Semantic Foundations of Commutativity Analysis

Martin C. Rinard[†] and Pedro C. Diniz[‡]

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
{martin,pedro}@cs.ucsb.edu
http://www.cs.ucsb.edu/~{martin,pedro}

Abstract. This paper presents the semantic foundations of commutativity analysis, an analysis technique for automatically parallelizing programs written in a sequential, imperative programming language. Commutativity analysis views the computation as composed of operations on objects. It then analyzes the program at this granularity to discover when operations 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. This paper shows that the basic analysis technique is sound. We have implemented a parallelizing compiler that uses commutativity analysis as its basic analysis technique; this paper also presents performance results from two automatically parallelized applications.

1 Introduction

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

This paper presents the semantic foundations of a new analysis technique called commutativity analysis. Instead of preserving the relative order of individual reads and writes to single words of memory, commutativity analysis views the computation as composed of operations on objects. It then analyzes the computation at this granularity to discover when pieces of the computation 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. While the resulting parallel program may violate the data dependences of the original serial program, it is still guaranteed to generate the same result.

We expect commutativity analysis to eliminate many of the limitations of existing approaches. Compilers that use commutativity analysis will be able

[†] Supported in part by an Alfred P. Sloan Research Fellowship.
[‡] Sponsored by the PRAXIS XXI program administrated by Portugal's JNICT – Junta Nacional de Investigação Científica e Tecnológica, and holds a Fulbright travel grant.

to automatically generate parallel code for many applications that periodically update shared data structures using commuting operations and/or manipulate recursive, pointer-based data structures such as lists, trees and graphs. Commutativity analysis allows compilers to automatically parallelize computations even though they have no information about the global topology of the manipulated data structure. It may therefore be especially useful 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) that analyzes the code to verify or discover global properties of the data structure topology.

The rest of the paper is structured as follows. In Section 2 we present an example that illustrates how commutativity analysis can automatically parallelize graph traversals. In Section 3 we describe the basic approach and informally state the conditions that the compiler uses to recognize commuting operations. In Section 4 we provide a formal model of computation and define what it means for two operations to commute in this model. This section contains a key theorem establishing that if all of the operations in a computation commute, than all parallel executions are equivalent to the serial execution. In Section 5 we state that we have developed the analysis algorithms required for the practical application of commutativity analysis and provide a reference to a technical report that presents the analysis algorithms. In Section 6 we present experimental results for two complete scientific applications parallelized by a prototype compiler that uses commutativity analysis as its basic analysis technique. In Section 7 we conclude.

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++.

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

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

Fig. 1. Serial Graph Traversal

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

Fig. 2. Parallel Graph Traversal

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 the entire parallel computation completes. The `parallel_visit` method executes the recursive calls concurrently using the `spawn` construct, which creates a new task for the execution of each method invocation. The compiler also augments the `graph` data structure with a mutual exclusion lock `mutex`. The `parallel_visit` method uses this lock to ensure that all of its invocations execute atomically.

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 executes the code in that method. Each object implements its state using a set of instance variables. When an operation 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, or methods whose 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 — although the current compiler does not do so, it is possible to automatically convert any method into a collection of separable methods via the introduction of auxiliary methods.

The following conditions, which the compiler can use to test if two operations A and B commute, form the foundation of commutativity analysis.

1. **(Instance Variables)** The new value of each instance variable of the receiver objects of A and B must be the same after the execution of the object section of A followed by the object section of B as after the execution of the object section of B followed by the object section of A.
2. **(Invoked Operations)** The multiset of operations directly invoked by either A or B under the execution order A followed by B must be the same as the multiset of operations directly invoked by either A or B under the execution order B followed by A. [1]

Note that these conditions do not deal with the entire recursively invoked computation that each operation generates — they only deal with the object and

[1] Two operations are the same if they execute the same method and have the same parameters.

invocation sections of the two operations. Furthermore, they are not designed to test that the entire computations of the two operations commute. They only test that the object sections of the two operations commute and that the operations together invoke the same multiset of operations regardless of the order in which they execute. As we argue below, if all pairs of operations in the computation satisfy the conditions, then all parallel executions generate the same result as the serial execution.

