Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers

MARTIN C. RINARD
Massachusetts Institute of Technology
and
PEDRO C. DINIZ
University of Southern California / Information Sciences Institute

This article 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 technique. We have used this system to automatically parallelize three complete scientific computations: the Barnes-Hut N-body solver, the Water liquid simulation code, and the String seismic simulation code. This article 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.

Categories and Subject Descriptors: D.3.4 [**Compilers**]: Parallelizing Compilers

General Terms: Parallelizing Compilers, compilers, parallel computing

Additional Key Words and Phrases: Parallel computing

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

The research reported in this article was conducted while the authors were affiliated with the University of California, Santa Barbara. Martin Rinard was partially supported by an Alfred P. Sloan Foundation Research fellowship. Pedro Diniz was sponsored by the PRAXIS XXI program administered by JNICT - Junta Nacional de Investigação Científica e Tecnológica from Portugal and held a Fulbright travel grant.
Authors' current addresses: Martin C. Rinard, Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Mass., 02139. E-mail: martin@lcs.mit.edu; Pedro C. Diniz, University of Southern California/Information Sciences Institute, 4676 Admiralty Way, Suite 1001, Marina del Rey, Calif., 90292. E-mail: pedro@isi.edu

data dependence analysis to parallelize computations that manipulate dense arrays using affine access functions [Banerjee 1988; Eigenmann et al. 1991; Hall et al. 1995; Pugh and Wonnacott 1992]. 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 [Banerjee et al. 1993].

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 [Lampson and Redell 1980]. 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 [Singh et al. 1992] and three of the four parallel applications in the SAM benchmark suite [Scales and Lam 1994] 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 recognize and exploit commuting operations.

We have developed a new analysis technique called commutativity analysis. This technique 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 machine [Lenoski 1992] and on multiprocessors from Silicon Graphics.

We have used the compilation system to automatically parallelize three complete scientific applications: the Barnes-Hut N-body solver [Barnes and Hut 1986], the Water simulation code [Woo et al. 1995], and the String seismic code [Harris et al. 1990]. 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. Water is a more traditional scientific computation that organizes its data as arrays of objects representing water molecules. String manipulates an array data structure, but has very irregular data access and control patterns. 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 article makes the following contributions:

—It describes a new analysis technique, commutativity analysis, that can automatically recognize and exploit commuting operations to generate parallel code.

—It describes extensions to the basic commutativity analysis technique. 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. These algorithms allow a compiler to discover parallelizable computations and generate parallel code.

—It presents performance results for automatically parallelized versions of three 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 have designed commutativity analysis to parallelize serial programs, it may also benefit other areas of computer science. For example, commuting operations allow computations on persistent data in object-oriented databases to execute in parallel. Transaction processing systems can exploit commuting operations to use more efficient locking algorithms [Weihl 1988]. 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 article is structured as follows. Section 2 presents an example that shows how commuting operations enable parallel execution. Section 3 presents an overview of commutativity analysis. 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 three 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 (written in C++) that shows how commuting operations enable parallel execution. The `node::visit` method in Figure 1 defines a computation that serially traverses the nodes of a graph. When the traversal completes, each node's `sum` instance variable contains the sum of its original value and the values of the `value` instance variables in all of the nodes that directly point to that node.

The traversal invokes the `node::visit` method once for each edge in the graph. Each invocation specifies the node to visit (this node is called the *receiver object*) and the value of the parameter `p`. By definition, an operation is a method together with a receiver object and parameter values. Each invocation of the `node::visit` method therefore corresponds to a `node::visit` operation.

When a `node::visit` operation executes, it first adds the parameter `p` into the running sum stored in the receiver's `sum` instance variable. It then checks the receiver's `marked` instance variable to see if the traversal has already visited the receiver. If the traversal has not visited the receiver, it marks the receiver and invokes the `node::visit` method for the left and right subgraphs of the receiver.

```
class node {
  private:
    boolean marked;
    int value, sum;
    node *left, *right;
  public:
    void visit(int);
};

void node::visit(int p) {
  sum = sum + p;
  if (!marked) {
    marked = TRUE;
    if (left != NULL) left->visit(value);
    if (right != NULL) right->visit(value);
  }
}
```

Fig. 1. Serial graph traversal.

The way to parallelize the computation is to execute the two recursive invocations of the `node::visit` method in parallel. But this parallelization may violate the data dependences of the original serial computation. Consider what may happen if the traversals of a left subgraph and a right subgraph access the same node. In the serial computation, all of the accesses generated by the left traversal execute before all of the accesses generated by the right traversal. In a parallel computation, however, 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, the traversal executes the recursive invocations only the first time it visits a node. 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 traversal even though the resulting computations may differ substantially from the original serial computation. The key properties that enable the parallelization are that the parallel computation executes the same set of `node::visit` operations as the serial computation and that the `node::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 `node::visit` method in Figure 2. The top level `node::visit` method first invokes the `node::parallel_visit` method, then invokes the **wait** construct, which blocks until the entire parallel computation completes. The `node::parallel_visit` method executes the recursive invocations concurrently using the **spawn** construct, which creates a new task for each operation. A straightforward application of lazy task creation techniques [Mohr et al. 1990] can increase the granularity of the resulting parallel computation. The current implementation of the compiler generates code that uses a load-based

```
class node {
  private:
    lock mutex;
    boolean marked;
    int value, sum;
    node *left, *right;
  public:
    void visit(int);
    void parallel_visit(int);
};

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

void node::parallel_visit(int p) {
  mutex.acquire();
  sum = sum + p;
  if (!marked) {
    marked = TRUE;
    mutex.release();
    if (left != NULL) spawn(left->parallel_visit(value));
    if (right != NULL) spawn(right->parallel_visit(value));
  } else {
    mutex.release();
  }
}
```

Fig. 2. Parallel graph traversal.

serialization strategy for **spawn** constructs to reduce excessive parallelism.[1]

The compiler also augments each `node` object with a mutual exclusion lock `mutex`. The generated parallel operations use this lock to ensure that they update the receiver object atomically.

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

[1]The run-time system maintains a count of the number of idle processors. The generated code for a **spawn** construct first checks the count to determine if any processors are idle. If any processors are idle, the generated code creates a new task for the operation. If not, the generated code executes the operation serially as part of the task executing the **spawn** construct.

language such as an integer, a double or a pointer to an object, or an array of nested objects or primitive types. In the example in Figure 1, each graph node is an object.

By definition, an operation consists of a method, a receiver object, and the values of the parameters. In the graph traversal example in Figure 1, operations consist of the `node::visit` method, a graph node that is the receiver object, and an integer parameter. To execute an operation, the machine binds the operation's receiver and parameter values to the formal receiver and parameter variables of the method, then executes the code in the method. When the operation executes, it can access the parameters, invoke other operations, or access the instance variables of the receiver.

This model of computation is designed to support an imperative programming style. As the graph traversal example illustrates, computations often interact by imperatively updating objects. Operations often calculate a contribution to the final result, then update the receiver to integrate the contribution into the computation.

3.1.1 *Instance Variable Access Restrictions.* There are several restrictions on instance variable accesses. If the instance variable is an instance variable of a nested object, the operation may not directly access the instance variable — it may access the variable only 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 may not directly access the instance variable — it may access the variable only indirectly by invoking operations whose receiver's class is the parent class.

3.1.2 *Separability.* Commutativity analysis is designed to work with *separable* operations. An operation is separable if its execution can be decomposed into an *object section* and an *invocation section*. The object section performs all accesses to the receiver. The invocation section invokes 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 separable methods via the introduction of additional auxiliary methods.

The execution of a separable operation in a parallel computation consists of two phases: the atomic execution of the object section[2] and the invocation in parallel of all of the operations in the invocation section. In this model of parallel computation, the parallel execution invokes operations in a breadth-first manner: the computations of the invoked operations from each invocation section are generated and execute in parallel. The precise order in which operations on the same object execute is determined only at run time and may vary from execution to execution. The serial execution, on the other hand, executes the invoked operations in a depth-first manner: the entire computation of an invoked operation completely finishes before the next operation in the invocation section is invoked.

Separability may appear to make it difficult to develop computations that read values from the receiver, invoke operations that use the values to compute a result, then use the result to imperatively update the receiver. As mentioned above, it would be possible

[2]Automatically generated lock constructs ensure that the object section executes atomically.

to decompose the operations in such computations to make them conform to the strict definition of separability. A potential problem is that such an operation decomposition would make the analysis granularity finer. In the worst case, the coarser-granularity operations in the original computation would commute, but the finer-granularity operations in the transformed computation would not commute. Sections 3.4.1 and 3.4.2 discuss two extensions to the programming model — *extent constants* and *auxiliary methods* — that often allow the compiler to successfully analyze coarse-grain operations that read values from the receiver, invoke operations that use the values to compute a result, then use the result to imperatively update the receiver.

3.2 Commutativity Testing

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

—*Instance Variables:* The new value of each instance variable of the receiver objects of A and B must be the same after the execution of the object section of A followed by the object section of B as after the execution of the object section of B followed by the object section of A.

—*Invoked Operations:* The multiset of operations directly invoked by either A or B under the execution order A followed by B must be the same as the multiset of operations directly invoked by either A or B under the execution order B followed by A.[3]

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

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

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

3.2.1 *Symbolic Commutativity Testing.* The compiler can use *symbolic commutativity testing* to apply the commutativity-testing conditions. Symbolic commutativity testing works with *symbolic operations* to reason about the values computed in the two different execution orders. By definition, a symbolic operation consists of a method and symbolic expressions that denote the receiver and parameter values.

We illustrate the concept of symbolic commutativity testing by applying it to the example in Figure 1. The compiler determines that it must test if two symbolic operations

[3]Two operations are the same if they execute the same method and have the same receiver and parameter values.

`r->visit(`p_1`)` and `r->visit(`p_2`)` commute. The symbolic operation `r->visit(`p_1`)` has parameter p_1; the symbolic operation `r->visit(`p_2`)` has parameter p_2; and both operations have the same receiver `r`.

