Commutativity Analysis:

A New Analysis Framework for Parallelizing Compilers

Martin C. Rinard (martin@cs.ucsb.edu) *
Pedro C. Diniz (pedro@cs.ucsb.edu) †
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106

Abstract

This paper presents a new analysis technique, commutativity analysis, for automatically parallelizing computations that manipulate dynamic, pointer-based data structures. 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 final 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.

We have implemented a prototype compilation system that uses commutativity analysis as its primary analysis framework. We have used this system to automatically parallelize two complete scientific computations: the Barnes-Hut N-body solver and the Water code. This paper presents performance results for the generated parallel code running on the Stanford DASH machine. These results provide encouraging evidence that commutativity analysis can serve as the basis for a successful parallelizing compiler.

1 Introduction

Parallelizing compilers promise to dramatically reduce the difficulty of developing software for parallel computing environments. Existing parallelizing compilers use data dependence analysis to detect independent computations (two computations are independent if neither accesses data that the other writes), then generate code that executes these computations in parallel. In the right context this approach works well — researchers have successfully used data dependence analysis to parallelize computations that manipulate dense arrays using affine access functions [1, 30, 12, 16]. But data dependence analysis is, by itself, inadequate for computations that manipulate dynamic, pointer-based data structures. Its limitations include a need to perform complicated analysis to extract global properties of the data structure topology and an inherent inability to parallelize computations that manipulate graphs [2].

We believe the key to automatically parallelizing dynamic, pointer-based computations is to recognize and exploit commuting operations, or operations that generate the same final result regardless of the order in which they execute. Even though traditional compilers have not exploited commuting operations, these operations play an important role in other areas of parallel computing. Explicitly parallel programs, for example, often use locks, monitors and critical regions to ensure that operations execute atomically [22]. For the program to execute correctly, the programmer must ensure that all of the atomic operations commute. Four of the six parallel applications in the SPLASH benchmark suite [37] and three of the four parallel applications described in [35] rely on commuting operations to expose the concurrency and generate correct parallel execution. This experience suggests that compilers will be unable to parallelize a wide range of computations unless they to recognize and exploit commuting operations.

We have developed a new analysis framework called commutativity analysis. This framework is designed to automatically recognize and exploit commuting operations to generate parallel code. It views the computation as composed of arbitrary operations on arbitrary objects. It then analyzes the computation at this granularity to determine if operations commute. If all of the operations in a given computation commute, the compiler can automatically generate parallel code. Even though the code may violate the data dependences of the original serial program, it is still guaranteed to generate the same result.

We have built a complete prototype compilation system based on commutativity analysis. This compilation system is designed to automatically parallelize unannotated programs written in a subset of C++. The dynamic nature of our target application set means that the compiler must rely on a run-time system to provide basic task management functionality such as synchronization and dynamic load balancing. We have implemented a run-time system that provides this functionality. It currently runs on the Stanford DASH multiprocessor [25] and on multiprocessors from Silicon Graphics.

We have used the compilation system to automatically par-

* 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.

allelize two complete scientific applications: the Barnes-Hut N-body solver [3] and the Water code [42]. The Barnes-Hut is representative of our target class of dynamic computations: it performs well because it uses a pointer-based data structure (a space subdivision tree) to organize the computation. The Water code is a more traditional scientific computation that organizes its data as arrays of objects representing water molecules. We have collected performance results for the generated parallel code running on the Stanford DASH machine. These results indicate that commutativity analysis may be able to serve as the basis for a successful parallelizing compiler.

This paper makes the following contributions:

- It describes an new analysis framework, commutativity analysis, that can automatically recognize and exploit commuting operations to generate parallel code.
- It describes extensions to the basic commutativity analysis framework. These extensions significantly increase the range of programs that commutativity analysis can effectively parallelize.
- It presents several analysis algorithms that a compiler can use to automatically recognize commuting operations, discover parallelizable computations and generate parallel code.
- It presents performance results for automatically parallelized versions of two scientific computations. These results support the thesis that it is possible to use commutativity analysis as the basis for a successful parallelizing compiler.

Although we designed commutativity analysis to parallelize serial programs, it may also benefit other areas of computer science. For example, commuting operations allow computations on the persistent data in object-oriented databases to execute in parallel. Transaction processing systems can exploit commuting operations to use more efficient locking algorithms [40]. Commuting operations make protocols from distributed systems easier to implement efficiently; the corresponding reduction in the size of the associated state space may make it easier to verify the correctness of the protocol. In all of these cases the system relies on commuting operations for its correct operation. Automatically recognizing or verifying that operations commute may therefore increase the efficiency, safety and/or reliability of these systems.

The remainder of the paper is structured as follows. Section 2 presents an example that shows how commuting operations enable parallel execution. Section 3 presents an overview of the commutativity analysis framework. Section 4 presents the analysis algorithms that the compiler uses. Section 5 describes how the compiler generates parallel code. Section 6 presents the experimental performance results for two automatically parallelized applications. Section 7 describes some directions for future research. We survey related work in Section 8 and conclude in Section 9.

2 An Example

This section presents an example that shows how commuting operations enable parallel execution. The `visit` operation 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 mark;
  int val, sum;
  graph *left; graph *right;
};

graph::visit(int p) {
  sum = sum + p;
  if (!mark) {
    mark = TRUE;
    if (left != NULL)
      left->visit(val);
    if (right != NULL)
      right->visit(val);
  }
}
```

Figure 1: Serial Graph Traversal

The traversal generates one `visit` operation for each edge in the graph. Each operation traverses a graph node; this node is called the receiver of the operation. Each `visit` operation takes as a parameter `p` the value of the instance variable `val` of the node that points to the receiver. The 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` operation 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` operation in Figure 2. The top level `visit` operation first invokes the `parallel_visit` operation, then invokes the `wait` construct, which blocks until all parallel tasks created by the current task or its descendant tasks finishes. The `parallel_visit` operation executes the recursive calls concurrently using the `spawn` construct, which creates a new task for each operation. A straightforward application of lazy task creation techniques [26] can increase the granularity of the resulting parallel computation. The compiler also augments each `graph` node with a mutual exclusion lock `mutex`. The generated parallel operations use this lock to ensure that they execute atomically.

```
class graph {
  lock mutex;
  boolean mark;
  int val, sum;
  graph *left; graph *right;
};

graph::visit(int p) {
  this->parallel_visit(p);
  wait();
}

graph::parallel_visit(int p) {
  mutex.acquire();
  sum = sum + p;
  if (!mark) {
    mark = TRUE;
    mutex.release();
    if (left != NULL)
      spawn(left->parallel_visit(val));
    if (right != NULL)
      spawn(right->parallel_visit(val));
  } else {
    mutex.release();
  }
}
```

Figure 2: Parallel Graph Traversal

3 Basic Concepts

Commutativity analysis exploits the structure present in object-based programs to guide the parallelization process. In this section we present the basic concepts behind this approach.

3.1 Model of Computation

We explain the basic model of computation for commutativity analysis as applied to pure object-based programs. Such programs structure the computation as a sequence of operations on objects. Each object implements its state using a set of instance variables. Each instance variable can be either a nested object, a primitive type from the underlying language such as an integer, a pointer to an object or a double, or an array of nested objects or primitive types. In the example in Figure 2 each graph node is an object.

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. Each operation has a receiver object and several parameters that are passed by value to the operation. When an operation executes it can access the parameters, invoke other operations or access the instance variables of the receiver. There are several restrictions on instance variable access. If the instance variable is an instance variable of a nested object, the operation can not directly access the instance variable — it can only access the variable indirectly by invoking operations that have the nested object as the receiver. If the instance variable is declared in a parent class from which the receiver's class inherits, the operation can not directly access the instance variable — it can only access the variable indirectly by invoking operations whose receiver's class is the parent class.

Commutativity analysis is designed to work with *separable* operations. An operation is separable if it 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. The motivation for separability is that the commutativity testing algorithm (which determines if operations commute) requires that each operation's accesses to the receiver execute atomically with respect to the operations that it invokes. Separability ensures that the actual computation obeys this constraint. Separability imposes no expressibility limitations — it is possible to automatically decompose any method into a collection of methods whose invocations are all separable via the introduction of auxiliary methods.

3.2 Extents

A policy in the compiler must choose computations to attempt to parallelize. The current policy is that the compiler analyzes one computation for each method; that computation consists of all operations either directly or indirectly executed as a result of executing the given method. The compiler computes a conservative approximation to the set of executed operations. This approximation is called the *extent* of the method. If the compiler can verify that all pairs of operations in an extent commute, it marks the method that generated the extent as a parallel method. If some of the pairs may not commute, it marks the method as a serial method.

3.3 Commutativity Testing

The foundation of commutativity analysis is a set of conditions that the compiler can use to test if two operations A and B commute. These 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. The two operations commute if they meet the following commutativity testing 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 or if neither operation writes an instance variable that the other accesses — in both of these cases the operations are independent. If the operations may not be independent, the compiler reasons about the values computed in the two execution orders. We illustrate this concept by applying it to the `sum` instance variable in the example in Figure 1. We assume two invocations `r->visit(p1)` and `r->visit(p2)` of the `visit` operation. `r->visit(p1)` has parameter `p1`, `r->visit(p2)` has parameter `p2` and both operations have the same receiver `r`.

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 operation executes. It is possible to determine by algebraic reasoning that both expressions denote the same value.[1] The compiler can use a similar approach to discover that the values of the other instance variables are the same in both execution orders and that together the operations always directly invoke the same multiset of operations.

[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.

Execution Order	New Value of `sum`
`r->visit(p1);r->visit(p2)`	`(sum+p1)+p2`
`r->visit(p2);r->visit(p1)`	`(sum+p2)+p1`

Table 1: New Values of `sum` Under Different Execution Orders

3.4 Symbolic Execution

The compiler uses symbolic execution [20] to extract the expressions that denote the new values of instance variables and the multiset of invoked operations. 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.

In certain circumstances the compiler may be unable to extract expressions that precisely represent the values that an operation computes. In the current compiler this may happen, for example, if the method contains unstructured flow of control constructs such as `goto` constructs. In this case the compiler marks the method as unanalyzable; the commutativity testing phase conservatively assumes that invocations of unanalyzable methods commute with no operation.

3.5 Extensions

We have found it useful to extend the analysis framework to handle several situations that fall outside the basic model of computation outlined in Section 3.1. These extensions significantly increase the range of programs that the compiler can successfully analyze.

3.5.1 Extent Constants

All of the conditions in the commutativity testing algorithm check expressions for equality. In certain cases the compiler may be able to prove that two values are equal without representing the values precisely in closed form. Consider the execution of an operation in the context of a given extent. If the operation reads a variable that none of the operations in the extent write, the variable will have the same value regardless of when the operation executes relative to all of the other operations in the extent. We call such a variable an extent constant variable.

If the operation computes a value that does not depend on state modified by the other operations in the extent, the value will be the same regardless of when the operation executes relative to all of the other operations in the extent. In this case the compiler can represent the value with an opaque constant instead of attempting to derive a closed form expression. We call such a value an extent constant value, the expression that generated it an extent constant expression and the opaque constant an extent constant. Extent constants improve the analysis in several ways:

- They support operations that directly access global variables and instance variables of objects other than the receiver of the operation. The constraint is that such variables must be extent constants.

- They improve the efficiency of the compiler by supporting compact representations of expressions. These representations support efficient simplification and equality testing algorithms.

- They extend the range of constructs that the compiler can effectively analyze to include otherwise unanalyzable constructs that only access extent constants.

The compiler relaxes the notion of separability to allow the invocation section to compute extent constant values, even if the invocation section must access instance variables to do so.

3.5.2 Auxiliary Operations

For modularity purposes programmers often encapsulate the computation of values inside an operation. The caller obtains the computed values either as the return value of the operation or via local variables passed by reference into the operation. We call such operations *auxiliary* operations. Integrating such operations into their callers for analysis purposes can improve the effectiveness of the commutativity testing algorithm. The integration coarsens the granularity of the analysis, reducing the number of pairs that the algorithm tests for commutativity and increasing the ability of the compiler to recognize parallelizable computations. Because auxiliary operations are conceptually part of the operation that invokes them, the compiler relaxes the notion of separability to allow the object section to invoke auxiliary operations.

4 Analysis Algorithms

In this section we present analysis algorithms that a compiler can use to realize the basic approach outlined in Section 3. The algorithms use the type information to characterize how method invocations access data. This data access information is then used to identify extent constant variables, auxiliary operations and independent operations. The basic assumption behind this approach is that the program does not violate its type declarations.

For presentation purposes we assume that no operation returns a value (it can achieve the same functionality using reference parameters); the algorithms generalize to handle auxiliary operations with return values.

4.1 Overview

Given a method, the compiler first identifies the set of variables that the computation rooted at the method reads but does not write. This set is called the set of extent constant variables; the compiler uses this set to identify auxiliary operations and extent constant expressions. The compiler then performs a depth-first search of the call graph, identifying call sites that always invoke auxiliary operations and computing the extent of the method. The presented algorithms represent the extent as a set of methods. When possible the implemented compiler also extracts an expression for each parameter of a potentially invoked method that denotes that parameter's value. These expressions improve the precision of the commutativity testing phase — they increase the compiler's ability to prove that expressions involving the parameters denote identical values.

Once it has extracted the extent, the compiler verifies that all of the operations in the extent are separable and do not violate several reference parameter usage constraints. It also checks that none of the operations perform any input or output or create new objects. Finally, it tests that all pairs of operations in the extent commute. If so, it is possible to parallelize the original method. Figure 3 presents the algorithm; in the succeeding sections we discuss each of the routines that it uses to perform the analysis.

