
 // symbolic_exec(op1, op2) returns a tuple ⟨i, s⟩.
 // Here s is an expression denoting the multiset of methods
 // directly invoked as a result of executing first op1 then op2.
 // i is a set of bindings that provide, for each instance variable,
 // an expression denoting the new value of that instance
 // variable after the execution of first op1 then op2.
 ⟨i1, s1⟩ = symbolic_exec(op1, op2)
 ⟨i2, s2⟩ = symbolic_exec(op2, op1)
 for all v ∈ instance_variables(recv(op1))
  // simplify converts expressions to a simpler form.
  // compare compares expressions for equality.
  // Section 4.3 discusses these two algorithms.
  if(!compare(simplify(i1(v)),simplify(i2(v))))
   return(FALSE)
 if(!compare(simplify(s1),simplify(s2))
  return(FALSE)
 return(TRUE)
```

Figure 3: Commutativity Analysis and Commutativity Testing Algorithms

4.2 Expressions and Symbolic Execution

The analysis represents arithmetic values using expressions such as $\texttt{ex}_1 + \texttt{ex}_2$ and $-\texttt{ex}$. Conditional expressions such as $\texttt{if}(\texttt{cx}, \texttt{ex}_1, \texttt{ex}_2)$ represent the values of variables assigned different values on different branches of conditional statements. Updates such as $[\texttt{ex}_1 \rightarrow \texttt{ex}_2]$ represent assignments to array elements. An array expression $\texttt{ax}[\texttt{ex}_1 \rightarrow \texttt{ex}_2]$ represents the array whose value at the index denoted by $\texttt{ex}$ is $\texttt{ex}_2$ if $\texttt{ex}_1 = \texttt{ex}$ and $\texttt{ax}[\texttt{ex}]$ otherwise.

The analysis represents invoked methods using sequences of method invocation expressions. The expression $\texttt{ex}_1\texttt{->op}(\texttt{ex}_2)$ represents an invocation of the method $\texttt{op}$ with receiver $\texttt{ex}_1$ and parameter $\texttt{ex}_2$. Conditional expressions of the form $\texttt{if}(\texttt{cx}, \texttt{mx})$ represent methods only invoked in one branch of a conditional statement.

The compiler extracts expressions using symbolic execution. When it symbolically executes a method, the compiler maintains a set of bindings that provide, for each instance variable, an expression that denotes the variable's value at the current point in the execution. To execute a statement, the analysis symbolically evaluates the expression on the right hand side using the current set of bindings, then binds the computed expression to the variable on the left hand side of the assignment. To symbolically execute a conditional statement, the analysis evaluates both branches of the conditional, then combines the results to derive conditional expressions that denote the values of variables modified in one or both of the branches.

In general it is impossible to extract closed-form expressions for values computed in loops. We therefore recognize special cases of loops, in particular loops that do not access state modified during the course of the analyzed computation and loops that perform simple vector operations. A general solution is to replace loops (for analysis purposes only) with tail-recursive auxiliary methods.

4.3 Expression Simplification and Comparison

The expression comparison algorithm reduces the expressions to a simplified form, then applies a simple recursive isomorphism test to determine if the two expressions always denote the same value. The algorithm handles the following kinds of expressions:

- **Scalar Expressions :** The simplifier applies several kinds of simple scalar rules:
  - **Distribution:** Rules such as $\texttt{ex+if}(\texttt{cx}, \texttt{ex}_1, \texttt{ex}_2) = \texttt{if}(\texttt{cx}, \texttt{ex+ex}_1, \texttt{ex+ex}_2)$ distribute conditionals out of condition, scalar and array expressions. Rules such as $\texttt{-}(\texttt{ex}_1\texttt{+ex}_2) = (\texttt{-ex}_1\texttt{+-ex}_2)$ simplify arithmetic expressions.
  - **Simplification:** Rules such as $\texttt{--ex} = \texttt{ex}$ eliminate redundant operations.
  - **Binary Conversion:** Rules such as $((\texttt{ex}_1\texttt{+ex}_2)\texttt{+ex}_3) = (\texttt{ex}_1\texttt{+ex}_2\texttt{+ex}_3)$ and $(\texttt{ex}_1\texttt{+}(\texttt{ex}_2\texttt{+ex}_3)) = (\texttt{ex}_1\texttt{+ex}_2\texttt{+ex}_3)$ convert binary applications of commutative and associative operators such as $+$, $\times$, $\&\&$ and $||$ into n-ary applications. The algorithm also sorts the operands according to an arbitrary, recursively defined total order on expressions. This sort facilitates the eventual comparison by making it easier to identify isomorphic operands of commutative and associative operators.