To apply the commutativity-testing conditions, the compiler symbolically executes the two symbolic operations in both execution orders. Symbolic execution simply executes the operations, computing with expressions instead of values [Kemmerer and Eckmann 1985; King 1976; 1981]. Table I contains the two expressions that the symbolic execution extracts for the `sum` instance variable. 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 the two expressions denote the same value.[4] The compiler can use a similar approach to discover that the values of the other instance variables are always the same in both execution orders and that, together, the operations always directly invoke the same multiset of operations.

Table I. New Values of `sum` under Different Execution Orders

Execution Order	New Value of `sum`
`r->visit(`p_1`); r->visit(`p_2`)`	`(sum +` p_1`) +` p_2
`r->visit(`p_2`); r->visit(`p_1`)`	`(sum +` p_2`) +` p_1

In certain circumstances, the compiler may be unable to extract expressions that precisely represent the values that operations compute. In the current compiler, this may happen, for example, if one of the methods contains unstructured flow of control constructs such as **goto** constructs. In this case, the compiler marks the method as unanalyzable. The commutativity-testing algorithm conservatively assumes that invocations of unanalyzable methods do not commute with any operation.

3.2.2 *Independence Testing.* Independent operations commute (two operations are independent if neither accesses a variable that the other writes). It may improve the efficiency or effectiveness of the commutativity-testing algorithm to test first for independence, then apply the full symbolic commutativity-testing algorithm only if the independence test fails. Although it is possible to apply arbitrarily complicated independence-testing algorithms [Hendren et al. 1994; Pugh and Wonnacott 1992], the current compiler applies two simple, efficient independence tests. It tests 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.

3.3 Extents

A policy in the compiler must choose the computations to attempt to parallelize. The current policy is that the compiler analyzes one computation for each method in the program. The computation consists of all operations that may be either directly or indirectly invoked as a result of executing the method. To reason about the computation, the compiler computes a conservative approximation to the set of invoked operations. This approximation is called

[4] 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.

the *extent* of the method. By definition, the extent of a method is the set of methods that could be invoked either directly or indirectly during an execution of the method. In the example in Figure 1, the extent of the `node::visit` method is {`node::visit`}. This extent specifies that any execution of the `node::visit` method will invoke only the `node::visit` method, but that the `node::visit` method may be invoked multiple times with different receivers and arguments. By definition, an operation is in an extent if its method is in the extent.

If the compiler can verify that all pairs of operations in the extent commute, it marks the method as a parallel method. If some of the pairs may not commute, the compiler marks the method as a serial method.

3.4 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.

3.4.1 *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 computation. If the operation reads a variable that none of the operations in the computation write, the variable will have the same value regardless of when the operation executes relative to all of the other operations. We call such a variable an *extent constant variable*.

If an operation computes a value that does not depend on state modified by other operations in the computation, the value will be the same regardless of when the operation executes relative to the other operations. 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 access only extent constants.

The compiler relaxes the model of computation to allow operations to freely access extent constant variables.

3.4.2 *Auxiliary Methods.* For modularity purposes, programmers often encapsulate the computation of values inside methods. The method obtains the computed values either as the return value of the called method or via local variables passed by reference into the called method. We call these methods *auxiliary methods*. Integrating auxiliary methods 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.

The compiler relaxes the model of computation to allow programs to contain two kinds of methods: full methods and auxiliary methods. Full methods may access objects, read reference parameters, or invoke full or auxiliary methods, but they do not return values or write values into reference parameters. Auxiliary methods may return values, read and write reference parameters, compute extent constant values, or invoke other auxiliary methods, but they do not write instance variables, read instance variables that other methods in the extent may write, read variables that may not contain extent constant values, or invoke full methods. As described in Sections 4.4 and 5.3, the compiler automatically recognizes full and auxiliary methods and treats them differently in the analysis and code generation.

Auxiliary methods are conceptually part of their callers, and they execute serially with respect to the enclosing full methods. Full methods therefore observe the correct sequence of values for local variables passed by reference to auxiliary methods. Extents contain only full methods. We also relax the notion of separability to allow the object section to invoke auxiliary methods. Auxiliary methods therefore support computations that read values from the receiver, invoke an auxiliary method to compute new results, and then use the computed results to imperatively update the receiver.

4. ANALYSIS ALGORITHMS

In this section, we present analysis algorithms that a compiler can use to realize the basic approach outlined in Section 3. We first generalize several concepts to include methods. By definition, two methods are independent if all invocations of the two methods are independent. By definition, two methods commute if all invocations of the two methods commute. By definition, a method is separable if all invocations of the method are separable.

The analysis algorithms use the type information to characterize how methods access variables. This variable access information is then used to identify extent constant variables, auxiliary methods, and independent methods. The basic assumption behind this approach is that the program does not violate its type declarations.

To simplify the presentation of the algorithms, we assume that no method returns a value to its caller using the call/return mechanism. Auxiliary methods instead write the computed values into reference parameters. The caller can then access the values by reading the variables passed by reference into the auxiliary method. The presented algorithms generalize in a straightforward way to handle auxiliary methods that return values. The implemented algorithms in the prototype compiler handle auxiliary methods that return values.

4.1 Overview

The commutativity analysis algorithm determines if it is possible to parallelize a given method. The algorithm performs the following steps:

—*Extent Constant Variable Identification:* The algorithm traverses the call graph to find the set of extent constant variables.

—*Extent and Full and Auxiliary Call Site Identification:* The algorithm performs another traversal of the call graph to compute the extent and to divide the call sites into call sites that invoke full methods and call sites that invoke auxiliary methods. The classification is based on how the entire executions of methods read and write externally visible variables.

If the set of externally visible variables that any execution of the method writes includes only instance variables and if the set of externally visible variables that any execution of the method reads includes only reference parameters and instance variables, the method is classified as a full method, and all call sites that invoke that method are classified as full call sites. Otherwise, the method is classified as an auxiliary method, and all call sites that invoke the method are classified as auxiliary call sites.

—*Full Call Site Checks:* The analysis algorithms require that all variables passed by reference into full methods contain only extent constant values.[5] The algorithm checks this condition, in part, by visiting the full call sites to check that the caller writes only extent constant values into variables that are passed by reference into full methods.

—*Auxiliary Call Site Checks:* The analysis algorithms require that any execution of an auxiliary method computes only extent constant values. They also require that the set of externally visible variables that any execution of an auxiliary method writes includes only local variables of the caller. The algorithm checks all of the auxiliary call sites to make sure that they satisfy these conditions.

—*Separability and Input/Output Checks:* The algorithm checks that all of the methods in the extent are separable and that they do not perform input or output. Parallelizing computations that may perform input or output may change the order of the input or output operations, which may violate the semantics of the original serial program.

—*Commutativity Testing:* The algorithm uses independence tests and symbolic commutativity tests to determine if all operations in the extent commute.

Figure 3 presents the algorithm that determines if it is possible to parallelize the execution of a given method. In the succeeding sections, we present the basic functionality that the algorithm is based on and discuss each of the routines that it uses to perform the analysis.

4.2 Basic Functionality

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. Parameters of primitive type are treated as local variables. An array of nested objects is treated as a nested object of the corresponding type.

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.

[5]A variable that contains only extent constant values is different from an extent constant variable. An extent constant variable is a variable that is not written by any operation in the extent. A variable that contains only extent constant values may be written multiple times during the computation, but each time the written value is an extent constant value. The concept of variable that contains only extent constant values is useful because, as described in Sections 3.4.1 and 3.4.2, extent constant values improve the analysis framework in many ways. Programs typically use local variables to transfer extent constant values between full and auxiliary methods and between computations in the same method. Recognizing local variables that contain only extent constant values therefore enhances the ability of the compiler to recognize when computations only access extent constant values, which in turn improves the analysis framework as described in Sections 3.4.1 and 3.4.2.

```
IsParallel(m)
    ec = ExtentConstantVariables(m);
    ⟨extent, full, aux⟩ = ExtentAndFullAndAuxiliaryCallSites(m, ∅, ∅, ∅);
    if (not CheckFullCallSites(m, full, ec)) return false;
    if (not CheckAuxiliaryCallSites(aux, ec)) return false;
    for all m' ∈ extent
        if (not Separable(m', aux, ec)) return false;
        if (MayPerformInputOrOutput(m')) return false;
    for all ⟨m1, m2⟩ ∈ extent × extent
        if (not Commute(m1, m2, aux, ec)) return false;
    return true;
```

Fig. 3. Algorithm to recognize parallel methods.

The example is a simple force calculation algorithm inspired by the force calculation phase of the Barnes-Hut algorithm described in Section 6.2. Figure 5 presents the *CL*, *V*, *N*, *L*, *M*, *C*, *T*, and *P* sets for the example.

The analysis uses storage descriptors $s \in S = P \cup L \cup T \cup (CL \times V) \cup (CL \times Q \times V)$ to represent how computations access variables. In this definition, $q \in Q = \text{seq}(N)$ is the set of nonempty sequences of nested object names. We write an element of Q in the form $n_1.n_2.\cdots.n_i$, an element of $CL \times Q \times V$ in the form $cl.n_1.n_2.\cdots.n_i.v$, and an element of $CL \times V$ in the form $cl.v$. The function $\texttt{class} : CL \times Q \to CL$ gives the class of a nested object. The function $\texttt{type} : S \to T$ gives the type of a storage descriptor. Figure 6 presents the `type` function in the example. The function $\texttt{lift} : S \to T \cup (CL \times V) \cup (CL \times Q \times V)$ translates local variables and parameters to their primitive types. By definition $\texttt{lift}(s) = \texttt{type}(s)$ 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 variables that s_1 represents is a subset of the set of variables that s_2 represents. By definition, $cl_1.v \preceq cl_2.v$ if cl_1 inherits from cl_2 or $cl_1 = cl_2$. Also by definition, $cl_1.q_1.v \preceq cl_2.v$ 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 if $\texttt{class}(cl_1.q_1) = cl_2$. Finally, $s_1 \preceq s_2$ if $\texttt{type}(s_1) = s_2$.