```
isParallel(m)
  ec = extentConstantVariables(m);
  ⟨ext, aux⟩ = extent(m, ec);
  if (!checkReferenceParameters(m, ext)) return false;
  ms = {m} ∪ map(method, ext);
  for all m1 ∈ ms
    if (!separable(m1, aux, ec)) return false;
    if (mayPerformIO(m1)) return false;
    if (mayCreateObject(m1)) return false;
  for all ⟨m1, m2⟩ ∈ ms × ms
    if (!commute(m1, m2, aux, ec)) return false;
  return true;
```

Figure 3: Algorithm to Recognize Parallel Methods

4.2 Basic Functionality

We first describe the basic functionality that provides the foundation for the analysis algorithms. The program defines a set of classes $cl \in CL$, primitive instance variables $v \in V$, nested object instance variables $n \in N$, local variables $l \in L$, methods $m \in M$, call sites $c \in C$, primitive types $t \in T$ and formal reference parameters $p \in P$. The compiler considers any parameter whose declared type is a pointer to a primitive type, an array of primitive types or a reference (in the C++ sense) to a primitive type to be a reference parameter. If a parameter's declared type is a pointer or reference to a class, it is not considered to be a reference parameter.

The analysis represents memory locations using storage descriptors $s \in S = P \cup L \cup T \cup CL \times V \cup CL \times Q \times V$, where $q \in Q = seq(N)$ is the set of nonempty sequences of nested object names. We write an element of Q in the form $n_1.n_2 \ldots n_i$, an element of $CL \times Q \times V$ in the form $cl.n_1.n_2 \ldots n_i.v$ and an element of $CL \times V$ in the form $cl.v$. `class` : $CL \times Q \to CL$ gives the class of a nested object. `type` : $S \to T$ gives the type of a storage descriptor. `lift` : $S \to T \cup CL \times V \cup CL \times Q \times V$ translates local variables and parameters to their primitive types. The definition is $\texttt{lift}(s) = \texttt{type}(s)$ when $s \in P \cup L$ and s otherwise.

There is a partial order $\preceq$ on S. Conceptually $s_1 \preceq s_2$ if the set of memory locations that s_1 represents is a subset of the set of memory locations that s_2 represents. $cl_1.v \preceq cl_2.v$ if cl_1 inherits from cl_2 or $cl_1 = cl_2$. $cl_1.q_1.v \preceq cl_2.v$ and $cl_1.q_1.q_2.v \preceq cl_2.q_2.v$ if $\texttt{class}(cl_1.q_1)$ inherits from cl_2 or $\texttt{class}(cl_1.q_1) = cl_2$. $s_1 \preceq s_2$ if $\texttt{type}(s_1) = s_2$.

Given a method, $\texttt{callsites} : M \to 2^C$ returns its set of call sites and $\texttt{referenceParameters} : M \to 2^P$ returns its set of formal reference parameters. Given a call site, $\texttt{method} : C \to M$ returns the invoked method. $\texttt{map}(f, A) = \{f(a).a \in A\}$.

$\mathcal{I}$ is the identity function on S. Bindings $b : B = P \to S$ represent bindings of formal reference parameters to storage descriptors. Given a binding b, the extension $\overline{b}$ of b to S is defined by $\overline{b}(s) = s$ when $s \notin P$ and $b(s)$ otherwise. Given a call site and a binding, $\texttt{bind} : C \times B \to B$ represents the binding of formal to actual parameters that takes place when the method at the call site is invoked in the context of the given binding.

The compiler performs some local analysis on each method to extract several functions that describe the way the method accesses memory. Given a method and a binding, $\texttt{read} : M \times B \to 2^S$ returns a set of storage descriptors that represent how the method reads data. For example, $\langle cl, v\rangle \in \texttt{read}(m, b)$ if the method m in the context of the binding b reads the instance variable v in an object of class cl. $p \in \texttt{read}(m, b)$ if m reads the reference parameter p in the context of the binding b. Similarly, $\texttt{write} : M \times B \to 2^S$ represents how the method writes data. $\texttt{dep} : C \to 2^S$ represents the memory locations that the surrounding method reads to compute the values in the reference parameters at the given call site.

In the remainder of this section we will use the code in Figure 4 as a running example to illustrate the application of the analysis algorithms. We have numbered the method invocation sites in this code to distinguish between different invocations of the same method. The example is a simplified version of the force computation phase of the Barnes-Hut algorithm described in Section 6.2.[2]

4.3 Extent Constant Variables

The `extentConstantVariables` routine in Figure 5 computes the set of extent constant variables for a given method. The `transitiveEffects` routine performs the core computation, using abstract interpretation to compute a read set and write set of storage descriptors that accurately represent how the computation may read and write data. `filter` prunes the read set so that it only contains storage descriptors that represent memory locations that the computation does not write.

[2]The example uses the type safe C++ dynamic cast construct. Because of front end limitations the current compiler does not handle this construct. The compiled code whose performance is described in Section 6.2.4 therefore uses the standard C cast construct.

```
extentConstantVariables(m)
  ⟨rd, wr⟩ = transitiveEffects(m);
  rd = map(lift, rd);
  wr = map(lift, wr);
  ⟨rd, wr⟩ = filter(rd, wr);
  return rd;

transitiveEffects(m)
  rd = ∅; wr = ∅;
  visited = ∅;
  current = {⟨m, I⟩};
  while (current ≠ ∅)
    next = ∅;
    for all ⟨m', b⟩ ∈ current
      for all c ∈ callsites(m')
        next = next ∪ {⟨method(c), bind(c, b)⟩};
      rd = rd ∪ read(m', b);
      wr = wr ∪ write(m', b);
    visited = visited ∪ current;
    current = next − visited;
  rd = rd − L;
  wr = wr − L;
  return ⟨rd, wr⟩;

filter(rd, wr)
  for all s ∈ rd
    for all s' ∈ wr
      if (s ⪯ s')
        rd = rd − {s};
        wr = wr ∪ {s};
      else if (s' ≺ s)
        rd = rd − {s};
  return ⟨rd, wr⟩;
```

Figure 5: Extent Constant Variables Algorithm

```
const int NDIM 3;
class vector {
   double val[NDIM];
public:
   void vecAdd(double v[NDIM]){
      for(int i=0; i < NDIM; i++) val[i] += v[i];
   }
};

class node {
public:
   double mass; // body's mass of combined cell/leaf mass
   vector pos; // body's position or cell/leaf aggregate center of mass
};

const int NSUB 8; // 2**NDIM for NDIM Dimension problems
class cell : public node {
public:
   node *subp[NSUB];
};
const int LEAFMAXBODIES 16;
class leaf : public node {
public:
   int numbodies;
   body *bodyp[LEAFMAXBODIES];
};
class body : public node {
   vector vel; // velocity
   vector acc; // acceleration
   double phi; // interaction potential
public:
   double subdivp(node *p, double dsq);
   void gravsub(node *n);
   double computeInter(node *n, double *res);
   void openCell(node *n, double dsq);
   void openLeaf(node *n);
   void walksub(node *n, double dsq);
};
class nbody {
public:
   int numbodies; // total number of bodies in simulation
   body **bodies; // set of bodies in the simulation
   node *BH_root; // root of the Barnes-Hut tree
   double size; // Space bounding box maximum side size
   void computeForces();
};
class parms {
public:
   double tolSq; // square of numeric tolerance
   double eps; // epsilon
   double epsSq; // epsilon square
};
// Global Variables
parms Parms;
nbody Nbody;

double body::subdivp(node *n, double dsq){
   double drsq, d;
   drsq = Parms.epsSq;
   for(int i=0; i < NDIM; i++){
      d = n->pos.val[i] - pos.val[i];
      drsq += d * d;
   }
   return ((Parms.tolSq *drsq) < dsq);
}
```