- **Array Expressions:** The array expression simplification algorithm applies a rule that eliminates redundant updates and two rules that simplify array accesses:

  $\texttt{ax}[\texttt{ex}_1 \rightarrow \texttt{ex}_2] \cdots [\texttt{ex}_3 \rightarrow \texttt{ex}_4] = \texttt{ax} \cdots [\texttt{ex}_3 \rightarrow \texttt{ex}_4]$ if $\texttt{ex}_1 = \texttt{ex}_3$
  $\texttt{ax}[\texttt{ex}_1 \rightarrow \texttt{ex}_2][\texttt{ex}_3] = \texttt{ex}_2$ if $\texttt{ex}_1 = \texttt{ex}_3$
  $\texttt{ax}[\texttt{ex}_1 \rightarrow \texttt{ex}_2][\texttt{ex}_3] = \texttt{ax}[\texttt{ex}_3]$ if $\texttt{ex}_1 \neq \texttt{ex}_3$

  It also attempts to sort the update list, using the indexes as the sort key and an arbitrary, recursively defined total order on expressions as the sort order. Sorting the update list facilitates the eventual comparison by making it easier to identify isomorphic updates. The sort algorithm repeatedly identifies a pair of adjacent unsorted updates and attempts to replace the updates with an equivalent pair of sorted updates. The algorithm can replace the updates with any two updates that meet the following conditions:

  Observation 1 $\texttt{ax}[\texttt{ex}_1 \rightarrow \texttt{ex}_2][\texttt{ex}_3 \rightarrow \texttt{ex}_4] = \texttt{ax}[\texttt{ex}_5 \rightarrow \texttt{ex}_6][\texttt{ex}_7 \rightarrow \texttt{ex}_8]$ *if*
  - $\texttt{ex}_1 = \texttt{ex}_3$ *implies* $\texttt{ex}_3 = \texttt{ex}_7, \texttt{ex}_5 = \texttt{ex}_7, \texttt{ex}_4 = \texttt{ex}_8$ *and*
  - $\texttt{ex}_1 \neq \texttt{ex}_3$ *implies either* $\texttt{ex}_3 = \texttt{ex}_7, \texttt{ex}_1 = \texttt{ex}_5, \texttt{ex}_2 = \texttt{ex}_6, \texttt{ex}_4 = \texttt{ex}_8$ *or* $\texttt{ex}_3 = \texttt{ex}_5, \texttt{ex}_1 = \texttt{ex}_7, \texttt{ex}_2 = \texttt{ex}_8, \texttt{ex}_4 = \texttt{ex}_6$

  Because each update is generated by an assignment to an array element, the algorithm constructs the updates that correspond to executing the assignments in the reverse order. It then checks that this new pair of updates is sorted, and if so uses the conditions in Observation 1 to check if the new pair is equivalent to the original pair. The equality checking algorithm first assumes that $\texttt{ex}_1 = \texttt{ex}_3$, then attempts to check that $\texttt{ex}_3 = \texttt{ex}_7, \texttt{ex}_5 = \texttt{ex}_7, \texttt{ex}_4 = \texttt{ex}_8$ under this assumption. Before checking the equality conditions,

the algorithm first applies any of the array expression simplification rules enabled by the assumption. It goes through a similar process when it checks the second condition and assumes that $ex_1 \neq ex_3$. If the array assignments commute, the updates meet the conditions in Observation 1 and the algorithm can replace the original unsorted pair of updates with the new sorted pair.

- **Conditional Expressions :** The simplification algorithm builds a condition table for each expression that contains conditional subexpressions. This table enables the equality testing algorithm to use a simple isomorphism test for expressions containing conditionals. Each condition table contains the maximal condition-free subexpressions of the original expression. Each subexpression is stored under an index which consists of a conjunction of basic terms. If a subexpression is stored under a given index, it denotes the value of the original expression when all of the basic terms in the index are true. The algorithm builds the table by recursively traversing the outer conditional expressions to identify the minimal conjunctions of basic terms that select each maximal conditional-free subexpression as the value of the original expression. It is possible to further simplify the table using logic minimization techniques as proposed in [6].

- **Invoked Method Expressions :** The algorithm sorts the sequence of invoked method expressions. This sort facilitates the comparison of sets of invoked method expressions by making it easier to identify isomorphic expressions.

To compare two expressions for equality the algorithm performs a simple recursive isomorphism test. The algorithm checks that the condition tables have the same indexes and that corresponding subexpressions are isomorphic. In the worst case the expression manipulation algorithms may take exponential running time, although in practice we do not expect this worst-case behavior to prevent the application of commutativity analysis.

4.4 Code Generation

If it is possible to parallelize a method the compiler generates two methods: a parallel method and a wrapper method that invokes the parallel method then waits until the computation has finished. The parallel method directly invokes the parallel versions of all of the invoked methods. The compiler also augments the data structure declaration of the receiver object with a mutual exclusion lock. If the parallel method accesses an instance variable of the receiver that another potentially invoked method may modify, the parallel method uses the lock to ensure that executes atomically. The generated code acquires the lock before the first access to the receiver, then releases the lock before the first operation invocation site. The compiler may generate multiple lock release sites; only one will be executed per invocation. Figure 2 contains an example of the generated code.