Given a method, the function $\texttt{callSites} : M \to 2^C$ returns its set of call sites, and the function $\texttt{referenceParameters} : M \to 2^P$ returns its set of formal reference parameters. Given a call site, the function $\texttt{method} : C \to M$ returns the invoked method. We also use the standard functions $\texttt{map}(f, A) = \{f(a).a \in A\}$ and $\mathcal{I} : S \to S$ (the identity function from S to 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 by $b(s)$ 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. Figure 7 presents the `callSites`, `referenceParameters`, `method` and `bind` functions in the example.

The compiler performs some local analysis on each method to extract several functions that describe the way the method accesses variables. 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 variables. 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. Similarly, $p \in \texttt{read}(m, b)$ if m reads

```
class node {
  private:
    double mass, position, force;
    node *left, *right;
  public:
    void stop(node *, double, double, boolean *);
    void interact(double, double);
    void traverse(node *, double, double);
};

void node::stop(node *n, double l, double r, boolean *ret) {
  if ((n->left == NULL) && (n->right == NULL)) {
    *ret = true;
  } else {
    double ratio = (l - r) / (this->position - n->position);
    ratio = ratio * ratio;
    if (ratio < 1.0) *ret = true; else *ret = false;
  }
}

void node::interact(double m, double d) {
  this->force = this->force + (this->mass * m)/(d * d);
}

void node::traverse(node *n, double l, double r) {
  boolean flag = 0;
1:this->stop(n, l, r, &flag);
  if (flag) {
2:  this->interact(n->mass, n->position - this->position);
  } else {
    double m = (r + l) / 2;
3:  if (n->left != NULL) this->traverse(n->left, l, m);
4:  if (n->right != NULL) this->traverse(n->right, m, r);
  }
}

class system {
  private:
    int num_bodies;
    node bodies[MAX_NUM_BODIES];
    node *root;
    double l,r;
  public:
    void forces();
};

void system::forces() {
  int i;
  for (i = 0; i < this->num_bodies; i++) {
5:  this->bodies[i].traverse(root, this->l, this->r);
  }
}
```

Fig. 4. Simple force calculation example.

$$
\begin{array}{lcl}
CL & = & \{\texttt{node}, \texttt{system}\} \\
V & = & \{\texttt{mass}, \texttt{position}, \texttt{force}, \texttt{left}, \texttt{right}, \texttt{num\_bodies}, \texttt{root}, \texttt{l}, \texttt{r}\} \\
N & = & \{\texttt{bodies}\} \\
L & = & \{\texttt{node::stop::n}, \texttt{node::stop::l}, \texttt{node::stop::r}, \texttt{node::stop::ret}, \\
 & & \quad \texttt{node::stop::ratio}, \texttt{node::interact::m}, \texttt{node::interact::d}, \\
 & & \quad \texttt{node::traverse::n}, \texttt{node::traverse::l}, \texttt{node::traverse::r}, \\
 & & \quad \texttt{node::traverse::flag}, \texttt{node::traverse::m}, \texttt{system::forces::i}\} \\
M & = & \{\texttt{node::stop}, \texttt{node::interact}, \texttt{node::traverse}, \texttt{system::forces}\} \\
C & = & \{1, 2, 3, 4, 5\} \\
T & = & \{\texttt{double}, \texttt{int}, \texttt{boolean}, \texttt{boolean *}, \texttt{node *}, \texttt{system *}\} \\
P & = & \{\texttt{*node::stop::ret}\}
\end{array}
$$

Fig. 5. *CL*, *V*, *N*, *L*, *M*, *C*, *T* and *P* in the example.

```
type(*node::stop::ret)          =  boolean
type(node::stop::n)             =  node *
type(node::stop::l)             =  double
type(node::stop::r)             =  double
type(node::stop::ret)           =  boolean *
type(node::stop::ratio)         =  double
type(node::interact::m)         =  double
type(node::interact::d)         =  double
type(node::traverse::n)         =  node *
type(node::traverse::l)         =  double
type(node::traverse::r)         =  double
type(node::traverse::flag)      =  boolean
type(node::traverse::m)         =  double
type(system::forces::i)         =  int
type(double)                    =  double
type(int)                       =  int
type(boolean)                   =  boolean
type(boolean *)                 =  boolean *
type(node *)                    =  node *
type(system *)                  =  system *
type(node.mass)                 =  double
type(node.position)             =  double
type(node.force)                =  double
type(node.left)                 =  node *
type(node.right)                =  node *
type(system.num_bodies)         =  int
type(system.bodies.mass)        =  double
type(system.bodies.position)    =  double
type(system.bodies.force)       =  double
type(system.bodies.left)        =  node *
type(system.bodies.right)       =  node *
type(system.root)               =  node *
type(system.l)                  =  double
type(system.r)                  =  double
```

Fig. 6. The `type` function in the example.

```
callSites(node::stop) = ∅
callSites(node::interact) = ∅
callSites(node::traverse) = {1,2,3,4}
callSites(system::forces) = {5}

referenceParameters(node::stop) = {*node::stop::ret}
referenceParameters(node::interact) = ∅
referenceParameters(node::traverse) = ∅
referenceParameters(system::forces) = ∅

method(1) = node::stop
method(2) = node::interact
method(3) = node::traverse
method(4) = node::traverse
method(5) = node::traverse

bind(1,b) = [*node::stop::ret ↦ node::traverse::flag]
bind(2,b) = ∅
bind(3,b) = ∅
bind(4,b) = ∅
bind(5,b) = ∅
```

Fig. 7. The `callSites`, `referenceParameters`, `method`, and `bind` functions in the example.

```
read(node::stop,b) = {node.left, node.right, node.position}
read(node::interact,b) = {node.force, node.mass}
read(node::traverse,b) = {node.mass, node.position, node.left, node.right}
read(system::forces,b) = {system.num_bodies, system.l, system.r}

write(node::stop,b) = {b(*node::stop::ret)}
write(node::interact,b) = {node.force}
write(node::traverse,b) = ∅
write(system::forces,b) = ∅

dep(1) = ∅
dep(2) = ∅
dep(3) = ∅
dep(4) = ∅
dep(5) = ∅
```

Fig. 8. The `read`, `write`, and `dep` functions in the example.

the reference parameter p in the context of the binding b. The function $\texttt{write} : M \times B \rightarrow 2^S$ returns the set of storage descriptors that represent how the method writes variables. The function $\texttt{dep} : C \rightarrow 2^S$ returns the set of storage descriptors that represent the variables that the surrounding method reads to compute the values in the reference parameters at the given call site. Figure 8 presents the `read`, `write`, and `dep` functions in the example.

```
ExtentConstantVariables(m)
    ⟨rd, wr⟩ = ExternallyVisibleReadsAndWrites(m);
    rd = map(lift, rd);
    wr = map(lift, wr);
    return ReadOnlyVariables(rd, wr);

ExternallyVisibleReadsAndWrites(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⟩;

ReadOnlyVariables(rd, wr)
    readonly = rd;
    for all s ∈ rd
        for all s' ∈ wr
            if ((s ⪯ s') or (s' ⪯ s))
                readonly = readonly − {s};
    return readonly;
```

Fig. 9. Algorithm to recognize extent constant variables.

4.3 Extent Constant Variables

The ExtentConstantVariables routine in Figure 9 computes the set of extent constant variables for a given method. The ExternallyVisibleReadsAndWrites routine performs the core computation, using abstract interpretation to compute a read set and write set of storage descriptors that accurately represent how the entire execution of the method may read and write variables. The ReadOnlyVariables routine prunes the read set so that it contains only storage descriptors that represent variables that the computation does not write. Figure 10 presents the results of the extent constant variables computation for all of the methods in the example.

4.3.1 *Correctness of Extent Constant Variables Algorithm.* We next argue that the algorithm is correct. We start with the ExternallyVisibleReadsAndWrites algorithm. Consider the execution of any operation *o* invoked as a result of executing the method *m*. The ExternallyVisibleReadsAndWrites algorithm must produce a read set and a write set that accurately represent how *o* accesses variables. By definition, a pair $\langle m', b \rangle$ is a *representative pair* for *o* if m' is *o*'s method and if *b* accurately represents its reference parameters. If the algorithm ever inserts a representative pair for *o* into $current$, the produced read and write sets

will accurately represent how o accesses variables: the ExternallyVisibleReadsAndWrites algorithm will put $\texttt{read}(m', b)$ and $\texttt{write}(m', b)$ into the read and write sets.

If an operation is executed as a result of executing *m*, there must be a call chain from an invocation of *m* to the invoked operation. We outline a proof, by induction on the length of this call chain, that for every operation o that may be invoked either directly or indirectly as a result of executing *m*, the algorithm inserts a representative pair $\langle m', b \rangle$ for o into $current$. In the base case of the induction, the call chain is of length zero; in other words, the operation is an invocation of *m*. In this case, the algorithm inserts the pair $\langle m, \mathcal{I} \rangle$ into $current$, which is a representative pair for o.

In the induction step, we assume that for every operation o_n that may be invoked at depth n of a call chain, there exists a representative pair $\langle m_n, b_n \rangle$ for o_n that is inserted into $current$. We show that, for every operation o_{n+1} invoked at depth $n + 1$ of a call chain, a representative pair $\langle m_{n+1}, b_{n+1} \rangle$ is inserted into $current$. Any operation o_{n+1} invoked at depth $n + 1$ of a call chain must be invoked by some call site c in one of the methods of an operation o_n invoked at depth n of the call chain. By the induction hypothesis, a representative pair $\langle m_n, b_n \rangle$ for o_n must have been inserted into $current$. The algorithm will therefore insert the pair $\langle \texttt{method}(c), \texttt{bind}(c, b) \rangle$ into $current$, and $\langle \texttt{method}(c), \texttt{bind}(c, b) \rangle$ is a representative pair for o_{n+1}.