```
double body::computeInter(node *n, double *res){
   double inc, r, drsq, d;
   drsq = Parms.eps;
   for(int i=0; i < NDIM; i++){
      d = n->pos.val[i] - pos.val[i];
      drsq += d * d;
   }
   inc = n->mass / sqrt(drsq);
   r = inc / drsq;
   for (int i=0; i < NDIM; i++)
      res[i] *= r;
   return inc;
}

void body::gravsub(node *n){
   double d;
   double tmpv[NDIM];
1: d = this->computeInter(n,tmpv);
   phi -= d;
2: acc.vecAdd(tmpv);
}

void body::openCell(cell *c, double dsq){
   node *n;
   for(int i=0; i < NSUB; i++){
      n = c->subp[i];
      if(n != NULL)
3:       this->walksub(n,(dsq/4.0));
   }
}

void body::openLeaf(leaf *l){
   body *b;
   for(int i=0; i < l->numbodies; i++){
      b = l->bodyp[i];
      if(b != this)
4:       this->gravsub(b);
   }
}

void body::walksub(node *n, double dsq){
   cell *c;
   leaf *l;
5: if(this->subdivp(n,dsq)){
      c = dynamic_cast<cell*>n;
      if(c != NULL){
6:       this->openCell(c,dsq);
      } else {
         l = dynamic_cast<leaf*>n;
         if(l != NULL)
7:          this->openLeaf(l);
      }
   } else {
8:    this->gravsub(n);
   }
}
void nbody::computeForces(){
   body *b;
   for(int i=0; i < numbodies; i++){
      b = bodies[i];
9:    b->walksub(BH_root,size*size);
   }
}
```

Figure 4: Simplified Barnes-Hut Force Computation code

Figure 6 presents the extracted `read`, `write` and `dep` functions for the methods in the example. Figure 7 presents the results of the extent constant variables computation for all of the methods in the example. We show the intermediate results of the computation for the method body::gravsub.

Method: vector::vecAdd
`read`(vector::vecAdd, b) = {this.val, b(v)}
`write`(vector::vecAdd, b) = {this.val}

Method: body::computeInter
`read`(body::computeInter, b) =
{node.mass,node.pos.val,parms.eps}
`write`(body::computeInter, b) = {b(res)}

Method: body::subdivp
`read`(body::subdivp, b) =
{node.pos.val,parms.epsSq,parms.tolSq}
`write`(body::subdivp, b) = $\emptyset$

Method: body::gravsub
`read`(body::gravsub, b) = {body.phi}
`write`(body::gravsub, b) = {body.phi}
`dep`(1) = $\emptyset$
`dep`(2) = {node.mass,node.pos.val,parms.eps}

Method: body::openCell
`read`(body::openCell, b) = {cell.subp}
`write`(body::openCell, b) = $\emptyset$
`dep`(3) = {cell.subp}

Method: body::openLeaf
`read`(body::openLeaf, b) = {leaf.numbodies,leaf.bodyp}
`write`(body::openLeaf, b) = $\emptyset$
`dep`(4) = {leaf.numbodies,leaf.bodyp}

Method: body::walksub
`read`(body::walksub, b) = $\emptyset$
`write`(body::walksub, b) = $\emptyset$
`dep`(5) = $\emptyset$
`dep`(6) = {node.pos.val,parms.epsSq,parms.tolSq}
`dep`(7) = {node.pos.val,parms.epsSq,parms.tolSq}
`dep`(8) = {node.pos.val,parms.epsSq,parms.tolSq}

Method: nbody::computeForces
`read`(nbody::computeForces, b) =
{nbody.numbodies,nbody.bodies}
`write`(nbody::computeForces, b) = $\emptyset$
`dep`(9) = {nbody.numbodies,nbody.bodies,
nbody.BH_root,nbody.size}

Figure 6: `read`, `write` and `dep` Functions for Barnes-Hut

`transitiveEffects`(body::computeInter) = $\langle rd_0, wr_0\rangle$ where
rd_0 = {node.mass,node.pos.val,parms.eps}
$wr_0 = \emptyset$

`transitiveEffects`(body::gravsub) = $\langle rd_1, wr_1\rangle$ where
rd_1 = {node.mass,node.pos.val,body.phi,body.acc.val,
parms.eps}
wr_1 = {body.phi,body.acc.val}
rd_2 = `map(lift,` rd_1`)` = {node.mass,node.pos.val,
body.phi,body.acc.val,parms.eps}
wr_2 = `map(lift,` wr_1`)` = {body.phi,body.acc.val}
`filter`(rd_2, wr_2) = $\langle rd_3, wr_3\rangle$ where
rd_3 = {node.mass,node.pos.val,parms.eps}
wr_3 = {body.phi,body.acc.val}
`extentConstantVariables`(body::gravsub) = rd_3 where
rd_3 = {node.mass,node.pos.val,parms.eps}

`transitiveEffects`(body::openLeaf) = $\langle rd_4, wr_4\rangle$ where
rd_4 = {node.mass,node.pos.val,body.phi,body.acc.val,
parms.eps,leaf.numbodies,leaf.bodyp}
wr_4 = {body.phi,body.acc.val}
`extentConstantVariables`(body::openLeaf) =
{node.mass,node.pos.val,parms.eps,leaf.numbodies,
leaf.bodyp}

`transitiveEffects`(body::openCell) = $\langle rd_5, wr_5\rangle$ where
rd_5 = {node.mass,node.pos.val,body.phi,body.acc.val,
leaf.numbodies,leaf.bodyp,cell.subp,
parms.eps,parms.epsSq,parms.tolSq}
wr_5 = {body.phi,body.acc.val}
`extentConstantVariables`(body::openCell) =
{node.mass,node.pos.val,leaf.numbodies,
leaf.bodyp,cell.subp,parms.eps,parms.epsSq,parms.tolSq}

`transitiveEffects`(body::walksub) = $\langle rd_6, wr_6\rangle$ where
rd_6 = {node.mass,node.pos.val,body.phi,body.acc.val,
leaf.numbodies,leaf.bodyp,cell.subp,
parms.eps,parms.epsSq,parms.tolSq}
wr_6 = {body.phi,body.acc.val}
`extentConstantVariables`(body::walksub) =
{node.mass,node.pos.val,leaf.numbodies,leaf.bodyp,
cell.subp,parms.eps,parms.epsSq,parms.tolSq}

`transitiveEffects`(nbody::computeForces) = $\langle rd_7, wr_7\rangle$
where
rd_7 = {node.mass,node.pos.val,body.phi,body.acc.val,
leaf.numbodies,leaf.bodyp,cell.subp,
parms.eps,parms.epsSq,parms.tolSq,
nbody.numbodies,nbody.bodies,nbody.BH_root,nbody.size}
wr_7 = {body.phi,body.acc.val}
`extentConstantVariables`(nbody::computeForces) =
{node.mass,node.pos.val,leaf.numbodies,
leaf.bodyp,cell.subp,parms.eps,parms.epsSq,parms.tolSq,
nbody.numbodies,nbody.bodies,nbody.BH_root,nbody.size}

Figure 7: Extent Constant Variables Computation

4.4 Extents and Auxiliary Operations

Given a method and a set of extent constant variables, the algorithm in Figure 8 computes the set of call sites that always invoke auxiliary operations; each such call site is called an auxiliary call site. The algorithm also computes the extent of the method. Note that auxiliary operations are not included in the extent.

```
global visited = ∅;
global ext = ∅;
global aux = ∅;
extent(m, ec)
  visited = ∅;
  ext = ∅;
  aux = ∅;
  compute_ext_aux(m, ec);
  return ⟨ext, aux⟩;

compute_ext_aux(m, ec)
  if (m ∉ visited)
    visited = visited ∪ {m};
    for all c ∈ callsites(m)
      ⟨rd, wr⟩ = transitiveEffects(method(c));
      rd = map(bind(c, I), rd);
      wr = map(bind(c, I), wr);
      d = dep(c);
      if (wr ⊆ L && rd ⊆ ec ∪ L && d ⊆ ec)
        aux = aux ∪ {c};
      else
        ext = ext ∪ {c};
        compute_ext_aux(method(c), ec);
```

Figure 8: Extent Algorithm

The algorithm performs a depth first search of the call graph, terminating search paths when it encounters a call site that always invokes an auxiliary operation. The need to accurately represent the values that auxiliary operations write into reference parameters partly determines the conditions that they must satisfy. The current condition is that auxiliary operations only compute extent constant values, which allows the compiler to represent the values in reference parameters using extent constants. An interprocedural symbolic execution algorithm would relax this condition by allowing the compiler to extract a more precise representation for the values computed in auxiliary operations. The compiler also checks that auxiliary operations write all of their return values into local variables of the caller. The compiler can therefore omit auxiliary operations from the commutativity testing phase.

We now illustrate the applicaton of the extent computation algorithm in Figure 8 as it computes the extent of nbody::computeForces. This computation uses the set of extent constant variables from Figure 7.

4.5 Reference Parameter Checks

The compiler performs several checks to ensure that the symbolic execution operates correctly. To preserve the property that reference parameters always hold extent constant values, the compiler enforces the constraint that none of the operations in the extent write their reference parameters. To help ensure that the symbolic execution builds expressions that correctly denote the new values of instance variables, it also enforces the constraint that none of the methods are invoked with a reference parameter that points into the receiver. The current policy requires that all reference parameters be local variables of primitive types. Figure 10 presents the algorithm that checks these two restrictions.

```
extentConstantVariables(nbody::computeForces) =
    ec1 = {node.mass,node.pos.val,leaf.numbodies,
    leaf.bodyp,cell.subp,parms.eps,parms.epsSq,parms.tolSq,
    nbody.numbodies,nbody.bodies,nbody.BH_root,nbody.size}
extent(body::gravsub, ec1) = ⟨aux1, ext1⟩ where
  aux1 = {1}
  ext1 = {2}

extent(body::openCell, ec1) = ⟨aux2, ext2⟩ where
  aux2 = {1, 5}
  ext2 = {2, 3, 4, 6, 7, 8}

extent(body::openLeaf, ec1) = ⟨aux3, ext3⟩ where
  aux3 = {1}
  ext3 = {2, 4}

extent(body::walksub, ec1) = ⟨aux4, ext4⟩ where
  aux4 = {1, 5}
  ext4 = {2, 3, 4, 6, 7, 8}

extent(nbody::computeForces, ec1) = ⟨aux5, ext5⟩ where
  aux5 = {1, 5}
  ext5 = {2, 3, 4, 6, 7, 8, 9}
```

Figure 9: Extent Computation for body::gravsub and nbody::computeForces

For example, checkReferenceParameters(nbody::computeForces, ext_5) (where ext_5 is the ext_5 from Figure 9) returns true because nbody::computeForces has no reference parameters and, at all of the call sites in its extent, the call site's reference parameters are local variables of the enclosing method.