The instance variables condition ensures that if the parallel execution invokes the same multiset of operations as the serial execution, the values of the instance variables will be the same at the end of the parallel execution as at the end of the serial execution. The basic reasoning is that for each object, the parallel execution will execute the object sections of the operations on that object in some arbitrary order. The instance variables condition ensures that all orders yield the same final result.

The invoked operations condition provides the foundation for the application of the instance variables condition: it ensures that all parallel executions invoke the same multiset of operations (and therefore execute the same object sections) as the serial execution.

Both commutativity testing conditions are trivially satisfied if the two operations have different receivers — in this case their executions 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 followed by B. The other denotes the new value under the execution order B followed by 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.

To use the commutativity testing conditions, the compiler first computes a conservative approximation to the set of methods invoked as a result of executing a given computation. The compiler then applies the commutativity testing conditions 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 Formal Treatment

We next provide a formal treatment of how commutativity enables parallel execution. We first present a formal model of computation for separable operations. We define parallel and serial execution in this model, and define what it means for two operations to commute. The key theorem establishes that if all of the

operations in the parallel executions commute, then all parallel executions are equivalent to the serial execution.

4.1 The Model of Computation

We next describe a formal model of computation for separable programs. We assume a set of objects $r, o \in O$, a set of constants $\mathtt{c} \in \mathtt{C} \subseteq O$, a set of instance variable names $\mathtt{v} \in \mathtt{IV}$ and a set of operation names $\mathtt{op} \in \mathtt{OP}$. The operational semantics uses several functions. We model instance variable values with functions $i : I = \mathtt{IV} \to O$. We say that an object r is in state i if the value of each instance variable v of r is $i(\mathtt{v})$. We model object memories with functions $m : M = O \to I$. We also have operations $a \in A = O \times \mathtt{OP} \times O$, and write an element a in the form $r\text{->}\mathtt{op}(o)$. Each operation has a receiver r, an operation name op and a parameter o. To simplify the presentation we assume that each operation takes only one parameter, but the framework generalizes in a straightforward way to include operations with multiple parameters.

We next describe how we model computation. The execution of each invoked operation first updates the receiver, then invokes several other operations. We model the effect on the receiver with a receiver effect function $R : A \times I \to I$. $R(r\text{->}\mathtt{op}(o), i)$ is the new state of r when operation $r\text{->}\mathtt{op}(o)$ executes with the receiver r in state i; it models the effect of the operation's object section on the receiver.

We model the invoked operations using sequences $s \in S = seq(A)$ of the form $a_1 \circ \cdots \circ a_k$. ϵ is the empty sequence, and $\epsilon \circ s = s \circ \epsilon = s$. We use an invoked operation function $N : A \times I \to seq(A)$ to model the sequence of operations invoked as a result of executing an operation. $N(r\text{->}\mathtt{op}(o), i)$ is the sequence of operations directly invoked when $r\text{->}\mathtt{op}(o)$ executes with the receiver r in state i; it models the multiset of operations invoked in the invocation section of the operation.

The serial operational semantics of the program uses a transition function $\to$ on serial states $\langle m, s \rangle \in M \times seq(A)$, where m is the current object memory and s is the sequence of operations left to invoke. $\to$ models the execution of the next operation in s. As Definition 1 shows, $\to$ updates the memory to reflect the new state of the receiver of the executed operation. It also removes the executed operation from the current sequence of operations left to invoke and prepends the multiset of operations that the executed operation invokes. The operations therefore execute in the standard depth-first execution order. Strictly speaking, $\to$ depends on R and N, but we do not represent this dependence explicitly as the correct R and N are always obvious from context.