The ExternallyVisibleReadsAndWrites algorithm removes all of the local variable storage descriptors from the read and write sets before it returns them. Because the lifetimes of the local variables are contained in the execution of the method *m*, accesses to these variables are not visible outside the execution of *m*.

The ReadOnlyVariables algorithm extracts the set of extent constant variables from the read and write sets. The potential complication is that the computation may read and write the same variable, but that the storage descriptor that represents the read may be different from the storage descriptor that represents the write. The algorithm therefore uses the partial order $\preceq$ to remove any storage descriptors from the read set that represent variables that the computation may write. The end result is a set of storage descriptors that represent variables that the computation may read but does not write.

4.3.2 *Alternatives to the Extent Constant Variables Algorithm.* The extent constant variables algorithm uses the type system to characterize how the method accesses externally visible variables. Two advantages of this approach are its simplicity and ease of implementation. An obvious alternative is to use pointer analysis [Emami et al. 1994; Landi et al. 1993; Wilson and Lam 1995] to identify the variables that each operation may access. An advantage of this approach for non-type-safe languages is that it would allow the compiler to analyze programs that may violate their type declarations. It would also characterize the accessed variables at a finer granularity than the type system, which could 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 data structure, it would have to analyze the code that built the data structure.

It would also be possible to use an *effect system* to characterize how operations access data [Gifford et al. 1987; Hammel and Gifford 1988]. The advantage would be a more precise characterization of how the program accesses data. The integration with the type system would also allow the programmer to manage the granularity of the characterization. The disadvantage would be the need to change the type system of the language.

ExternallyVisibleReadsAndWrites(`node::stop`) =
⟨{`node.left`,`node.right`,`node.position`}, {`*node::stop::ret`}⟩
ExternallyVisibleReadsAndWrites(`node::interact`) =
⟨{`node.force`,`node.mass`},{`node.force`}⟩
ExternallyVisibleReadsAndWrites(`node::traverse`) =
⟨{`node.left`,`node.right`,`node.position`, `node.force`,`node.mass`},
{`node.force`}⟩
ExternallyVisibleReadsAndWrites(`system::forces`) =
⟨{`node.left`,`node.right`,`node.position`, `node.force`,`node.mass`,
`system.num_bodies`, `system.l`,`system.r`},{`node.force`}⟩

ExtentConstantVariables(`system::forces`) =
{`node.left`,`node.right`,`node.position`, `node.mass`,
`system.num_bodies`, `system.l`,`system.r`}

Fig. 10. Results of externally visible reads and writes and extent constant variables computations.

ExtentAndFullAndAuxiliaryCallSites(m, *extent*, *full*, *aux*)
if ($m \notin$ *extent*)
extent = *extent* $\cup \{m\}$;
for all $c \in$ `callSites`(m)
m' = `method`(c);
$\langle rd, wr \rangle$ = ExternallyVisibleReadsAndWrites(m');
if $((wr \not\subseteq (CL \times V) \cup (CL \times Q \times V))$ **or** $(rd \not\subseteq (CL \times V) \cup (CL \times Q \times V) \cup P))$
aux = *aux* $\cup \{c\}$;
else
full = *full* $\cup \{c\}$;
$\langle$*extent*, *full*, *aux*$\rangle$ = ExtentAndFullAndAuxiliaryCallSites(m', *extent*, *full*, *aux*);
return $\langle$*extent*, *full*, *aux*$\rangle$;

Fig. 11. Full and auxiliary call site algorithm.

ExtentAndFullAndAuxiliaryCallSites(`system::forces`,∅,∅,∅) =
⟨{`node::interact`, `node::traverse`, `system::forces`}, {2,3,4,5},{1}⟩

Fig. 12. Results of full and auxiliary call site computation for `system::forces`.

4.4 Extent Full and Auxiliary Call Sites

The algorithm in Figure 11 performs a traversal of the call graph to compute the extent and the set of full and auxiliary call sites. The algorithm does not traverse edges in the call graph that correspond to auxiliary call sites. The union of the set of full call sites and the set of auxiliary call sites are the sets of call sites in the methods in the extent. Figure 12 presents the results of this computation for the `system::forces` method.

The ExtentAndFullAndAuxiliaryCallSites algorithm must satisfy several correctness conditions. The first correctness condition is that the produced extent must contain all full methods m' that can be invoked either directly or indirectly via a call chain of invocations of full methods that starts with an invocation of the method m. The second condition is that the extent contains only m or full methods (the CheckFullCallSites algorithm presented in Section 4.5 will check that m is a full method). The third condition is that the produced full

and auxiliary call site sets are disjoint and that their union is the set of all call sites in the full methods in the extent.

We outline a simple induction proof on the length of the call chain from an invocation of *m* to a potentially invoked full method *m′* that establishes the first condition. In the base case of the induction, the length of the call chain is zero; in other words, $m' = m$. In this case, the algorithm clearly inserts *m* into the extent. In the induction step, we assume that all full methods invoked at depth n of a call chain from *m* have been inserted into the extent. We show that all full methods invoked at depth $n + 1$ of a call chain are inserted into the extent. All methods invoked at depth $n + 1$ of a call chain are invoked from a call site in a method invoked at depth n.

A method is inserted into the extent if and only if ExtentAndFullAndAuxiliaryCallSites is called on the method. In this case, the algorithm visits each call site in the method, classifying it either as a full call site or as an auxiliary call site. If the call site is a full call site, the algorithm recursively calls itself on the method at the call site, and that method will be inserted into the extent. Therefore, the algorithm visits all of the call sites in the full methods that are invoked at depth n of a call chain, and it inserts all of the full methods that may be invoked at depth $n + 1$ of a call chain into the extent.

We next address the second correctness condition. The condition in the algorithm for recursively calling the ExtentAndFullAndAuxiliaryCallSites routine on a method *m′* ensures that it is called on *m′* only if *m′* is a full method. The ExtentAndFullAndAuxiliaryCallSites routine is therefore called only on *m* and for full methods. The extent therefore contains only *m* and full methods.

We next address the third correctness condition. Every time the algorithm inserts a method into the extent, the algorithm visits all of its call sites and inserts them into either the set of full call sites or the set of auxiliary call sites. Every call site in methods in the extent is classified either as a full call site or as an auxiliary call site. Furthermore, the condition that determines how call sites are classified ensures that no call site is classified as both a full call site and as an auxiliary call site.

4.5 Full Call Site Checks

The current symbolic execution algorithm uses extent constants to represent the values of variables passed by reference into full methods. To help ensure that this representation is correct, the CheckFullCallSites algorithm checks each full call site to make sure that the caller writes only extent constant values into the variables that are passed by reference into the method. Auxiliary methods are the only other methods that may write these variables. The CheckAuxiliaryCallSites algorithm presented in Section 4.6 ensures that if the reference parameters contain only extent constant values, the auxiliary methods will compute only extent constant values. The full call site checks therefore combine with the auxiliary call site checks to ensure that all variables passed by reference into either full or auxiliary methods contain only extent constant values.

The algorithm also checks that the method *m* that the commutativity analysis algorithm is attempting to parallelize is a full method.

4.6 Auxiliary Call Site Checks

To ensure that the symbolic execution operates correctly, the compiler performs several auxiliary call site checks. To help ensure that the symbolic execution builds expressions that correctly denote the new values of instance variables, the compiler checks that no

```
CheckFullCallSites(m, full, ec)
    ⟨rd, wr⟩ = ExternallyVisibleReadsAndWrites(m);
    if ((wr ⊈ (CL × V) ∪ (CL × Q × V)) or (rd ⊈ (CL × V) ∪ (CL × Q × V) ∪ P)) return false;
    for all c ∈ full
        if (dep(c) ⊈ ec) return false;
    return true;
```

Fig. 13. Algorithm to check full call sites.

```
CheckAuxiliaryCallSites(aux, ec)
    for all c ∈ aux
        m = method(c);
        ⟨rd, wr⟩ = ExternallyVisibleReadsAndWrites(m);
        rd = map(bind(c, I), rd);
        wr = map(bind(c, I), wr);
        if (rd ⊈ L ∪ ec or wr ⊈ L or dep(c) ⊈ ec) return false;
    return true;
```

Fig. 14. Algorithm to check auxiliary call sites.

auxiliary method writes an instance variable of the receiver. It enforces this constraint by requiring that, at each auxiliary call site, the set of externally visible variables that the auxiliary method writes includes only local variables of the caller.

The current symbolic execution algorithm uses extent constants to represent the values of variables passed by reference into auxiliary methods. The compiler checks several conditions to ensure that these variables hold only extent constant values:

—The set of externally visible variables that the auxiliary method reads must include only extent constant variables and its reference parameters.

—Before any auxiliary method is invoked, its reference parameters must contain extent constant values. The compiler enforces this constraint, in part, by checking that the caller writes only extent constant values into variables that are passed by reference into auxiliary methods. The other conditions ensure that other auxiliary methods are the only other methods that may write these variables. The caller site condition check, in combination with the other auxiliary call site checks, ensures that the variables passed by reference to auxiliary methods always contain extent constant values.

These conditions ensure that the entire executions of auxiliary methods are not visible outside the caller. It is therefore possible for the compiler to check only full methods for commutativity. Figure 14 presents the algorithm that checks that auxiliary call sites satisfy the conditions. It is a direct translation of the conditions presented above.

The fact that the symbolic execution uses extent constants to represent the values computed in auxiliary methods drives the conditions that auxiliary call sites must meet. An interprocedural symbolic execution algorithm would enable the compiler to extract a more precise representation of the values that auxiliary methods compute, which would allow the compiler to relax these conditions.