```
checkReferenceParameters(m, ext)
  if (referenceParameters(m) ≠ ∅) return false;
  for all c ∈ ext
    m' = method(c);
    for all s ∈ map(bind(c, I), referenceParameters(m'))
      if (isBaseLocal(s)) return false;
    ⟨rd, wr⟩ = transitiveEffects(m');
    wr = map(bind(c, I), wr);
    if (wr ⊈ CL × V) return false;
  return true;

isBaseLocal(s)
  return !(s ∈ L && type(s) ∈ T);
```

Figure 10: Check Reference Parameters Algorithm

4.6 Separability

The compiler must also check that each operation in the extent is separable. It therefore scans each method to make sure that it never accesses an instance variable after it executes a call site that may invoke an operation in the extent. As part of the separability test it also makes sure that the method only writes local variables or instance variables of the receiver and only reads parameters, local variables, instance variables of the receiver or extent constant variables. The separability of a method invocation may depend on the set of auxiliary call sites and the set of extent constant variables. The separability testing routine, $\texttt{separable}(m, aux, ec)$, therefore takes these two sets as parameters.

4.7 Commutativity Testing

The commutativity testing algorithm presented in Figure 11 determines if all invocations of two methods in the context of a given set of auxiliary call sites and extent constant variables commute. The algorithm first checks to see if the invocations are always independent. The check can be performed using the type system — if the classes of the receivers are different, the two methods are guaranteed to be independent.[3] The compiler can also analyze the instance variable usage to check that neither method writes an instance variable that the other accesses. We generalize the notion of independence to methods in the context of a given set of auxiliary call sites and extent constant variables by defining that two methods are independent if all invocations of the first method are independent of all invocations of the second method.

The algorithm next checks if it can symbolically execute each of the methods. If not, it conservatively assumes that invocations of the two methods may not commute. If it can symbolically execute the methods, it does so, then simplifies the resulting expressions and compares corresponding expressions for equality. If all of the expressions denote the same value, the operations commute.

The compiler has two routines that deal with the symbolic execution: $\texttt{analyzable}(m, aux, ec)$, which determines if it is possible to symbolically execute a method, and $\texttt{symbolicallyExecute}(m_1, m_2, aux, ec)$, which actually performs the execution. The result of the symbolic execution is a pair $\langle i, n\rangle$, where $i(v)$ is the expression denoting the new value of the instance variable v and n is a multiset of MX expressions denoting the multiset of directly invoked operations. Because the symbolic execution depends on the set of auxiliary call sites and the set of extent constant variables, both routines take these two sets as parameters.

Consider the parallelization of nbody::computeForces. The compiler must check that all pairs of operations in its extent commute. Because nbody::computeForces, body::walksub, body::openCell, and body::openLeaf only compute extent constant values and write local variables, they commute with all methods in the extent. Because vector does not inherit from body and vice-versa, body::gravsub and vector::vecAdd can never be invoked with the same receiver object. body::gravsub and vector::vecAdd are therefore independent.

The compiler is left to check that all pairs of invocations of body::gravsub commute and that all pairs of invocations of vector::vecAdd commute. The compiler uses symbolic analysis to check that these pairs commute.

[3] The two methods are independent even if one of the classes inherits from the other. Recall that the model of computation imposes the constraint that a method cannot access an instance variables declared in a class from which its receiver's class inherits.