Definition 1. $\to$ is the function on $M \times seq(A)$ defined by:

$$\frac{m' = m[r \mapsto R(r\text{->}\mathtt{op}(o), m(r))], s' = N(r\text{->}\mathtt{op}(o), m(r)) \circ s}{\langle m, r\text{->}\mathtt{op}(o) \circ s \rangle \to \langle m', s' \rangle}$$

The states $\langle m, p \rangle$ in the parallel operational semantics are similar to the states in the serial operational semantics, but have a multiset p, rather than a

sequence s, of operations left to invoke. Such multisets $p \in mst(A)$ are of the form $\{a_1\} \uplus \cdots \uplus \{a_k\}$. We use the sequence to multiset function $s2p : seq(A) \rightarrow mst(A)$ defined by $s2p(a_1 \circ \cdots \circ a_k) = \{a_1\} \uplus \cdots \uplus \{a_k\}$ to map sequences of operations into the corresponding multisets.

We model parallel execution by generating all possible interleavings of the execution of the operations. In the parallel execution any of the current operations may execute next, with each operation taking the current state to a potentially different state than the other operations. The parallel operational semantics therefore models execution using a transition relation $\Rightarrow$ rather than a transition function. As the following definition illustrates, $\Rightarrow$ relates a state to all of the states reachable via the execution of any of the current multiset of operations left to invoke. The definition of $\Rightarrow$ completes the semantic framework required to precisely state when two operations commute.

Definition 2. $\Rightarrow$ is the smallest relation [2] on $M \times mst(A)$ satisfying the following condition:

$$\frac{\langle m, r\text{->op}(o)\rangle \rightarrow \langle m', s'\rangle, p' = s2p(s') \uplus p}{\langle m, \{r\text{->op}(o)\} \uplus p\rangle \Rightarrow \langle m', p'\rangle}$$

For two operations to commute, they must leave their receivers in identical states and invoke the same multiset of operations regardless of the order in which they execute.

Definition 3. $r_1\text{->op}_1(o_1)$ and $r_2\text{->op}_2(o_2)$ commute in m if
$\langle m, \{r_1\text{->op}_1(o_1)\}\rangle \Rightarrow \langle m_1, p_1\rangle$ and $\langle m_1, \{r_2\text{->op}_2(o_2)\}\rangle \Rightarrow \langle m_{12}, p_{12}\rangle$ and
$\langle m, \{r_2\text{->op}_2(o_2)\}\rangle \Rightarrow \langle m_2, p_2\rangle$ and $\langle m_2, \{r_1\text{->op}_1(o_1)\}\rangle \Rightarrow \langle m_{21}, p_{21}\rangle$
implies $m_{12} = m_{21}$ and $p_1 \uplus p_{12} = p_2 \uplus p_{21}$

Definition 4. $r_1\text{->op}_1(o_1)$ and $r_2\text{->op}_2(o_2)$ commute if $\forall m : r_1\text{->op}_1(o_1)$ and $r_2\text{->op}_2(o_2)$ commute in m.

Theorem 5. *If* $r_1 \neq r_2$ *then* $r_1\text{->op}_1(o_1)$ *and* $r_2\text{->op}_2(o_2)$ *commute.*

Proof Sketch: Intuitively, if the receivers are different, the operations are independent because they access disjoint pieces of data.

4.2 Correspondence Between Parallel and Serial Semantics

We next present several lemmas that characterize the relationship between the parallel and serial operational semantics. Lemma 6 says that for each partial serial execution, there exists a partial parallel execution that generates an equivalent state.

Lemma 6.
If $\langle m, r\text{->op}(o)\rangle \rightarrow \cdots \rightarrow \langle m', s\rangle$ *then* $\langle m, \{r\text{->op}(o)\}\rangle \Rightarrow \cdots \Rightarrow \langle m', s2p(s)\rangle$.