4.7 Separability

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

4.8 Commutativity Testing

The commutativity-testing algorithm presented in Figure 15 determines if two methods commute. The algorithm first applies a series of simple tests that determine if the two methods are independent. The first test relies on the type system; if the classes of the receivers of the two methods are different, then the two methods are guaranteed to be independent.[6] The second test analyzes the instance variable usage to check that neither method writes an instance variable that the other accesses. If the methods are independent, then they commute and the algorithm performs no further checks.

If the independence tests fail to determine that the methods are independent, the commutativity-testing algorithm checks if it can symbolically execute the methods. If it cannot symbolically execute the methods, the commutativity-testing algorithm conservatively assumes that the methods may not commute. Otherwise, the compiler generates a symbolic receiver and symbolic parameter values for the two methods, then symbolically executes the methods in both execution orders. It then simplifies the resulting expressions and compares corresponding expressions for equality. If corresponding expressions denote the same value, the operations commute.

Two routines deal with the symbolic execution. The routine Analyzable(m, aux, ec) determines if it is possible to symbolically execute a method. The actual symbolic execution is performed by the routine SymbolicallyExecute(m_1, m_2, aux, ec). 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 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.

We illustrate how the commutativity-testing algorithm works by applying it to the `system::forces` method. The compiler must check that all of the methods in its extent commute. Because `node::traverse` and `system::forces` compute only extent constant values and write only local variables, they commute with all methods in the extent. The compiler is left to check that `node::interact` commutes with itself. The compiler uses the symbolic commutativity-testing algorithm to check this property.

4.9 The Symbolic Commutativity Testing Algorithm

To test that methods commute, the compiler must reason about the new values of the receiver's instance variables and the multiset of operations directly invoked when the

[6]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 variable declared in a class from which its receiver's class inherits.

```
Commute(m1, m2, aux, ec)
    if (Independent(m1, m2)) return true;
    if (not Analyzable(m1, aux, ec)) return false;
    if (not 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 (not Compare(Simplify(i1(v)), Simplify(i2(v)))) return false;
    if (not Compare(Simplify(n1), Simplify(n2))) return false;
    return true;
```

Fig. 15. Commutativity-testing algorithm.

```
ex ∈ EX   ::=   EX ⊕ EX | ⊖EX | if (EX,EX,EX) | v | e
mx ∈ MX   ::=   EX->op(EX,...,EX) | if (EX,MX) | for (l = EX; l < EX; l += EX) MX |
                for (l = EX; l < EX; l++) MX
```

Fig. 16. Expressions for symbolic analysis.

methods execute. The compiler represents the new values and multisets of invoked methods using symbolic expressions. Figure 16 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 methods. The symbol $\oplus$ represents an arbitrary binary operator. The symbol $\ominus$ represents an arbitrary unary operator. The compiler uses EX expressions to represent instance variable values and multisets of MX expressions to represent invoked methods.

4.9.1 *Symbolic Execution.* The symbolic execution algorithm can operate successfully on only 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 methods by internally generating a new extent constant for each variable that is passed by reference into the method, then binding each variable to its extent constant.

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

```
for (l = 0; l < ex1; l++) v[l] = v[l] ⊕ ex2;
```

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

ec= ExtentConstantVariables(`system::forces`) =
{`node.left`,`node.right`,`node.position`, `node.mass`,
`system.num_bodies`, `system.l`,`system.r`}
ExtentAndFullAndAuxiliaryCallSites(`system::forces`,∅,∅,∅) =
⟨{`node::interact`,`node::traverse`,`system::forces`}, {2,3,4,5},*aux*⟩, where
aux = {`1`}
$\texttt{mx}_1$ = `r->interact(`$\texttt{m}_1$`,`$\texttt{d}_1$`)`
$\texttt{mx}_2$ = `r->interact(`$\texttt{m}_2$`,`$\texttt{d}_2$`)`

$\langle i_1,n_1\rangle$ = SymbolicallyExecute($\texttt{mx}_1$,$\texttt{mx}_2$,*aux*,*ec*), where
i_1 = [`force` ↦ `(force + (mass * `$\texttt{m}_1$`)/(`$\texttt{d}_1$` * `$\texttt{d}_1$`)) + (mass * `$\texttt{m}_2$`)/(`$\texttt{d}_2$` * `$\texttt{d}_2$`)]`
n_1 = ∅
$\langle i_2,n_2\rangle$ = SymbolicallyExecute($\texttt{mx}_2$,$\texttt{mx}_1$,*aux*,*ec*), where
i_2 = [`force` ↦ `(force + (mass * `$\texttt{m}_2$`)/(`$\texttt{d}_2$` * `$\texttt{d}_2$`)) + (mass * `$\texttt{m}_1$`)/(`$\texttt{d}_1$` * `$\texttt{d}_1$`)]`
n_2 = ∅

Fig. 17. Results of symbolic execution of two symbolic invocations of `node::interact`.

```
for (l = ex1; l <ex2; l += ex3) ex4->op(ex5,...,exn);
for (l = ex1; l <ex2; l++) ex4->op(ex5,...,exn);
```

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 could also replace unanalyzable loops with tail-recursive methods that perform the same computation.

In our example, the compiler determines that it must symbolically execute two symbolic invocations of `node::interact`. Figure 17 presents the results of the symbolic execution. In this figure, `r` represents the receiver, $\texttt{m}_1$ and $\texttt{d}_1$ represent the parameters for one symbolic operation, and $\texttt{m}_2$ and $\texttt{d}_2$ represent the parameters for the other symbolic operation.

4.9.2 *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 expression equality test.

The compiler currently applies simple arithmetic rewrite rules such as $\texttt{ex}_1\texttt{-ex}_2 \Rightarrow \texttt{ex}_1\texttt{+(-ex}_2\texttt{)}$, $\texttt{-(-ex)} \Rightarrow \texttt{ex}$ and $\texttt{ex}_1\texttt{*(ex}_2\texttt{+ex}_3\texttt{)} \Rightarrow \texttt{(ex}_1\texttt{*ex}_2\texttt{)+(ex}_1\texttt{*ex}_3\texttt{)}$. It also applies rules such as $\texttt{(ex}_1\texttt{+ex}_2\texttt{)+ex}_3 \Rightarrow \texttt{(ex}_1\texttt{+ex}_2\texttt{+ex}_3\texttt{)}$ 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 equivalent subexpressions. We have also developed rules for conditional and array expressions [Rinard and Diniz 1996].

In the worst case, the expression manipulation algorithms may take exponential running time. Like other researchers applying similar expression manipulation techniques in other analysis contexts [Blume and Eigenmann 1995], we have not observed this behavior in practice. Finally, it is undecidable in general to determine if two expressions always denote the same value [Ibarra et al. 1996]. We therefore focus on developing algorithms that work well for the cases that occur in practice.

Figure 18 presents the results of the expression simplification algorithm for the ex-

Initial Expressions:
i_1 = [force $\mapsto$ (force + (mass * m_1)/(d_1 * d_1)) + (mass * m_2)/(d_2 * d_2)]
i_2 = [force $\mapsto$ (force + (mass * m_2)/(d_2 * d_2)) + (mass * m_1)/(d_1 * d_1)]
Conversion into n-ary application of + operator:
i_1 = [force $\mapsto$ force + (mass * m_1)/(d_1 * d_1) + (mass * m_2)/(d_2 * d_2)]
i_2 = [force $\mapsto$ force + (mass * m_2)/(d_2 * d_2) + (mass * m_1)/(d_1 * d_1)]
Sorting operands of + operator:
i_1 = [force $\mapsto$ force + (mass * m_1)/(d_1 * d_1) + (mass * m_2)/(d_2 * d_2)]
i_2 = [force $\mapsto$ force + (mass * m_1)/(d_1 * d_1) + (mass * m_2)/(d_2 * d_2)]

Fig. 18. Expression simplification for two symbolic invocations of `node::interact`.

pressions generated during the symbolic execution of the two invocations of the method `node::interact`. 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 `system::forces` commute. It therefore marks `system::forces` as a parallel method.

4.9.3 *New Objects.* The current compiler classifies any method that creates new objects as unanalyzable. The primary motivation for this restriction is to simplify the implementation. It is possible to extend the symbolic execution to handle programs that create new objects, although there are some complications.

To symbolically execute a statement that creates a new object, the compiler must create an expression that denotes the new object. We call such an expression a *new object name*. New object names must be different from any expression that denotes any other object. In particular, they must be different from new object names that denote other new objects.

The compiler tests if two methods commute by comparing expressions generated during the symbolic execution. The current compiler simply tests if the expressions are equivalent. But the presence of new object names introduces some additional complexity. The result of the program does not depend on where the memory allocator places objects. Operations therefore commute if corresponding expressions are equivalent under any one-to-one mapping from the new object names in one expression to the new object names in the other expression. An effective expression comparison algorithm must therefore construct and maintain a mapping between new object names as it compares expressions.

5. CODE GENERATION

According to the compiler's analysis, every method in the program is classified as either a parallel method or a serial method. Operations invoked by a serial method execute sequentially: each invoked operation completes its execution before the next invoked operation starts to execute. Operations invoked by a parallel method execute concurrently.

Each parallel method has two versions: a serial version and a parallel version. The parallel version invokes operations and returns without waiting for the operations to complete their executions. Furthermore, the invoked operations execute in parallel. The serial version simply invokes the parallel version, then waits for the entire parallel execution to

complete.

When a serial method invokes a parallel method, it invokes the serial version. The serial version then triggers a transition from sequential to parallel execution by invoking the parallel version. The transition from parallel back to sequential execution also takes place inside the serial version: when the serial version returns, the parallel computation has completed.

To generate code for the parallel version of a parallel method, 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 (but see Sections 5.2 and 5.3). Auxiliary methods 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 iteration of the loop. The generated code can then apply standard parallel loop execution techniques; it currently uses guided self-scheduling [Polychronopoulos and Kuck 1987].

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 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 into 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 full call sites, it invokes the mutex version. This code generation strategy also ensures that operations observe the correct values of local variables that multiple invocations of auxiliary methods write.