```
commute(m1, m2, aux, ec)
  if (independent(m1, m2)) return true;
  if (!analyzable(m1, aux, ec)) return false;
  if (!analyzable(m2, aux, ec)) return false;
  ⟨i1, n1⟩ = symbolicallyExecute(m1, m2, aux, ec);
  ⟨i2, n2⟩ = symbolicallyExecute(m2, m1, aux, ec);
  for all v ∈ instanceVariables(receiverClass(m1))
    if (!compare(simplify(i1(v)), simplify(i2(v))))
      return false;
  if (!compare(simplify(n1), simplify(n2)))
    return false;
  return true;
```

Figure 11: Commutativity Testing Algorithm

4.8 Symbolic Analysis

To test that method invocations commute, the compiler represents and reasons about the new values of the receiver's instance variables and the multiset of operations directly invoked when the two methods execute. The compiler represents the new values and multisets of invoked methods using symbolic expressions. Figure 12 presents the symbolic expressions that the compiler uses. These expressions include standard arithmetic and logical expressions, conditional expressions and expressions that represent values computed in simple `for` loops.

The compiler uses extent constants $e \in E$ to represent values computed in auxiliary operations. $\oplus$ represents an arbitrary binary operator; $\ominus$ represents an arbitrary unary operator. The compiler uses EX expressions to represent instance variable values and multisets of MX expressions to represent invoked methods.

$$
\begin{array}{lcl}
\texttt{ex} \in \texttt{EX} & ::= & \texttt{EX}\oplus\texttt{EX} \mid \ominus\texttt{EX} \mid \texttt{if(EX, EX, EX)} \mid v \mid e \\
\texttt{mx} \in \texttt{MX} & ::= & \texttt{EX->op(EX)} \mid \texttt{if(EX, MX)} \mid \\
 & & \texttt{for(}l\texttt{=EX;}l\texttt{<EX;}l\texttt{+= EX)MX}
\end{array}
$$

Figure 12: Expressions for Symbolic Analysis

4.8.1 Symbolic Execution

The symbolic execution algorithm can operate successfully only on a subset of the constructs in the language. To execute an assignment statement, the algorithm 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. It executes conditional statements by symbolically executing the two branches, then using conditional expressions to combine the results. It executes auxiliary operations by internally generating a new extent constant for each reference parameter, then binding each reference parameter to its constant.

The symbolic execution does not handle loops in a general way. If the loop is in the following form, where $\mathtt{ex}_1$ is the upper bound of the array v and $\mathtt{ex}_2$ is an extent constant expression, the algorithm can represent the new value of v.

$$\begin{array}{l}\mathtt{for}(l = 0; l < \mathtt{ex}_1; l++)\\ \quad v[l] = v[l] \oplus \mathtt{ex}_2;\end{array}$$

If the loop is in the following form, where $\mathtt{ex}_1, \ldots, \mathtt{ex}_n$ are all extent constant expressions, the algorithm can represent the invoked set of methods.

$$\begin{array}{l}\mathtt{for}(l = \mathtt{ex}_1; l < \mathtt{ex}_2; l += \mathtt{ex}_3)\\ \quad \mathtt{ex}_4\mathtt{->op}(\mathtt{ex}_5, \ldots, \mathtt{ex}_n);\end{array}$$

The algorithm cannot currently represent expressions computed in loops that are not in one of these two forms. We expect to enhance the algorithm to recognize a wider range of loops. For analysis purposes the compiler can also replace unanalyzable loops with tail recursive methods that perform the same computation.

In our example the compiler symbolically executes two pairs of methods: ⟨body::gravsub,body::gravsub⟩ and ⟨vector:vecAdd,vector:vecAdd⟩. The symbolic execution starts by generating one variable to represent the receiver and different variables to represent the parameter values of the two invoked methods. the two invoked methods.[4] Figure 13 presents the results of the symbolic execution of the pair ⟨body::gravsub,body::gravsub⟩. In this example the compiler generated the variables n_1 and n_2 to represent the potentially distinct parameter values of the invocations of body::gravsub. The variable $this$ represents the receiver. The compiler assumes both invocations have the same receiver; if they have different receivers they are independent and therefore trivially commute. The symbolic execution also uses the internally generated extent constants $const_1, const_2, const_3$ and $const_4$.

Figure 14 presents the results of the symbolic execution of the pair ⟨vector::vecAdd,vector::vecAdd⟩. In this example the compiler generated the variables n_1 and n_2 to represent the potentially distinct parameter values of the invocations of vector::vecAdd. The variable $this$ represents the receiver. The compiler assumes both invocations have the same receiver; if they have different receivers they are independent and therefore trivially commute. Because the method vector::vecAdd uses a vector loop update the compiler is able to represent the loop effects on the instance variable `val` by a scalar closed form expression. The symbolic execution also uses the internally generated extent constants $const_1, const_2$.

[4] The implemented compiler maintains information about the parameter values of invoked methods. If a parameter of a given method always has the same value, the compiler uses the same variable to represent the parameter's value in all symbolic executions of that method.

$$\begin{array}{l}
ec = \mathtt{extentConstantVariables}(\text{nbody::computeForces})\\
\mathtt{extent}(\text{nbody::computeForces}, ec) = \langle aux, ext\rangle \text{ where}\\
\quad aux = \{1, 5\}\\
\quad ext = \{2, 3, 4, 6, 7, 8, 9\}\\
\\
m_1 = this\text{->gravsub}(n_1)\\
m_2 = this\text{->gravsub}(n_2)\\
\\
\langle i_1, n_1\rangle = \mathtt{symbolicallyExecute}(m_1, m_2, aux, ec) \text{ where}\\
\quad i_1 = \{\text{phi} \mapsto (\text{phi} - const_1) - const_2\}\\
\quad n_1 = \{\text{acc.vecAdd}(const_3), \text{acc.vecAdd}(const_4)\}\\
\langle i_2, n_2\rangle = \mathtt{symbolicallyExecute}(m_2, m_1, aux, ec) \text{ where}\\
\quad i_2 = \{\text{phi} \mapsto (\text{phi} - const_2) - const_1\}\\
\quad n_2 = \{\text{acc.vecAdd}(const_4), \text{acc.vecAdd}(const_3)\}
\end{array}$$

Figure 13: Symbolic Execution of Two Invocations of body::gravsub

$$\begin{array}{l}
ec = \mathtt{extentConstantVariables}(\text{nbody::computeForces})\\
\mathtt{extent}(\text{nbody::computeForces}, ec) = \langle aux, ext\rangle \text{ where}\\
\quad aux = \{1, 5\}\\
\quad ext = \{2, 3, 4, 6, 7, 8, 9\}\\
\\
m_1 = this\text{->vecAdd}(n_1)\\
m_2 = this\text{->vecAdd}(n_2)\\
\\
\langle i_1, n_1\rangle = \mathtt{symbolicallyExecute}(m_1, m_2, aux, ec) \text{ where}\\
\quad i_1 = \{\text{val} \mapsto (\text{val} + const_1) + const_2\}\\
\quad n_1 = \emptyset\\
\langle i_2, n_2\rangle = \mathtt{symbolicallyExecute}(m_2, m_1, aux, ec) \text{ where}\\
\quad i_2 = \{\text{val} \mapsto (\text{val} + const_2) + const_1\}\\
\quad n_2 = \emptyset
\end{array}$$

Figure 14: Symbolic Execution of Two Invocations of vector::vecAdd

4.8.2 Expression Simplification and Comparison

The expression simplifier is organized as a set of rewrite rules designed to reduce expressions to a simplified form for comparison. The comparison itself consists of a simple isomorphism test.

The compiler currently applies simple arithmetic rewrite rules such as $\mathtt{ex}_1 - \mathtt{ex}_2 \Rightarrow \mathtt{ex}_1 + (-\mathtt{ex}_2)$, $\mathtt{--ex} \Rightarrow \mathtt{ex}$ and $\mathtt{ex}_1 \times (\mathtt{ex}_2 + \mathtt{ex}_3) \Rightarrow \mathtt{ex}_1 \times \mathtt{ex}_2 + \mathtt{ex}_1 \times \mathtt{ex}_3$. The simplifier also applies rules such as $((\mathtt{ex}_1{+}\mathtt{ex}_2){+}\mathtt{ex}_3) \Rightarrow (\mathtt{ex}_1{+}\mathtt{ex}_2{+}\mathtt{ex}_3)$ that convert binary applications of commutative and associative operators to n-ary applications. It then sorts the operands according to an arbitrary order on expressions. This sort facilitates the eventual expression comparison by making it easier to identify isomorphic subexpressions. We have also developed rules for conditional and array expressions [33].

In the worst case the expression manipulation algorithms may take exponential running time. Like other researchers

$m_1 = this$->gravsub(n_1)
$m_2 = this$->gravsub(n_2)

$\langle i_1, n_1\rangle$ = `symbolicallyExecute`(m_1, m_2, aux, ec) where
$i_1 = \{$phi$\mapsto$ (phi$-const_1) - const_2\}$
$n_1 = \{$acc.vecAdd$(const_3)$,acc.vecAdd$(const_4)\}$

Application of rule: $\texttt{ex}_1 - \texttt{ex}_2 \Rightarrow \texttt{ex}_1 + (-\texttt{ex}_2)$
$i_1 = \{$phi$\mapsto$ (phi$+(-const_2)) + (-const_1)\}$
$n_1 = \{$acc.vecAdd$(const_3)$,acc.vecAdd$(const_4)\}$
Conversion into n-ary application of + operator:
$i_1 = \{$phi$\mapsto$phi$+(-const_2) + (-const_1)\}$
$n_1 = \{$acc.vecAdd$(const_3)$,acc.vecAdd$(const_4)\}$
Sorting operands of + operator:
$i_1 = \{$phi$\mapsto$phi$+(-const_1) + (-const_2)\}$
$n_1 = \{$acc.vecAdd$(const_3)$,acc.vecAdd$(const_4)\}$

$\langle i_2, n_2\rangle$ = `symbolicallyExecute`(m_2, m_1, aux, ec) where
$\langle i_2, n_2\rangle$ = `symbolicallyExecute`(m_2, m_1, aux, ec) where
$i_2 = \{$phi$\mapsto$ (phi$-const_2) - const_1\}$
$n_2 = \{$acc.vecAdd$(const_4)$,acc.vecAdd$(const_3)\}$

Application of rule: $\texttt{ex}_1 - \texttt{ex}_2 \Rightarrow \texttt{ex}_1 + (-\texttt{ex}_2)$
$i_2 = \{$phi$\mapsto$ (phi$+(-const_1)) + (-const_2)\}$
$n_2 = \{$acc.vecAdd$(const_4)$,acc.vecAdd$(const_3)\}$
Conversion into n-ary application of + operator:
$i_2 = \{$phi$\mapsto$phi$+(-const_1) + (-const_2)\}$
$n_2 = \{$acc.vecAdd$(const_4)$,acc.vecAdd$(const_3)\}$
Sorting operands of + operator:
$i_2 = \{$phi$\mapsto$phi$+(-const_1) + (-const_2)\}$
$n_2 = \{$acc.vecAdd$(const_3)$,acc.vecAdd$(const_4)\}$

Figure 15: Expression Simplification for Invocations of body::gravsub

applying similar expression manipulation techniques in other analysis contexts [6], we have not observed this behavior in practice. Finally, it is undecidable in general to determine if two expressions always denote the same value [19]. We therefore focus on developing algorithms that work well for the cases that occur in practice.

Figure 13 presents the results of the expression simplification algorithm for the expressions generating during the symbolic execution of the two invocations of body::gravsub. Figure 16 presents the corresponding results for the two invocations of vector::vecAdd. After the expression simplification, the compiler compares corresponding expressions for equality. In these two cases the expressions denoting the new values of the instance variables and the multisets of invoked operations are equivalent. The compiler has determined that all of the operations in the computation rooted at nbody::computeForces commute. It therefore marks nbody::computeForces as a parallel method.

4.9 Complexity

In this section we briefly discuss the complexity of the analysis algorithms.

$m_1 = this$->vecAdd(n_1)
$m_2 = this$->vecAdd(n_2)

$\langle i_1, n_1\rangle$ = `symbolicallyExecute`(m_1, m_2, aux, ec) where
$i_1 = \{$val$\mapsto$ (val$+const_1) + const_2\}$
$n_1 = \emptyset$
Convertion into n-ary application of + operator:
$i_1 = \{$val$\mapsto$val$+const_1 + const_2\}$
$n_1 = \emptyset$
Sorting operands of + operator:
$i_1 = \{$val$\mapsto$val$+const_1 + const_2\}$
$n_1 = \emptyset$

$\langle i_2, n_2\rangle$ = `symbolicallyExecute`(m_2, m_1, aux, ec) where
$i_2 = \{$val$\mapsto$ (val$+const_2) + const_1\}$
$n_2 = \emptyset$
Conversion into n-ary application of + operator:
$i_2 = \{$val$\mapsto$val$+const_2 + const_1\}$
$n_2 = \emptyset$
Sorting operands of + operator:
$i_2 = \{$val$\mapsto$val$+const_1 + const_2\}$
$n_2 = \emptyset$

Figure 16: Simplification of the Instance Variable Bindings From Two Invocations of the Method vector::vecAdd

4.9.1 Extent Constant Computation

To compute the `read` and `write` functions, the compiler linearly scans all of the methods in the program. For each method the compiler simply records which instance variables and reference parameters the method reads and writes. For each method the running time of this step is proportional to the length of the method.

The complexity of the `transitiveEffects` computation is determined by the number of distinct pairs of call sites and bindings of formal reference parameters to storage descriptors that the compiler generates during the abstract interpretation. $cs \times |S|^{rp}$ is an upper bound on this number of pairs, where cs is the number of call sites in the program and rp is the maximum number of reference parameters of any method. In our experience the analysis generates very few bindings for each call site.

One way to perform the `dep` analysis is to perform a reaching definition analysis for all of the reference parameters at each call site in each method. Given the restricted set of flow of control constructs that the current compiler supports, it is possible to implement a simplified algorithm that computes the `dep` function with a single pass over the method. This pass uses the information computed in the `transitiveEffects` phase.

4.9.2 Extent Computation

The complexity of the extent computation is determined by the number of call sites that it visits. In the worst case the program may visit each call site in the program once.

4.9.3 Commutativity Analysis

To check if a given method is parallel the algorithm in Figure 3 tests that all operations in the extent of the method commute. The given algorithm represents the extent using a set of methods and performs $O(n^2)$ commutativity tests, where n is the number of methods in the extent.

As mentioned above in Section 4.8.2, the expression simplification algorithms may generate expressions that grow exponentially with respect to the number of terms in original expression.

5 Code Generation

If the analysis marks a method as parallel, the compiler generates two versions of the method: a serial version and a parallel version. Methods marked as serial invoke the serial version of any parallel method that they invoke. The generated code for the serial version of the parallel method simply invokes the parallel version, then blocks until the generated parallel computation terminates. It then returns back to the caller.

To generate code for the parallel version, the compiler first generates the object section of the method. The generated code acquires the mutual exclusion lock in the receiver when it enters the object section, then releases the lock when it exits. The generated code for the invocation section invokes the parallel version of each invoked method, using the `spawn` construct to execute the operation in parallel. Auxiliary operations are an exception to this code generation policy; they execute serially with respect to the caller.

5.1 Parallel Loops

The compiler applies an optimization that exposes parallel loops to the run-time system. If a `for` loop contains nothing but invocations of parallel versions of methods, the compiler generates parallel loop code instead of code that serially spawns each invoked operation. The generated code can then apply standard parallel loop execution techniques; it currently uses guided self-scheduling [29].

5.2 Suppressing Excess Concurrency

In practice parallel execution inevitably generates overhead in the form of synchronization and task management overhead. If the compiler exploits too much concurrency, the resulting overhead may overwhelm the performance benefits of parallel execution. The compiler uses a heuristic that attempts to suppress the exploitation of unprofitable concurrency; this heuristic suppresses the exploitation of nested concurrency within parallel loops.