The compiler also applies an optimization that exposes parallel loops to the run time system. If a loop contains nothing but parallel method invocations, the compiler generates parallel loop code instead of code that serially spawns the parallel operations in the loop.

5 Experimental Results

We have conducted an experimental feasibility study designed to evaluate how well commutativity analysis works in practice for a given application. This study evaluates the ability of commutativity analysis to expose concurrency in the application, the performance improvements that result from exploiting this concurrency, and the overhead of the dynamic lock operations and task management required to execute the generated parallel code.

We performed this feasibility study on a complete scientific application — the Barnes-Hut hierarchical N-body solver [2]. The Barnes-Hut is a challenging application because it manipulates a recursive, pointer-based data structure – a space subdivision tree. We applied commutativity analysis by hand to this application to generate parallel code.[2] We then executed the generated code on a shared memory multiprocessor (the Stanford DASH machine [8]), comparing its performance with that of a highly optimized, explicitly parallel version of the same computation from the SPLASH-2 benchmark set. The fact that the two versions exhibit comparable performance demonstrates that a compiler can use commutativity analysis to successfully parallelize this application.

5.1 The Barnes-Hut application

The Barnes-Hut algorithm simulates the trajectories of a set of interacting bodies under Newtonian forces. The algorithm improves on the all-pairs computation by approximating an interaction between a body and a cluster of distant bodies with a single interaction between the body and the combined center of mass of the cluster. We refer to an interaction between a body and a cluster as a *body-cell* interaction, and an interaction between two "close" bodies as a *body-body* interaction. The algorithm organizes the computation using a space subdivision tree, completely reconstructing the tree at each iteration of the system. Internal

[2]The sequential source code and the parallel generated code can be found at `http://www.cs.ucsb.edu/~pedro/CA/barnes`

nodes of the tree are called cells; each cell represents a rectangular spatial domain. The immediate descendants of a given cell evenly subdivide its spatial domain. The tree leaves store the bodies.

Each iteration of the algorithm performs the following four steps. It first constructs the space subdivision tree by inserting the bodies one by one into an initially empty tree. It then computes, for each cell in the tree, the center of mass of the bodies in that cell. It next computes, for each body, the force acting on that body from other bodies in the system. It finally uses the computed forces to advance each body in the system. The commutativity analysis algorithm is capable of exploiting the concurrency in the force calculation and position/velocity update phases. Because of space constraints we outline here the force calculation step only. The position/velocity update phases are in essence similar and have been omitted. In [5] we present a full description of the analyzed code and a discussion of commuting operations.

The force computation step consists of a loop that invokes the force calculation computation for each body. Each force computation recursively traverses the space subdivision tree. At each spatial cell it computes the validity of the center-of-mass interaction approximation. If the approximation is valid the algorithm avoids a recursive traversal of the subtree rooted at the cell by computing a *body-cell* interaction. If the approximation is not valid the algorithm recursively traverses all of the cell's children. If the traversal reaches a leaf it computes a *body-body* interaction.

Whenever it computes an interaction, the algorithm updates both the acceleration and the potential field vectors of the body. For any two interactions these updates commute. All tree cell interaction operations (which compute *body-cell* interactions) and body interaction operations (which compute *body-body* interactions) therefore commute. All the remaining operations in the force computation step only read variables that none of the operations in the step modify. They are therefore independent of the operations that update the body objects. The compiler can therefore generate code that executes all iterations of the force computation loop in parallel.

5.2 Results

Starting with an object-based version of the code, we generated parallel code for the force calculation and position/velocity phases. We also implemented a run-time library that provided the basic synchronization and concurrency management functionality. We expect the compiler to generate the same code and use the same library. We ran this version on the Stanford DASH machine. We compare its performance with that of the *barnes* code, a highly tuned hand-parallelized code from the SPLASH-2 benchmark set [16]. Tables 4 and 5 present the timing results.

Number of Bodies	Processors				
	1	2	4	8	16
1024	7.94	4.05	2.19	1.38	0.92
2048	20.46	10.70	5.72	3.37	2.18
4096	51.23	26.82	14.20	8.09	5.16
8192	131.2	64.10	33.88	15.59	11.56
16384	301.4	146.9	77.56	45.60	26.13

Figure 4: Execution times for commutativity analysis version (seconds).