[2] Under the subset ordering on relations considered as sets of pairs defined as follows. Consider two relations $\Rightarrow_1, \Rightarrow_2 \subseteq (M \times mst(A)) \times (M \times mst(A))$. $\Rightarrow_1$ is less than $\Rightarrow_2$ if $\Rightarrow_1 \subset \Rightarrow_2$.

Proof Sketch: At each step the parallel execution may execute the same operation as the serial execution.

We next establish the impact that commutativity has on the different parallel executions. Lemma 8 states that if all of the invoked operations in given computation commute and one of the parallel executions terminates, then all parallel executions terminate with the same memory. This lemma uses the function $gen : M \times A \rightarrow 2^A$, which tells which operations can be invoked as a result of invoking a given operation.

Definition 7. $gen(m, r\text{->}\mathtt{op}(o)) = \cup\{p : \langle m, \{r\text{->}\mathtt{op}(o)\}\rangle \Rightarrow \cdots \Rightarrow \langle m', p\rangle\}$

Lemma 8. *If* $\forall r_1\text{->}\mathtt{op}_1(o_1), r_2\text{->}\mathtt{op}_2(o_2) \in gen(m, r\text{->}\mathtt{op}(o)) : r_1\text{->}\mathtt{op}_1(o_1)$ *and* $r_2\text{->}\mathtt{op}_2(o_2)$ *commute, then* $\langle m, \{r\text{->}\mathtt{op}(o)\}\rangle \Rightarrow \cdots \Rightarrow \langle m_1, \emptyset\rangle$ *implies*

- *not* $\langle m, \{r\text{->}\mathtt{op}(o)\}\rangle \Rightarrow \cdots$ *(i.e. there is no infinite parallel execution), and*
- $\langle m, \{r\text{->}\mathtt{op}(o)\}\rangle \Rightarrow \cdots \Rightarrow \langle m_2, \emptyset\rangle$ *implies* $m_1 = m_2$

Proof Sketch: If all of the invoked operations commute then the transition system is confluent, which guarantees deterministic execution [3].

Lemma 9 characterizes the situation when the computation may not terminate. It says that if all of the operations invoked in the parallel executions commute, then it is possible to take any two partial parallel executions and extend them to identical states.

Lemma 9. *If* $\forall r_1\text{->}\mathtt{op}_1(o_1), r_2\text{->}\mathtt{op}_2(o_2) \in gen(m, r\text{->}\mathtt{op}(o)) : r_1\text{->}\mathtt{op}_1(o_1)$ *and* $r_2\text{->}\mathtt{op}_2(o_2)$ *commute, then* $\langle m, \{r\text{->}\mathtt{op}(o)\}\rangle \Rightarrow \cdots \Rightarrow \langle m_1, p_1\rangle$ *and*
$\langle m, \{r\text{->}\mathtt{op}(o)\}\rangle \Rightarrow \cdots \Rightarrow \langle m_2, p_2\rangle$ *implies*
$\exists m' \in M, p \in mst(A) : \langle m_1, p_1\rangle \Rightarrow \cdots \Rightarrow \langle m', p\rangle$ *and* $\langle m_2, p_2\rangle \Rightarrow \cdots \Rightarrow \langle m', p\rangle$

Proof Sketch: If all of the invoked operations commute then the transition system is confluent, which guarantees deterministic execution [3].

An immediate corollary of these two lemmas is that if the serial computation terminates, then all parallel computations terminate with identical memories. Conversely, if a parallel computation terminates, then the serial computation also terminates with an identical memory.

5 Analysis

We have developed a formal semantics that, given a program, defines the receiver effect and invoked operation functions for that program [6]. We have also developed a static analysis algorithm that analyzes pairs of methods to determine if they meet the commutativity testing conditions in Section 3. The foundation of this analysis algorithm is symbolic execution [4]. Symbolic execution simply executes the methods, computing with expressions instead of values. It maintains a set of bindings that map variables to the expressions that denote their

values and updates the bindings as it executes the methods. To test if the executions of two methods commute, the compiler first uses symbolic execution to extract expressions that denote the new values of the instance variables and the multiset of invoked operations for both execution orders. It then simplifies the expressions and compares corresponding expressions for equality. If the expressions denote the same value, the operations commute. We have proved a correspondence between the static analysis algorithm and the formal semantics, and used the correspondence to prove that the algorithms used in the compiler correctly identify parallelizable computations [6].