To apply the heuristic, the compiler generates a third version of each parallel method, the mutex version. Like the parallel version, the mutex version uses the mutual exclusion lock in the receiver to make the object section execute atomically. But the generated invocation section serially invokes the mutex versions of all invoked methods. Any computation that starts with the execution of a mutex version therefore executes serially. The inserted synchronization constructs allow the mutex versions of methods to safely execute concurrently with parallel versions. The generated code for parallel loops invokes the mutex versions of methods rather than the parallel versions. Each iteration of the loop therefore executes serially.

The heuristic trades off parallelism for a reduction in the concurrency exploitation overhead. While it works well for our current application set, in some cases it may generate excessively sequential code. In the future we expect to tune the heuristic and explore efficient mechanisms for exploiting the concurrency in nested parallel loops.

5.3 Local Variable Lifetimes

The compiler must ensure that the lifetime of an operation's activation record exceeds the lifetimes of all operations that may access the activation record. The compiler currently uses a conservative strategy: if a method may pass a local variable by reference to an operation or create a pointer to a local variable, the compiler serializes the computation rooted at that method. At auxiliary call sites the generated code invokes the original version of the invoked method; at other call sites it invokes the mutex version. This code generation strategy also ensures that operations observe the correct values of local variables written by multiple auxiliary operations.

5.4 Lock Optimizations

Lock constructs are a significant potential source of overhead. The code generator therefore applies several optimizations designed to reduce the lock overhead.

5.4.1 Lock Elimination

If an operation's object section only computes extent constant values, the compiler generates no lock constructs — the operation executes atomically even without synchronization. If none of the parallel operations with receivers from a given class uses the mutual exclusion lock, the compiler omits the lock from the class declaration.

5.4.2 Lock Hoisting

The lock hoisting optimization eliminates lock constructs associated with nested objects and coarsens the lock granularity by generating a single lock acquire/release pair for multiple operations that access the same object. The optimization is based on the following observation:

> Assume that all operations with receiver r in a given extent acquire r's lock and transitively only invoke operations that have either r or nested objects of r as a receiver. Also assume that every operation in the extent whose receiver may be a nested object

of r is directly or indirectly invoked by an operation whose receiver is r. In this case the compiler may eliminate locking constructs as follows. Each operation in the extent with receiver r executes a customized version of its method that holds r's lock for both the object and invocation sections. In its invocation section, this customized version invokes the original version (from the original serial program) of each invoked method. This original version executes serially with no lock operations.

In effect, the generated code uses the lock on the enclosing object to ensure the atomicity of operations that access nested objects. It also generates only one lock acquire/release pair for computations rooted at outermost operations that meet the condition in the observation.

Lock hoisting trades off concurrency for a reduction in the lock overhead. In some cases the optimization may generate code that performs poorly because of excessive serialization. While we may refine the optimization in the future, it works well for our current set of applications.

6 Experimental Results

We have developed a prototype compiler based on the analysis algorithms in Section 4. It uses an enhanced version of the code generation algorithms in Section 5. We have used this compiler to automatically parallelize two applications: the Barnes-Hut hierarchical N-body solver [3] and the Water [37] code.[5] Explicitly parallel versions of the applications are available in the SPLASH [37] and SPLASH-2 [42] benchmark suites. This section presents performance results for both the automatically parallelized and explicitly parallel versions on a 32 processor Stanford DASH machine [25] running a modified version of the IRIX 5.2 operating system. The programs were compiled using the IRIX 5.3 CC compiler at the -O2 optimization level.

6.1 The Compilation System

The compiler is structured as a source-to-source translator that takes a serial program written in a subset of C++ and generates an explicitly parallel C++ program that performs the same computation. We use Sage++ [7] as a front end. The analysis phase consists of approximately 14,000 lines of C++ code, with approximately 1,800 devoted to interfacing with Sage++. The generated code contains calls to a runtime library that provides the basic concurrency management and synchronization functionality. The library consists of approximately 5,000 lines of C code.

The current version of the compiler imposes several restrictions on the dialect of C++ that it accepts. The goal of these restrictions is to simplify the implementation of the prototype while providing enough expressive power to allow the programmer to develop clean object-based programs. The major restrictions include:

- The program has no virtual methods and does not use operator or method overloading. The compiler imposes this restriction to simplify the extent computation.
- The program uses neither multiple inheritance nor templates.
- The program contains no typedef, union, struct or enum types.
- Global variables cannot be primitive data types; they must be class types.
- The program does not use pointers to members or static members.
- The program contains no casts between base types such as `int`, `float` and `double` that are used to represent numbers. The program may contain casts between pointer types; the compiler assumes that the casts do not cause the program to violate its type declarations.
- The program contains no default arguments or methods with variable numbers of arguments.
- No operation accesses an instance variable of a nested object of the receiver or an instance variable declared in a class from which the receiver's class inherits.

In addition to these restrictions the compiler assumes that the program has been type checked and does not violate its type declarations.

6.2 Barnes-Hut

Barnes-Hut is representative of our target class of applications. It performs well in part because it employs a sophisticated pointer-based data structure: a space subdivision tree that dramatically improves the efficiency of a key phase in the algorithm. Although it is considered to be an important, widely studied computation, all previously existing parallel versions were parallelized by hand using low-level, explicitly parallel programming systems [34, 36]. We are aware of no other compiler that is capable of automatically parallelizing this computation.

The space subdivision tree organizes the data as follows. The bodies are stored at the leaves of the tree; each internal node represents the center of mass of all bodies below that node in the tree. Each iteration of the computation first constructs a new space subdivision tree for the current positions of the bodies. It then computes the center of mass for all of the internal nodes in the new tree. The force computation phase executes next; this phase uses the space subdivision tree to compute the total force acting on each body. The final phase uses the computed forces to update the positions of the bodies.

[5] The sequential source codes and automatically generated parallel codes can be found at `http://www.cs.ucsb.edu/~pedro/CA/apps`.

6.2.1 The Serial C++ Code

We obtained serial C++ code for this computation by acquiring the explicitly parallel C version from the SPLASH-2 benchmark set, then removing the parallel constructs to obtain a serial version written in C. We then translated the serial C version into C++. The goal of the translation process was to obtain a clean object-based program that conformed to the model of computation presented in Section 3.

As part of the translation we eliminated several computations that dealt with parallel execution. For example, the parallel version used costzones partitioning to schedule the force computation phase [36]; the serial version eliminated the costzones code and the associated data structures. We also split a loop in the force computation phase into three loops. This transformation exposed the concurrency in force computation phase, enabling the compiler to recognize that two of the resulting three loops could execute in parallel. As part of this transformation we also introduced a new instance variable into the body class. The new variable holds the force acting on the body during the force computation phase.

When we ran the C++ version, we discovered that abstractions introduced during the translation process degraded the serial performance. We therefore hand optimized the computation by removing abstractions in the performance critical parts of the code until we had restored the original performance. These optimizations do not affect the parallelization; they simply improve the base performance of the computation.

6.2.2 Application Statistics

The final C++ version consists of approximately 1500 lines of code; the explicitly parallel version consists of approximately 1900 lines of code. The compiler detects five parallel loops in the C++ code. Two of the loops are nested inside other parallel loops, so the heuristic described in Section 5.2 suppresses the exploitation of concurrency in these loops. The generated parallel version contains three parallel loops. Table 2 presents several analysis statistics. For each parallel extent it presents the number of auxiliary call sites in the extent, the number of methods in the extent, the number of independent pairs of methods in the extent and the number of pairs that the compiler had to symbolically execute. All of the parallel extents have a significant number of auxiliary call sites; the compiler would be unable to parallelize any of the extents if the commutativity testing phase included the auxiliary operations. Most of the pairs of invoked methods in the extent are always independent, which means that the compiler has to symbolically execute relatively few pairs.

Parallel Extent	Auxiliary Operation Call Sites	Extent Size	Independent Pairs	Symbolically Executed Pairs
Velocity	5	3	5	1
Force	9	6	17	4
Position	8	3	5	1

Table 2: Analysis Statistics for Barnes-Hut

6.2.3 Compilation Time

The compiler starts by running the Sage++ parser to generate a dep file. This file represents the program in the Sage++ intermediate format. The analysis phase of the compiler reads in the dep file and converts the Sage++ intermediate format into its own internal simplified format. It then runs the analysis to find parallel methods, then generates an annotation file identifying the transformations to perform. A separate code generation file reads the annotation file and the original source code file, the generates the parallel code.

We report compilation times for the compiler running on a Sun Microsystems SparcStation 5 with 32 megabytes of memory. To compile the Barnes-Hut, it takes 0.133 seconds to load in the data structures from the dep file, 2.497 seconds to perform the analysis and 0.176 seconds to generate the annotations. It is important to realize that these numbers come from a compiler that is currently under development. We expect that the numbers may change in the future as we modify the analysis algorithms. In particular the compilation times may get longer as the compiler uses more sophisticated algorithms. We also believe that, given the current state of the art in parallelizing compilers, the performance of the compiler should be a secondary concern. While we believe that compilation time will eventually become an important issue for parallelizing compilers, at present the most important questions deal with functionality (i.e. the raw ability of the compiler to extract the concurrency) rather than compilation times.

6.2.4 Performance Results and Analysis

Table 3 presents the execution times for Barnes-Hut. To eliminate cold start effects, the instrumented computation omits the first two iterations. In practice the computation would perform many iterations and the amortized overhead of the first two iterations would be negligible. The column labeled Serial contains the execution time for the serial C++ program. This program contains only sequential C++ code and executes with no parallelization or synchronization overhead. The rest of the columns contain the execution times for the automatically parallelized version. Figure 17 presents the speedup for the computation. The computation scales reasonably well, exhibiting speedups of between 11 and 12 out of 16 processors and 17 and 18 out of 32 processors.

Number of Bodies	Processors						
	Serial	1	2	4	8	16	32
8192	65.0	63.4	31.9	15.8	8.8	5.3	3.6
16384	146.9	151.8	79.9	39.0	21.9	13.2	8.7

Table 3: Execution Times for Barnes-Hut (seconds)

We start our analysis of the performance with the parallelism coverage [16], which measures the amount of time

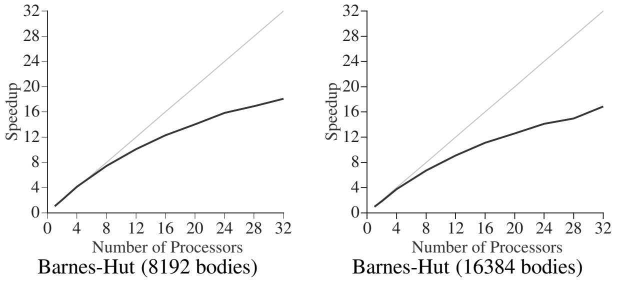


Figure 17: Speedup for Barnes-Hut

that the serial computation spends in parallelized sections. To obtain good parallel performance, the compiler must parallelize a substantial part of the computation. By Amdahl's law any remaining serial sections of the computation impose an absolute limit on the parallel performance. For example, even if the compiler parallelizes 90% of the computation, the parallel computation can run at most 10 times faster than the serial computation. Table 4 presents the parallelism coverage for Barnes-Hut; these statistics show that the compiler is able to parallelize almost all of the computation.

Number of Bodies	Serial Compute Time (seconds)	Time in Parallelized Sections (seconds)	Parallelism Coverage
8192	64.14	62.87	98.02%
16384	153.20	148.34	96.83%

Table 4: Parallelism Coverage for Barnes-Hut

Good parallelism coverage is by itself no guarantee of good parallel performance. To exploit parallelism, the compiler inevitably introduces synchronization and concurrency management overhead. If the granularity of the generated parallel computation is too small to successfully amortize the overhead, the parallel program will perform poorly even if it has good parallelism coverage. A standard problem with traditional parallelizing compilers, for example, has been the difficulty of successfully amortizing the barrier synchronization overhead at each parallel loop [39]. Our prototype compiler introduces four sources of overhead when it generates parallel code:

- **Loop Overhead:** The overhead generated by the execution of a parallel loop. Sources of this overhead include the communication at the beginning of the loop to inform all processors of the loop's execution and barrier synchronization at the end of the loop.

- **Chunk Overhead:** The overhead associated with acquiring a chunk of parallel loop iterations. Sources of this overhead include the computation that determines how many iterations the processor will take, the update of a centralized counter that records which iterations have yet to be assigned to a specific processor for execution and the lock constructs that make the chunk acquisition atomic.

- **Iteration Overhead:** The overhead generated by the execution of one iteration of a parallel loop. This includes function call and argument unpacking overhead.