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 [Diniz and Rinard 1996].

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 three applications: the Barnes-Hut hierarchical N-body solver [Barnes and Hut 1986], the Water simulation code [Singh et al. 1992], and the String seismic code [Harris et al. 1990]. Explicitly parallel versions of the first two applications are available in the SPLASH [Singh et al. 1992] and SPLASH-2 [Woo et al. 1995] benchmark suites. We have developed an explicitly parallel version of String using the ANL macro package [Lusk et al. 1987]. This section presents performance results for the automatically parallelized and explicitly parallel versions of these applications on a 16-processor Stanford DASH machine [Lenoski 1992] 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++ [Bodin et al. 1994] as a front end. The analysis and code generation phases consist of approximately 21,000 lines of C++ code. This count includes no code from the Sage++ system. The generated parallel code contains calls to a run-time library that provides the basic concurrency management and synchronization functionality. The library consists of approximately 6000 lines of C code.

The current version of the compiler imposes several restrictions on the dialect of C++ that it can analyze. 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 following:

—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.

The current compiler requires the entire program, with the exception of standard library routines, to appear in a single file. It would be straightforward to relax this constraint to allow separate compilation. To parallelize a given method, however, the envisioned compiler would still require that all of the methods that could be invoked as a result of executing the given method appear in the same file. It would require significant changes for the compiler to allow these methods to appear in separate files.

The current system starts with an unannotated, sequential C++ program. It then runs a preprocessor that replaces the standard include files with our own include files. The standard include files contain external declarations of methods, classes, and functions. Our include files contain empty definitions (not declarations) of these entities. The motivation for this replacement is that Sage++ does not handle the external declarations in the standard include files. At the end of the compilation, the compiler undoes the replacement so that the running parallel program uses the standard include files and libraries.

The preprocessor generates another unannotated, sequential C++ program. The system runs the `pC++2dep` tool from the Sage++ package on this file. This tool parses the C++ file and generates a *dep* file. The dep file contains a representation of the program in the Sage++ internal representation, which is based on abstract syntax trees.

The system then reads in the dep file and translates the Sage++ internal representation to our own internal representation. Our representation is designed to support the symbolic execution and expression manipulation algorithms. We also build an explicit call graph. The compiler uses these data structures to perform the analysis as described in Section 4. The implemented system also recognizes standard library functions such as `sqrt`, `pow`, etc. The symbolic execution generates expressions that contain invocations of these functions.

The implemented compiler does not require the entire program to conform to the model of computation described in Section 3. If part of the program fails to conform to the model, the compiler simply marks it as unanalyzable and does not try to parallelize it. The presence of unanalyzable code in one part of the program does not affect the ability of the compiler to parallelize other parts of the program.

The analysis phase builds data structures that control the generation of parallel code. These data structures refer back to the original Sage++ data structures. The code generation traverses these data structures to generate the different versions of parallel methods. The compiler uses a configuration file to control several code generation options. For our benchmark programs, all of the compiler flags are the same. The end result of the code generation is an explicitly parallel C++ program that contains calls to the run-time library. It is possible to use any standard C++ compiler to generate machine code for this parallel program.

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. And although Barnes-Hut 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 [Salmon 1990; Singh 1993]. 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.

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 serial 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 [Singh 1993]. 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 the 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.

We used the compiler to generate a parallel C++ version of the program. The compiler performed the complete parallelization automatically; we performed none of the analysis, transformations, or code generation by hand.

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 four 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 two parallel loops. Table II presents several analysis statistics. For each parallel phase, it presents the number of auxiliary call sites in the phase, the number of full methods in the phase, the number of independent pairs of methods in the phase, and the number of pairs that the compiler had to symbolically execute. All of the parallel phases have a significant number of auxiliary call sites; the compiler would be unable to parallelize any of the phases if the commutativity-testing phase included the auxiliary methods. Most of the pairs of invoked methods in the phases are always independent, which means that the compiler has to symbolically execute

relatively few pairs.

Table II. Analysis Statistics for Barnes-Hut

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

6.2.3 *Compilation Time.* We report compilation times for the compiler running on a `Sun Microsystems Ultra I` computer system with 64 megabytes of main memory. To compile Barnes-Hut, it takes 0.037 seconds to load in the data structures from the dep file, 0.648 seconds to perform the analysis, and 0.77 seconds to generate the parallel code. 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 III 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 19 presents the *speedup* as a function of the number of processors executing the computation. The speedup is the execution time of the serial C version divided by the execution time of the parallel version. The computation scales reasonably well, exhibiting speedups between 11 and 12 out of 16 processors.

Table III. Execution Times for Barnes-Hut (seconds)

Number of Bodies	Processors						
	Serial	1	2	4	8	12	16
8192	66.5	67.7	34.1	17.0	9.9	7.2	5.9
16384	147.8	149.9	76.3	37.8	21.9	15.6	12.9

We start our analysis of the performance with the parallelism coverage [Hall et al. 1995], which measures the amount of time 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

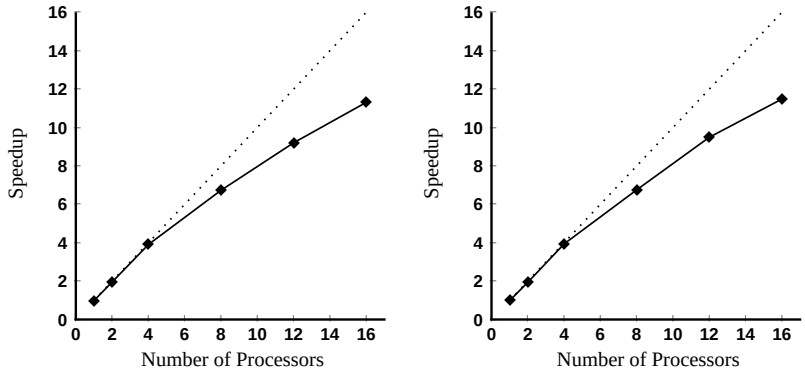

Fig. 19. Speedup for Barnes-Hut.

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 IV presents the parallelism coverage for Barnes-Hut; these statistics show that the compiler is able to parallelize almost all of the computation.

Table IV. Parallelism Coverage for Barnes-Hut

Number of Bodies	Serial Compute Time (seconds)	Time in Parallelized Sections (seconds)	Parallelism Coverage
8192	66.52	66.21	98.03%
16384	147.76	145.06	98.17%

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 [Tseng 1995]. 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 that the compiler automatically inserts into methods.

We developed a benchmark program to measure the cost of each source of overhead. Table V presents the results. The loop overhead increases with the number of processors; the table presents the loop overhead on 16 processors.

Table V. Parallel Construct Overhead on 16-Processors (microseconds)

Loop Overhead On 16 Processors	Chunk Overhead	Iteration Overhead	Lock Overhead
171.4	27.85	0.39	4.75

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. Table VI presents the mean sizes of the pieces of useful work for Barnes-Hut. 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 V 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 on 16 processors. A comparison with the Chunk Overhead in Table V 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 V 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 V shows that the amortized lock overhead is negligible.

Table VI. Granularities for Barnes-Hut (microseconds)

Number of Bodies	Loop Size	Chunk Size on 16 Processors	Iteration Size	Task Size
8192	22.07×10^6	182.40×10^3	2.69×10^3	2.69×10^3
16384	48.35×10^6	366.31×10^3	2.95×10^3	2.95×10^3

We instrumented the generated parallel code to measure how much time each processor spends in different parts of the parallel computation. This instrumentation makes use of a

low-overhead timer on the DASH machine. The run-time system measures the amount of time spent in each section of the program by reading the timer at the beginning and end of the section. This instrumentation breaks the execution time down into the following categories:

—*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 vary dramatically with the number of processors executing the computation.

—*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. Unlike all of the other components of the time breakdown, we measure this component using program-counter sampling [Graham et al. 1982; Knuth 1971]: we found that using the timer to directly measure this component significantly perturbed the performance results.[7]

—*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 20 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

[7]The key problem is that using the timer to measure the blocked component introduces an additional read of the timer into the critical section implemented by the acquired lock. This instrumentation can significantly increase the amount of time the processor spends holding the lock. Water is very sensitive to sizes of its critical regions, and we found that using the timer to measure the blocked component significantly degraded the overall performance of this application. We validated the use of program-counter sampling by using it to measure the parallel idle component, which does not suffer from the instrumentation effects described above. We found that, for all parallel programs and for all data sets, there was an excellent quantitative correlation between the parallel idle time measured with program-counter sampling and the directly measured parallel idle time: the two measurements never varied by more than 5%. These results give us confidence in the values obtained for the blocked component using program-counter sampling.

the number of processors increases.[8]

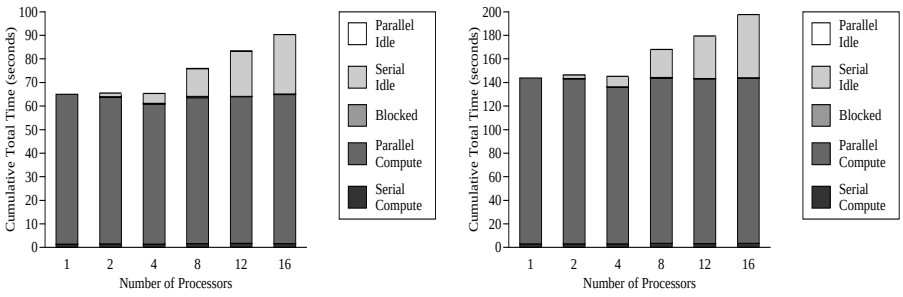

Fig. 20. Cumulative time breakdowns for Barnes-Hut.

These graphs show that the limit on the performance is the time spent in the serial phases of the computation: at 16 processors, the serial idle time accounts for approximately 30% of the cumulative compute time for both applications.