6 Experimental Results

We have implemented a prototype parallelizing compiler that uses commutativity analysis as its basic analysis technique. The compiler also uses several other analysis techniques to extend the model of computation significantly beyond the basic model of computation presented in Section 3 [5].

We used the compiler to automatically parallelize two applications: the Barnes-Hut hierarchical N-body code [2] and Water, which evaluates forces and potentials in a system of water molecules in the liquid state. We briefly present several performance results; we provide a more complete description of the applications and the experimental methodology elsewhere [5].

Figure 3 presents the speedup curve for the Barnes-Hut on two input data sets; this graph plots the running time of the sequential version running with no parallelization overhead divided by the running time of the automatically parallel version as a function of the number of processors executing the parallel computation. The primary limiting factor on the speedup is the fact that the compiler does not parallelize one of the phases of the computation; as the number of processors grows that phase becomes the limiting factor on the performance [5].

Figure 4 presents the speedup curve for Water running on two input data sets. The limiting factor on the speedup is contention for shared objects updated by multiple operations [5].

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 the semantic foundations of commutativity analysis and shows that the basic analysis technique is sound. It also presents experimental results from two complete scientific applications that were successfully and automatically parallelized by a prototype compiler that uses commutativity analysis as its basic analysis technique. Both applications exhibit respectable parallel performance.

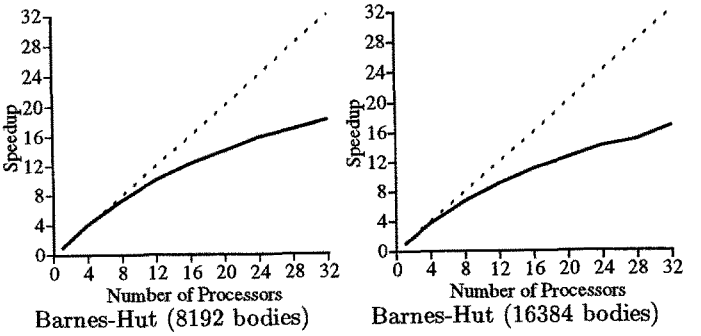

Fig. 3. Speedup for Barnes-Hut

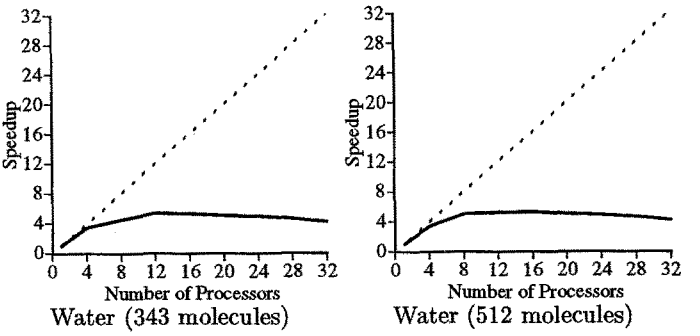

Fig. 4. Speedup for Water

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. G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.
4. R. Kemmerer and S. Eckmann. UNISEX: a UNIx-based Symbolic EXecutor for pascal. *Software—Practice and Experience*, 15(5):439–458, May 1985.
5. M. Rinard and P. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. Technical Report TRCS96-08, Dept. of Computer Science, University of California at Santa Barbara, May 1996.
6. M. Rinard and P. Diniz. Semantic foundations of commutativity analysis. Technical Report TRCS96-09, Dept. of Computer Science, University of California at Santa Barbara, May 1996.