- **Lock Overhead:** The overhead generated by the lock constructs automatically inserted into methods to make operations execute atomically.

We developed a benchmark program to measure the cost of each source of overhead. Table 5 presents the results. The loop overhead increases with the number of processors; the table presents the loop overhead on 32 processors.

Loop Overhead On 32 Processors	Chunk Overhead	Iteration Overhead	Lock Overhead
211	30	0.38	5.1

Table 5: Parallel Construct Overhead (microseconds)

For each source of overhead the applications execute a corresponding piece of useful work — the loop overhead is amortized by the parallel loop, the chunk overhead is amortized by the chunk of iterations, the iteration overhead is amortized by the iteration and the lock overhead is amortized by the computation between lock acquisitions. The relative size of each piece of work determines if the overhead will have a significant impact on the performance. Tables 6 and 11 present the mean sizes of the pieces of useful work. The numbers in the tables are computed as follows:

- **Loop Size:** The time spent in parallelized sections divided by the number of executed parallel loops. A comparison with the Loop Overhead number in Table 5 shows that the amortized loop overhead is negligible.

- **Chunk Size:** The time spent in parallelized sections divided by the total number of chunks. Because the number of chunks tends to increase with the number of processors, we report the chunk size at 32 processors. A comparison with the Chunk Overhead in Table 5 shows that the amortized chunk overhead is negligible.

- **Iteration Size:** The time spent in parallelized sections divided by the total number of iterations in executed parallel loops. A comparison with the Iteration Overhead in Table 5 shows that the amortized iteration overhead is negligible.

- **Task Size:** The time spent in parallelized sections divided by the number of times that operations acquire a lock. A comparison with the Lock Overhead in Table 5 shows that the amortized lock overhead is negligible.

We instrumented the generated parallel code to measure how much time each processor spends in different parts of the parallel computation. The instrumentation breaks the execution time down into the following categories:

Number of Bodies	Loop Size	Chunk Size 32 Processors	Iteration Size	Task Size
8192	10.5×10^6	68.7×10^3	1.28×10^3	1.28×10^3
16384	24.7×10^6	134×10^3	1.51×10^3	1.51×10^3

Table 6: Granularities for Barnes-Hut (microseconds)

- **Parallel Idle:** The amount of time the processor spends idle while the computation is in a parallel section. Increases in the load imbalance show up as increases in this component.

- **Serial Idle:** The amount of time the processor spends idle when the computation is in a serial section. Currently every processor except the main processor is idle during the serial sections. This component therefore tends to increase linearly with the number of processors, since the time the main processor spends in serial sections tends not to increase dramatically with the number of processors.

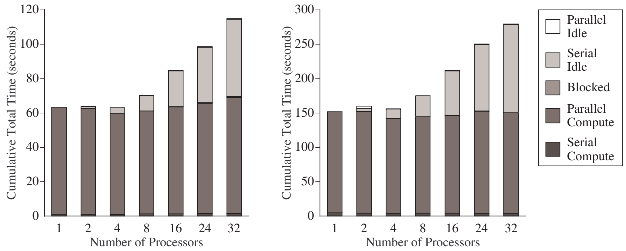
- **Blocked:** The amount of time the processor spends waiting to acquire a lock that an operation executing on another processor has already acquired. Increases in contention for objects are reflected in increases in this component of the time breakdown.

- **Parallel Compute:** The amount of time the processor spends performing useful computation during a parallel section of the computation. This component also includes the lock overhead associated with an operation's first attempt to acquire a lock, but does not include the time spent waiting for another processor to release the lock if the lock is not available. Increases in the communication of application data during the parallel phases show up as increases in this component.

- **Serial Compute:** The amount of time the processor spends performing useful computation in a serial section of the program. With the current parallelization strategy, the main processor is the only processor that executes any useful work in a serial part of the computation.

Given the execution time breakdown for each processor, we compute the cumulative time breakdown by taking the sum over all processors of the execution time breakdown at that processor. Figure 18 presents the cumulative time breakdowns as a function of the number of processors executing the computation. The height of each bar in the graph represents the total processing time required to execute the parallel program; the different gray scale shades in each bar represent the different time breakdown categories. If a program scales perfectly with the number of processors then the height of the bar will remain constant as the number of processors increases.

These graphs show that the limit on the performance is the time spent in the serial phases of the computation — at 32 processors the serial idle time accounts for approximately 40% of the cumulative compute time for both applications.



Barnes-Hut (8k bodies) Barnes-Hut (16k bodies)

Figure 18: Cumulative Time Breakdowns for Barnes-Hut

6.2.5 Comparison with the Explicitly Parallel Version

Table 7 contains the execution times for the explicitly parallel version of Barnes-Hut. For small numbers of processors the automatically parallelized and explicitly parallel versions exhibit roughly comparable performance. For larger numbers of processors the explicitly parallel version performs significantly better — at 32 processors it runs 50% faster for 8192 bodies and 70% faster for 16384 bodies than the automatically parallelized version. The largest contribution to the performance difference is that the explicitly parallel version builds the space subdivision tree in parallel, while the automatically parallelized version builds the tree serially. The explicitly parallel version also uses an application-specific scheduling algorithm called costzones partitioning in the force computation phase [36]. This algorithm provides better locality than the guided self-scheduling algorithm in the automatically parallelized version.

Number	Processors					
of Bodies	1	2	4	8	16	32
8192	73.1	35.2	16.8	8.4	4.4	2.4
16384	154.0	77.4	36.8	19.2	9.7	5.1

Table 7: Execution Times for Explicitly Parallel Barnes-Hut (seconds)

6.3 Water

The main data structure in Water is an array of molecule objects. Almost all of the compute time is spent in two $O(n^2)$ phases. One phase computes the total force acting on each molecule; the other phase computes the potential energy of the collection of molecules.

6.3.1 The Serial C++ Code

The original source of Water is the Perfect Club benchmark MDG, which is written in Fortran. Several students at Stanford University translated this benchmark from Fortran to C as part of a class project. We obtained the serial C++ version by translating this existing serial C version to C++.

As part of the translation process we converted the $O(n^2)$ phases to use auxiliary objects tailored for the way each phase accesses data. Before each phase the computation loads relevant data into an auxiliary object; at the end of the phase the computation unloads the computed values from the auxiliary object to update the molecule objects. This modification increases the precision of the data usage analysis in the compiler, enabling the compiler to recognize the concurrency in the phase.

6.3.2 Application Statistics

The final C++ version consists of approximately 1850 lines of code, the serial C version consists of approximately 1220 lines of code and the explicitly parallel version in the SPLASH benchmark suite consists of approximately 1600 lines of code. Much of the extra code in the C++ version comes from the pervasive use of classes and encapsulation. Instead of directly accessing many of the data structures (as the C versions do), the C++ version encapsulates data in classes and accesses the data via accessor methods. The class declarations and accessor method definitions significantly increase the size of the program. The use of a vector class instead of arrays of doubles, for example, added approximately 230 lines of code.

The analysis finds a total of seven parallel loops. Two of the loops are nested inside other parallel loops, so the generated parallel version contains five parallel loops. Table 8 contains the analysis statistics; as for the Barnes-Hut all of the extents contain auxiliary operation call sites and most of the pairs in the extents are independent.

Parallel Extent	Auxiliary Operation Call Sites	Extent Size	Independent Pairs	Symbolically Executed Pairs
Virtual	9	3	5	1
Energy	1	5	14	1
Loading	5	2	2	1
Forces	3	4	9	1
Momenta	2	2	2	1

Table 8: Analysis Statistics for Water

6.3.3 Compilation Time

For the Water code the compiler takes 0.215 seconds to load the intermediate format, 6.653 seconds to perform the analysis and 0.16 seconds to perform the annotation generation.

6.3.4 Performance Results and Analysis

Table 9 contains the execution times for Water. The measured computation omits initial and final I/O. In practice the computation would execute many iterations and the amortized overhead of the I/O would be negligible. Figure 19 presents the speedup curves. Water initially performs well (the speedup over the sequential C++ version at 8 processors is approximately 4.5 for 343 molecules and 5 for 512 molecules), but it does not scale beyond 8 processors.

Number of Molecules	Processors						
	Serial	1	2	4	8	16	32
343	73.3	77.1	41.0	21.3	16.2	13.8	17.7
512	158.1	167.5	90.4	46.7	31.5	29.9	37.8

Table 9: Execution Times for Water (seconds)

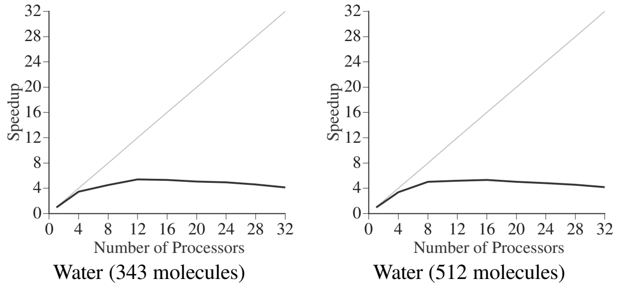


Figure 19: Speedup for Water

Table 10, which presents the parallelism coverage for this application, shows that the compiler parallelizes almost all of the computation. Table 11 shows that all of the sources of overhead are negligible except for the lock overhead. The lock overhead by itself, however, does not explain the lack of scalability.

Number of Molecules	Serial Compute Time (seconds)	Time in Parallelized Sections (seconds)	Parallelism Coverage
343	70.89	69.97	98.70%
512	154.29	152.85	99.07%

Table 10: Parallelism Coverage for Water

Number of Molecules	Loop Size	Chunk Size 32 Processors	Iteration Size	Task Size
343	3.50×10^6	34.0×10^3	10.2×10^3	74.2
512	7.64×10^6	69.1×10^3	14.9×10^3	72.8

Table 11: Granularities for Water (microseconds)

Figure 20, which presents the cumulative time breakdowns for Water, clearly shows why Water fails to scale beyond eight processors. The fact that the blocked component grows dramatically while all other components either grow relatively slowly or remain constant indicates that contention for objects is the primary source of the lack of scalability. It should, in principle, be possible to automatically eliminate the contention by replicating objects to enable conflict-free write access. We expect that this optimization would dramatically improve the scalability.

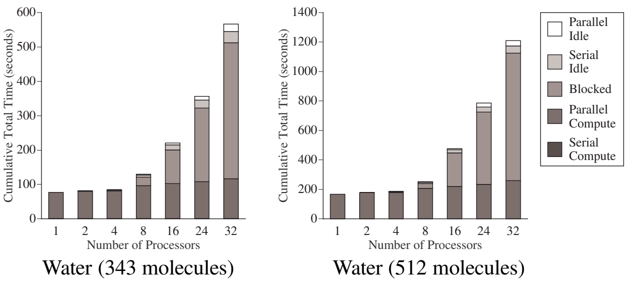


Figure 20: Cumulative Time Breakdowns for Water

6.3.5 Comparison with the Explicitly Parallel Version

The SPLASH parallel benchmark set contains an explicitly parallel version of Water; Table 12 contains the execution times for this version. Unlike the automatically parallelized version, the explicitly parallel version scales reasonably well to 32 processors. We attribute this difference to the fact that the explicitly parallel version replicates several data structures, eliminating the contention that limits the performance of the automatically parallelized version.

Number of Molecules	Processors					
	1	2	4	8	16	32
343	75.68	41.93	20.13	10.35	5.39	3.39
512	166.54	81.98	40.33	21.22	11.46	6.76

Table 12: Execution Times for Explicitly Parallel Water (seconds)

6.4 Caveats

The goal of our project is to enable programmers to exploit both the performance advantages of parallel execution and the substantial programming advantages of the sequential programming paradigm. We view the compiler as a tool that the programmer uses to obtain reliable parallel execution with a minimum of effort. We expect that the programmer will need a reasonable understanding of the compiler's capabilities to use it effectively. In particular, we do not expect to develop a compiler capable of automatically parallelizing a wide range of existing "dusty deck" programs.

Several aspects of our experimental methodology reflect this perspective. As part of the translation process from C to C++ we ensured that the C++ program conformed to the model of computation that the compiler was designed to analyze. We believe that this approach accurately reflects how parallelizing compilers in general will be used in practice. We expect that programmers may have to tune their programs to the capabilities of the compiler to get good performance. The experience of other researchers supports this hypothesis [4, 5].

For our two applications it was relatively straightforward to produce code that the compiler could successfully analyze. Almost all of the translation effort was devoted to expressing the computation in a clean object-based style with classes, objects and methods instead of structures and procedures. The basic structure of both applications remains intact in the final C++ versions, and the C++ versions have better encapsulation and modularity properties than the C versions.

We selected Barnes-Hut and Water in part because other researchers had developed explicitly parallel versions that performed well. We therefore knew that it was possible in principle to parallelize the applications. The question was whether commutativity analysis would be able to automatically discover and exploit the concurrency. In general we expect programmers to use the compiler to parallelize applications that have enough inherent concurrency to keep the machine busy.

7 Future Research

The ideas and results in this paper suggest many possible directions for future research. In this sense we believe the paper is only a start: the culmination and ultimate evaluation of commutativity analysis still lie ahead of us. In this section we briefly mention several future research directions.

7.1 Relative Commutativity

The current formulation of commutativity analysis is absolute. During the execution of a parallelized section of code, the data structures in the parallel and serial versions may diverge. But the compiler guarantees that by the end of the parallel section the data structures in the two versions have converged to become identical.

This formulation is obviously overly conservative. To preserve the semantics of the serial program, it is sufficient to preserve the property that the parallel and serial computations generate data structures that are equivalent with respect to the rest of the computation. For example, the output of the explicitly parallel tree construction algorithm in the Barnes-Hut depends on the relative execution speed of the different processors: different executions on the same input may generate different data structures. But because all of these data structures are equivalent with respect to the rest of the program, the program as a whole executes deterministically.

It may be possible to extend the commutativity analysis framework to automatically generate parallel code for algorithms such as the tree construction algorithm in Barnes-Hut. The current formulation works well for data structure traversal algorithms; the compiler may need to extend the framework to recognize operations that commute relative to the rest of the computation if it is to effectively parallelize data structure construction algorithms.