6.2.5 *Comparison with the Explicitly Parallel Version.* Table VII 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 16 processors it runs 37% faster for 8192 bodies and 25% 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 partitioning and scheduling algorithm called `Costzones` in the force computation phase [Singh 1993]. This algorithm provides better locality than the guided self-scheduling algorithm in the automatically parallelized version.

Number of Bodies	Processors					
	1	2	4	8	12	16
8192	70.8	34.4	16.3	8.2	5.5	4.3
16384	155.8	76.4	35.7	18.3	12.2	10.3

Table VII. Execution Times for Explicitly Parallel Barnes-Hut (seconds)

[8]There are some small discrepancies between the time breakdowns in Figure 20 and the execution times in Table III. We attribute these discrepancies to the fact that the performance numbers come from different executions; the time breakdowns come from a fully instrumented version of the code while the execution times come from a minimally instrumented version.

6.3 Water

Water computes the energy potential of a set of water molecules in the liquid state. The main data structure is an array of molecule objects. Almost all of the compute time is spent in two $O(N^2)$ phases, where N is the number of molecules. 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 [Berry et al. 1989]. 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.

We used the compiler to generate a parallel C++ version of the program. The compiler performed the complete parallelization automatically; we performed none of the analysis, transformations, or code generation by hand.

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. 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 VIII contains the analysis statistics. As for Barnes-Hut, all of the phases contain auxiliary call sites, and most of the pairs in the extents are independent.

Table VIII. Analysis Statistics for Water

Parallel Phase	Auxiliary 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

6.3.3 *Compilation Time.* For Water, the compiler takes 0.087 seconds to load the intermediate format, 1.443 seconds to perform the analysis, and 1.220 seconds to generate the parallel code.

6.3.4 *Performance Results and Analysis.* Table IX 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 21 presents the speedup curves. Water performs reasonably well, achieving a speedup of over eight out of 16 processors for both data sets.

Table IX. Execution Times for Water (seconds)

Number of Molecules	Processors						
	Serial	1	2	4	8	12	16
343	76.8	82.9	40.5	20.7	12.8	10.0	9.2
512	165.8	175.8	88.4	44.3	26.4	21.1	19.5

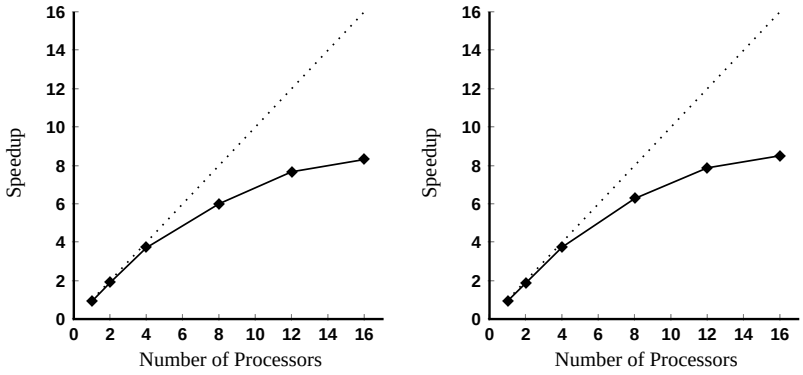

Fig. 21. Speedup for Water.

Table X, which presents the parallelism coverage for this application, shows that the compiler parallelizes almost all of the computation. Table XI 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.

Figure 22, which presents the cumulative time breakdowns for Water, clearly shows why Water fails to scale beyond 12 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. For this application 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.

Table X. Parallelism Coverage for Water

Number of Molecules	Serial Compute Time (seconds)	Time in Parallelized Sections (seconds)	Parallelism Coverage
343	76.85	75.87	98.73%
512	165.82	164.05	98.94%

Table XI. Granularities for Water (microseconds)

Number of Molecules	Loop Size	Chunk Size on 16 Processors	Iteration Size	Task Size
343	3.79×10^6	52.69×10^3	11.07×10^3	80.49
512	8.20×10^6	105.16×10^3	16.03×10^3	78.15

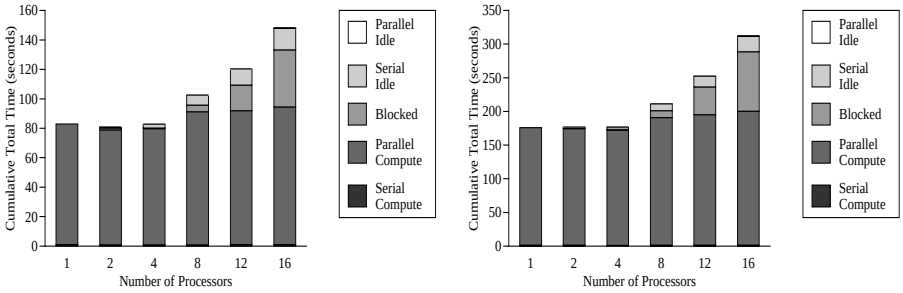

Fig. 22. Cumulative time breakdowns for Water.

6.3.5 *Comparison with the Explicitly Parallel Version.* The SPLASH parallel benchmark set contains an explicitly parallel version of Water; Table XII contains the execution times for this version. Unlike the automatically parallelized version, the explicitly parallel version scales reasonably well to 16 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.

Table XII. Execution Times for Explicitly Parallel Water (seconds)

Number of Molecules	Processors					
	1	2	4	8	12	16
343	74.2	37.3	18.2	9.5	6.7	5.5
512	161.1	81.4	40.1	20.8	14.3	12.1

6.4 String

String uses seismic travel-time inversion to construct a two-dimensional discrete velocity model of the geology between two oil wells. Each element of the velocity model records how fast sound waves travel through the corresponding part of the geology. The seismic

data are collected by firing non destructive wave sources in one well and recording the waves digitally as they arrive at the other well. The travel times of the waves can be measured from the resulting seismic traces. The application uses the travel-time data to iteratively compute the velocity model. The computationally intensive phase of the application traces rays from one well to the other. The velocity model determines both the simulated path and the simulated travel time of each ray. The computation records the difference between the simulated and the measured travel times and backprojects the difference linearly along the path of the ray. At the end of the phase, the computation uses the backprojected differences to construct an improved velocity model. The process continues for a specified number of iterations. The serial computation stores the velocity model in a one-dimensional array and the backprojected differences in another one-dimensional array. Each element of the difference array stores the running sum of the backprojected differences for the corresponding element of the velocity model. All updates to the difference array commute.

The analysis finds a total of five parallel loops. Two of the loops are nested inside other parallel loops, so the generated parallel version contains three parallel loops. The important parallel loop is the outermost loop in the phase that traces all the rays. This loop generates all of the ray computations. For each ray, the computation determines whether or not the ray falls within a given aperture angle and, if so, computes its propagation time through the model. This computation generates multiple updates to the data structure that records the difference between the simulated and measured travel times. The update operations use synchronization constructs to make the update atomic; this is the only synchronization in the parallel ray-tracing phase.

6.4.1 *Application Statistics.* The final C++ version consists of approximately 2100 lines of code. The serial C version consists of approximately 2000 lines of code. The explicitly parallel version consists of approximately 2400 lines of C code.

The analysis finds a total of five parallel loops. Two of the loops are nested inside other parallel loops, so the generated parallel version contains three parallel loops. Table XIII contains the analysis statistics. As for Barnes-Hut and Water, all of the phases contain auxiliary call sites, and most of the pairs in the extents are independent.

Table XIII. Analysis Statistics for String

Parallel Phase	Auxiliary Call Sites	Extent Size	Independent Pairs	Symbolically Executed Pairs
Project Forward	2	7	26	2
Project Backward	2	7	26	2
Slowness	1	4	8	2

6.4.2 *Compilation Time.* For String, the compiler takes 0.039 seconds to load the intermediate format, 1.266 seconds to perform the analysis, and 0.834 seconds to generate the parallel code.

6.4.3 *Performance Results and Analysis.* Table XIV contains the execution times for String. The measured computation includes initial and final I/O. We have collected performance results for this application for two input data sets, i.e., the small and big data sets. The small data set does not correspond to a realistic production work load; it is instead

designed solely for testing purposes. The big data set is representative of the production data sets used for this application. Figure 23 presents the speedup curves. While String performs very well for the big data set, for which is exhibits almost linear speedup, it does not scale beyond four processors for the small data set.

Table XIV. Execution Times for String (seconds)

Data Set	Processors						
	Serial	1	2	4	8	12	16
small	36.4	43.4	23.9	13.9	9.3	8.0	8.1
big	2181.3	2651.1	1333.8	663.9	337.1	230.2	171.6

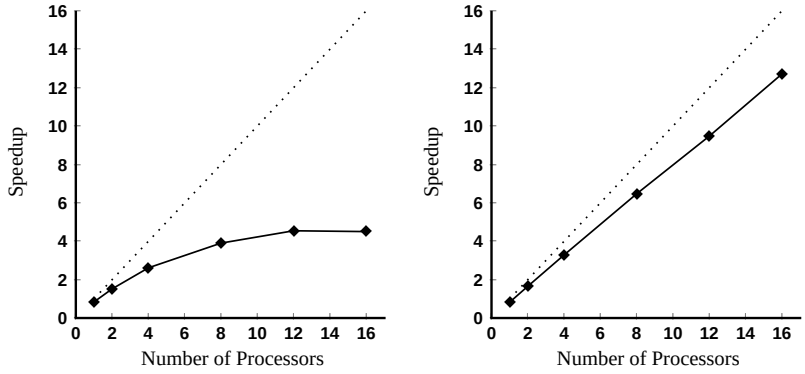

Fig. 23. Speedup for String.

Table XV, which presents the parallelism coverage for this application, shows that the compiler parallelizes almost all of the computation for the big data set. For the small data set the coverage is about 80%, which imposes a maximum speedup of 5. Table XVI shows that all of the sources of overhead are negligible except for the lock overhead.