Number of Bodies	Processors				
	1	2	4	8	16
1024	9.15	4.58	2.46	2.39	2.07
2048	22.36	11.78	6.04	3.65	4.08
4096	55.51	27.40	14.95	8.06	5.90
8192	127.1	65.67	35.79	19.73	11.60
16384	277.1	140.5	76.70	39.63	22.69

Figure 5: Execution times for *barnes* from the SPLASH-2 benchmark set (seconds).

The performance of the commutativity analysis version is comparable to the performance of the highly tuned SPLASH-2 code. Both computations scale reasonably well as the number of processors increases. This is a very encouraging result considering the amount of effort and sophistication of the optimizations in the *barnes* code.

6. Related Work

Compiler research on automatically parallelizing serial codes that manipulate dynamic, pointer-based data structures has focused on techniques that precisely represent the run-time topology of the heap [7, 10]. Ideally the compiler can use this representation to discover independent pieces of code. Unlike commutativity analysis, these techniques must analyze the code that builds the data structure and must propagate the results of this analysis through the program to the parallel sections that use the data structure. There are both advantages and disadvantages of data dependence analysis when compared with commutativity analysis. If the compiler can use data dependence analysis to discover independent pieces of code, it can parallelize the code without having to extract any further properties. For commutativity analysis to generate parallel code, all operations in the computation must commute. The advantages of commutativity analysis are its ability to parallelize computations that manipulate graphs, the fact that it eliminates the need to perform complex analysis to extract global properties of the data structure, the fact that it eliminates the need to propagate extracted properties of the data structure topology across large sections of the program and the fact that

it can parallelize code that manipulates pointer-based data even in the absence of the code that originally constructed the data structure.

Several existing compilers can recognize when a loop performs a reduction of many values into a single value [6, 4]. These compilers recognize when the reduction primitive (typically addition) is associative. They then exploit this algebraic property to eliminate the data dependence associated with the serial accumulation of values into the result. The generated program computes the reduction in parallel. These techniques exploit an associative property of language primitives in a specific context (parallel loops). Commutativity analysis is designed to recognize when programmer-defined operations commute and to exploit this property in the context of arbitrary computations on arbitrary programmer-defined objects.

Other researchers have recognized the value of including support for commuting operations in parallel computing systems [3, 15, 14]. These systems focus on exploiting commuting operations and rely on some external mechanism, typically the programmer, to specify when the operations actually commute. The goal of the presented research is to automatically recognize and exploit commuting operations.

7. Conclusion

Existing parallelizing compilers all preserve the data dependences of the original serial program. We believe that this strategy is too conservative: compilers must recognize and exploit commuting operations if they are to effectively parallelize a range of applications. This paper presents an algorithm that can statically recognize and exploit commuting operations to automatically generate parallel code. We state a general set of conditions that is sufficient to guarantee commutativity. We present a static analysis algorithm that uses symbolic execution and expression manipulation to determine if operations commute and show how a parallelizing compiler can exploit commuting operations to generate parallel code. Finally, we present an experiment that demonstrates that at least one application, the Barnes-Hut hierarchical N-body solver, exhibits good parallel performance under this approach.

References

[1] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993.

[2] J. Barnes and P. Hut. A hierarchical O(NlogN) force-calculation algorithm. *Nature*, pages 446–449, December 1976.

[3] P. Barth, R. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *Proceedings of the Fifth ACM Conference on Functional Programming Languages and Computer Architecture*, pages 538–568. Springer-Verlag, August 1991.

[4] D. Callahan. Recognizing and parallelizing bounded recurrences. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[5] P. Diniz and M. Rinard. Exploiting commuting operations in parallelizing serial programs. Technical Report TRCS95-11, Dept. of Computer Science, University of California at Santa Barbara, January 1995.

[6] A. Fisher and A. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*, Orlando, FL, June 1994.

[7] L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: improving the analysis and transformation of imperative programs. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[8] D. Lenoski. *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor*. PhD thesis, Stanford, CA, February 1992.

[9] E. Mohr, D. Kranz, and R. Halstead. Lazy task creation: a technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 185–197, June 1990.

[10] J. Plevyak, V. Karamcheti, and A. Chien. Analysis of dynamic structures for efficient parallel execution. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[11] M. Rinard and P. Diniz. Automatically parallelizing serial programs using commutativity analysis. Technical Report TRCS95-13, Dept. of Computer Science, University of California at Santa Barbara, July 1995.

[12] D. Scales and M. S. Lam. An efficient shared memory system for distributed memory machines. Technical Report CSL-TR-94-627, Computer Systems Laboratory, Stanford University, July 1994.

[13] J. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[14] J. Solworth and B. Reagan. Arbitrary order operations on trees. In *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Portland, OR, August 1993.

[15] G. Steele. Making asynchronous parallelism safe for the world. In *Proceedings of the Seventeenth Annual ACM Symposium on the Principles of Programming Languages*, pages 218–231, San Francisco, CA, January 1990.

[16] S. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.