7.2 Analysis Granularity

Our experience with auxiliary operations shows that the correct analysis granularity does not always correspond to the granularity of methods in the source program. In both Water and Barnes-Hut the method granularity is too fine: for the analysis to succeed it must coarsen the granularity by conceptually integrating auxiliary operations into their callers. We expect a generalized concept of auxiliary operations to eventually emerge, with the compiler promoting the success of the analysis by partitioning the program at an appropriate granularity.

Several issues confront the designer of a partioning algorithm. First, the analysis granularity interacts with the locking algorithm. If the analysis is performed at the granularity of computations that manipulate multiple objects, the generated code may need to hold multiple locks to make the computation atomic. The need to acquire these locks without deadlock may complicate the code generation algorithm. This issue does not arise in the current compiler because it analyzes the computation at the granularity of operations on single objects and generates code that only holds a single lock at a time.

There is tradeoff between increased granularity and analyzability: increasing the analysis granularity may make it difficult for the compiler to extract expressions that accurately denote the computed values. In some cases, the compiler may even need to analyze the program at a finer granularity than the method granularity. This can happen, for example, if a method contains an otherwise unanalyzable loop. Replacing the loop with a tail-recursive method and analyzing the computation at that finer granularity may enable the analysis to succeed.

Finally, coarsening the granularity may waste concurrency. The compiler must ensure that it analyzes the computation at a granularity fine enough to expose a reasonable amount of concurrency in the generated code.

7.3 A Message-Passing Implementation

The current compiler relies on the hardware to implement the abstraction of shared memory. It is clearly feasible, however, to generate code for message-passing machines. The basic required functionality is a software layer that uses message-passing primitives to implement the abstraction of a single shared object store [31, 35]. The key question is how well the generated code would perform on such a platform. Message-passing machines have traditionally suffered from much higher communication costs than shared-memory machines. Compilation research for message-passing machines has therefore emphasized the development of data and computation placement algorithms that minimize communication [18]. Given the dynamic nature of our target application set, the compiler would have to rely on dynamic techniques such as replication and task migration to optimize the locality of the generated computation [9, 32].

7.4 Pointer Analysis

The algorithms in Section 4 perform a data usage analysis at the granularity of the type system. An obvious alternative is to use pointer analysis [13, 41, 23] to identify the regions of memory that each operation may access. A major advantage of this approach for non type-safe languages like C and C++ is that it would allow the compiler to analyze programs that may violate their type declarations. It would also characterize the accessed regions of memory at a finer granularity than the type system, which would increase the precision of the data usage analysis. One potential drawback is a complication of the compiler. The use of pointer analysis would also increase the amount of code that the compiler would have to analyze. Before the compiler could parallelize a piece of code that manipulated a given pointer-based data structure, it would have to analyze the code that built the data structure.

8 Related Work

This section briefly surveys previous research in the area of parallelizing compilers for computations that manipulate irregular or pointer-based data structures. We also discuss reduction analysis and commuting operations in the context of parallel programming languages.

8.1 Data Dependence Analysis

Research on automatically parallelizing serial computations that manipulate pointer-based data structures has focused on techniques that precisely represent the run-time topology of the heap [17, 24, 11, 28]. The idea is that the analysis can use this precise representation to discover independent pieces of code. To recognize independent pieces of code, the compiler must understand the global topology of the manipulated data structures [17, 24]. It must therefore analyze the code that builds the data structures and propagate the results of this analysis through the program to the section that uses the data. A limitation of these techniques 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.

Commutativity analysis differs substantially from data dependence analysis in that it neither depends on nor takes advantage of the global topology of the data structure. This property enables commutativity analysis to parallelize computations that manipulate graphs. It also eliminates the need to analyze the data structure construction code. Commutativity analysis may therefore be appropriate for computations that do not build the data structures that they manipulate. An example of such a computation is a query that manipulates persistent data stored in an object-oriented database. But insensitivity to the data structure topology is not always an advantage. Standard code generation schemes for commutativity analysis insert synchronization constructs to ensure that operations execute atomically. These constructs impose unnecessary overhead when the operations access disjoint sets of objects. If a compiler can use data dependence analysis to recognize that operations are independent, it can generate parallel code that contains no synchronization constructs.

In the long run we believe parallelizing compilers will incorporate both commutativity analysis and data dependence analysis for pointer-based data structures, using each when it is appropriate.

8.2 Reductions

Several existing compilers can recognize when a loop performs a reduction of many values into a single value [15, 14, 27, 8]. 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. Researchers have recently generalized the basic reduction recognition algorithms to recognize reductions of arrays instead of scalars. The reported results indicate that this optimization is crucial for obtaining good performance for the measured set of applications [16].

There are interesting connections between reduction analysis and commutativity analysis. Many (but not all) of the computations that commutativity analysis is designed to handle can be viewed as performing multiple reductions concurrently across a large data structure. The need to exploit reductions in traditional data parallel computations suggests that less structured computations will require generalized but similar techniques.

8.3 Commuting Operations in Parallel Languages

Steele describes an explicitly parallel computing framework that includes primitive commuting operations such as the addition of a number into an accumulator [38]. The motivation is to deliver a flexible system for parallel computing that guarantees deterministic execution. The paper describes an enforcement mechanism that dynamically detects violations of the deterministic paradigm and mentions the possibility that a compiler could statically detect such violations.

There are two fundamental differences between Steele's framework and commutativity analysis: explicit parallelism as opposed to automatic parallelization and dynamic checking as opposed to static recognition of commuting operations. Steele's framework is designed to deliver an improved explicitly parallel programming environment by guaranteeing deterministic execution. The goal of commutativity analysis is to preserve the sequential programming paradigm while using parallel execution to deliver increased performance. While deterministic execution is one of the most important advantages of the serial programming paradigm, there are many others [31].

Commutativity analysis is also designed to recognize complex commuting operations that may recursively invoke other operations. Steele's framework focuses on atomic operations that only update memory. In a dynamically checked, explicitly parallel framework it is natural to view the computation as a set of atomic operations on a mutable store. The concurrency generation is already explicit in the program and the implementataion must only check that the generated primitive operations commute. But because a parallelizing compiler must statically extract the concurrency, it has to convert the serial invocation of operations into parallel execution. In this context it becomes clear that the compiler must reason about how operations are invoked as well as how they access memory.

The implicitly parallel programming language Jade explicitly supports the concept of commuting operations on user-defined objects [21]. In this case the motivation is to extend the range of expressible computations while preserving deterministic execution. It is the programmer's responsibility to ensure that operations that are declared to commute do in fact commute.

Many concurrent object-oriented languages support the notion of mutually exclusive operations on objects [10, 43]. Although the concept of commuting operations is never explicitly identified, the expectation is that all mutually exclusive operations that may attempt to concurrently access the same object commute.

Unlike implementations of Jade and concurrent object-oriented programming languages, a parallelizing compiler that uses commutativity analysis is responsible for verifying that operations commute. The result is therefore guaranteed deterministic execution. If a Jade program declares commuting operations or a program written in a concurrent object-oriented programming language uses mutually exclusive methods, it is the programmer's responsibility to ensure that the operations commute.

9 Conclusion

The difficulty of developing explicitly parallel software limits the enormous potential of parallel computing. The problem is especially acute for irregular, dynamic computations that manipulate pointer-based data structures such as graphs. Commutativity analysis addresses this problem by promising to extend the reach of parallelizing compilers to include pointer-based computations.

We have developed a parallelizing compiler that uses commutativity analysis as its main analysis paradigm. We have used this compiler to automatically parallelize two complete scientific applications. The performance of the generated code provides encouraging evidence that commutativity analysis can serve as the basis for a successful parallelizing compiler.

Acknowledgements

The authors would like to thank the anonymous referees for their many valuable comments.

References

[1] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, MA, 1988.

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

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

[4] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. ICASE Report 827, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, IL, May 1989.

[5] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[6] W. Blume and R. Eigenmann. Symbolic range propagation. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 357–363, Santa Barbara, CA, April 1995.

[7] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and Beata Winnicka. Sage++: An object-oriented toolkit and class library for building Fortran and C++ structuring tools. In *Proceedings of the Object-Oriented Numerics Conference*, 1984.

[8] 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.

[9] M. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[10] R. Chandra, A. Gupta, and J. Hennessy. Data locality and load balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[11] D. Chase, M. Wegman, and F. Zadek. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, White Plains, NY, June 1990.

[12] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four Perfect benchmark programs. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

[13] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*, Orlando, FL, June 1994.

[14] 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.

[15] A. Ghuloum and A. Fisher. Flattening and parallelizing irregular, recurrent loop nests. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[16] M.W. Hall, S.P. Amarasinghe, B.R. Murphy, S. Liao, and M.S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, December 1995.

[17] 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.

[18] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[19] O. Ibarra, P. Diniz, and M. Rinard. On the complexity of commutativity analysis. In *Proceedings of the 2nd Annual International Computing and Combinatorics Conference*, Hong Kong, June 1996.

[20] R. Kemmerer and S. Eckmann. UNISEX: a UNIx-based Symbolic EXecutor for pascal. *Software—Practice and Experience*, 15(5):439–458, May 1985.

[21] M. Lam and M. Rinard. Coarse-grain parallel programming in Jade. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 94–105, Williamsburg, VA, April 1991.

[22] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[23] W. Landi and B. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[24] J. Larus and P. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Program Language Design and Implementation*, Atlanta, GA, June 1988.

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

[26] 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.

[27] S. Pinter and R. Pinter. Program optimization and parallelization using idioms. In *Proceedings of the Eighteenth Annual ACM Symposium on the Principles of Programming Languages*, Orlando, FL, January 1991.

[28] 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.

[29] C. Polychronopoulos and D. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, pages 1425–1439, December 1987.

[30] W. Pugh and D. Wonnacott. Eliminating false data dependences using the Omega test. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[31] M. Rinard. *The Design, Implementation and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language*. PhD thesis, Stanford, CA, 1994.

[32] M. Rinard. Communication optimizations for parallel computing using data access information. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[33] M. Rinard and P. Diniz. Commutativity analysis: A technique for automatically parallelizing pointer-based computations. In *Proceedings of the 10th International Parallel Processing Symposium*, Honolulu, HI, April 1996.

[34] J. K. Salmon. *Parallel Hierarchical N-body Methods*. PhD thesis, California Institute of Technology, December 1990.

[35] D. Scales and M. S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, Monterey, CA, November 1994.

[36] J. Singh. *Parallel Hierarchical N-body Methods and their Implications for Multiprocessors*. PhD thesis, Stanford University, February 1993.

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

[38] 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.

[39] C. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 144–155, Santa Barbara, CA, July 1995.

[40] W. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.

[41] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995.

[42] 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.

[43] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object oriented concurrent programming in ABCL/1. In *Proceedings of the OOPSLA-86 Conference*, pages 258–268, Portland OR, September 1986.