Figure 24, which presents the cumulative time breakdowns for String, clearly shows why String fails to scale beyond four processors for the small data set. The fact that the serial computation component is a significant portion of the total computation time for four processors (approximately 30%) indicates that lack of parallelism is the primary source of the lack of scalability for this data set.

6.4.4 *Comparison with the Explicitly Parallel Version.* We have developed an explicitly parallel version of String using the ANL macro package [Lusk et al. 1987]. Table XVII contains the execution times for this version. For the small data set, the explicitly parallel version does not scale beyond four processors. We attribute this lack of performance to a lack of parallelism in the small data set. For the big data set, the explicitly parallel version scales perfectly to 16 processors, outperforming the automatically parallelized version. We attribute this difference to the fact that the explicitly parallel version replicates several

Table XV. Parallelism Coverage for String

Data Set	Serial Compute Time (seconds)	Time in Parallelized Sections (seconds)	Parallelism Coverage
small	36.4	29.2	80.34%
big	2478.3	2468.4	99.60%

Table XVI. Granularities for String (microseconds)

Data Set	Loop Size	Chunk Size on 16 Processors	Iteration Size	Task Size
small	14.60×10^6	202.78×10^3	37.92×10^3	78.89
big	411.40×10^6	5.71×10^6	1.07×10^6	81.50

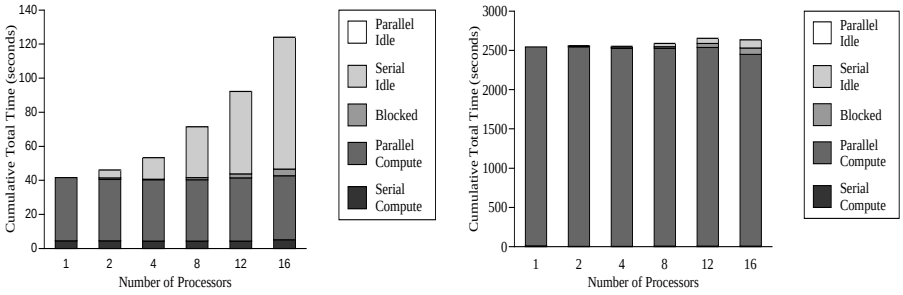

Fig. 24. Cumulative time breakdowns for String.

updated data structures. This replication eliminates two sources of overhead. First, because each processor updates its own replica, the program does not need to execute locking constructs to make the updates execute atomically. Second, the replication eliminates any contention for shared objects.

Table XVII. Execution Times for Explicitly Parallel String (seconds)

Data Set	Processors					
	1	2	4	8	12	16
small	36.9	20.8	13.8	10.7	11.5	14.4
big	2119.1	1197.6	594.7	301.1	194.4	160.8

6.5 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 [Berry et al. 1989; Blume and Eigenmann 1992].

For all of our 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 the 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, Water, and String as benchmark applications, 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 article suggest many possible directions for future research. 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 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 commutativity analysis to automatically generate parallel code for algorithms such as the tree construction algorithm in Barnes-Hut. The current technique works well for algorithms that traverse pointer-based data structures. The compiler may need to extend the technique to recognize operations that commute relative to the rest of the computation if it is to effectively parallelize algorithms that build pointer-based data strcutures.

7.2 Analysis Granularity

Our experience with auxiliary methods 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 methods into their callers. We expect a generalized concept of auxiliary methods 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 partitioning 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 holds only 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 [Rinard 1994a; Scales and Lam 1994]. 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 [Hiranandani et al. 1992]. 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 [Carlisle and Rogers 1995; Rinard 1995].

8. RELATED WORK

One of the very first papers in the field of parallelizing compilers identifies the concept of commuting computations as distinct from and more general than the concept of computations that can execute concurrently [Bernstein 1966]. The commutativity conditions, however, were formulated in terms of the variables that computations read and write and were therefore very restrictive. In effect, the conditions relaxed the independence conditions to allow commuting computations to write the same variable as long as no succeeding

computation read the variable. Furthermore, that paper did not identify the possibility of exploiting commutativity to parallelize computations.

The conditions presented in this article are much less restrictive. They require only that commuting operations generate the same values in both execution orders: there is no other restriction on how they read and write data.

Despite the early recognition of the concept of commuting computations, parallelizing compiler research has focused almost exclusively on data dependence analysis and, to a lesser extent, reduction analysis. The remainder of this section discusses related work in symbolic execution and 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 Symbolic Execution

Symbolic execution is a classic technique in computer science [Clarke and Richardson 1981]. It has been applied to a wide range of problems, including program testing [Douglas and Kemmerer 1994; King 1976], program verification [Dillon 1987a; 1987b], program reduction [King 1981], and program optimization [Darlington 1972; Urschler 1974]. We use symbolic execution as a tool to enable the application of commutativity analysis to object-based programs.

Constructs that complicate the application of symbolic execution include pointers, arrays and **while** loops. Our compiler, however, uses symbolic execution in a very structured context, which eliminates some of the complications. For example, the compiler uses symbolic execution only for pairs of symbolic operations that access the same object. There is no need to deal with accesses through arbitrary pointers. Extent constants allow the compiler to accurately represent values computed in auxiliary methods, even if it is not possible to symbolically execute the auxiliary methods.

It is also possible for the compiler to avoid problems caused by loops. If the compiler encountered a loop that it could not symbolically execute, it could simply replace the loop with a symbolically executable tail-recursive method. It would then perform the commutativity testing at the granularity of the individual loop iterations.

8.2 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 [Chase et al. 1990; Hendren et al. 1992; Larus and Hilfinger 1988; Plevyak et al. 1993]. 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 [Hendren et al. 1992; Larus and Hilfinger 1988]. 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 graphs 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.

8.3 Reductions

Several existing compilers can recognize when a loop performs a reduction of many values into a single value [Callahan 1991; Fisher and Ghuloum 1994; Ghuloum and Fisher 1995; Pinter and Pinter 1991]. 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 [Hall et al. 1995].

There are interesting connections between reduction analysis and commutativity analysis. Many (but not all) of the computations that commutativity analysis is designed to parallelize 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.

8.4 Commuting Operations in Parallel Languages

Steele [1990] describes an explicitly parallel computing framework that includes primitive commuting operations such as the addition of a number into an accumulator. The motivation is to deliver a flexible system for parallel computing that guarantees deterministic execution. He describes an enforcement mechanism that dynamically detects potential determinism violations 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 [Rinard 1994a; 1994b].

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 implementation must check only 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 [Lam and Rinard 1991]. 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.

Lengauer and Hehner [1982] propose a programming methodology in which the programmer exposes concurrency by declaring semantic relations between different parts of the program. One of these relations is that the parts of the program commute. The goal is to help the implementation exploit concurrency while ensuring that the parallelization does not change the semantics of the program.

Many concurrent object-oriented languages support the notion of mutually exclusive operations on objects [Chandra et al. 1993; Yonezawa et al. 1986]. 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 if 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 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 technique. We have used this compiler to automatically parallelize three 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.

ACKNOWLEDGMENTS

We wish to thank the anonymous referees for their many useful comments and suggestions.

REFERENCES

BANERJEE, U. 1988. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, MA.

BANERJEE, U., EIGENMANN, R., NICOLAU, A., AND PADUA, D. 1993. Automatic program parallelization. *Proceedings of the IEEE 81,* 2 (Feb.), 211–243.

BARNES, J. AND HUT, P. 1986. A hierarchical O(NlogN) force calculation algorithm. *Nature 324*, 4 (Dec.), 446–449.

BERNSTEIN, A. J. 1966. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers 15*, 5 (Oct.), 757–763.

BERRY, M., CHEN, D., KOSS, P., KUCK, D., LO, S., PANG, Y., POINTER, L., ROLOFF, R., SAMEH, A., CLEMENTI, E., CHIN, S., SCHNEIDER, D., FOX, G., MESSINA, P., WALKER, D., HSIUNG, C., SCHWARZMEIER, J., LUE, K., ORSZAG, S., SEIDL, F., JOHNSON, O., GOODRUM, R., AND MARTIN, J. 1989. 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.

BLUME, W. AND EIGENMANN, R. 1992. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *IEEE Transactions on Parallel and Distributed Systems 3*, 6 (Nov.), 643–656.

BLUME, W. AND EIGENMANN, R. 1995. Symbolic range propagation. In *Proceedings of the 9th International Parallel Processing Symposium*. IEEE Computer Society Press, Santa Barbara, CA, 357–363.

BODIN, F., BECKMAN, P., GANNON, D., GOTWALS, J., NARAYANA, S., SRINIVAS, S., AND WINNICKA, B. 1994. Sage++: An object-oriented toolkit and class library for building Fortran and C++ structuring tools. In *Proceedings of the Object-Oriented Numerics Conference*. Sunriver, Oregon.

CALLAHAN, D. 1991. Recognizing and parallelizing bounded recurrences. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, Santa Clara, CA, 169–184.

CARLISLE, M. AND ROGERS, A. 1995. Software caching and computation migration in Olden. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, Santa Barbara, CA, 29–38.

CHANDRA, R., GUPTA, A., AND HENNESSY, J. 1993. Data locality and load balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, San Diego, CA.

CHASE, D., WEGMAN, M., AND ZADEK, F. 1990. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*. ACM Press, White Plains, NY, 296–310.

CLARKE, L. AND RICHARDSON, D. 1981. Symbolic evaluation methods for program analysis. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Englewood Cliffs, NJ, 79–101.

DARLINGTON, J. 1972. A semantic approach to automatic program improvement. Ph.D. thesis, University of Edinburgh.

DILLON, L. 1987a. Verification of Ada tasking programs using symbolic execution, Part I: Partial Correctness. Tech. Rep. TRCS87-20, Dept. of Computer Science, University of California at Santa Barbara. Oct.

DILLON, L. 1987b. Verification of Ada tasking programs using symbolic execution, Part II: General Safety Properties. Tech. Rep. TRCS87-21, Dept. of Computer Science, University of California at Santa Barbara. Oct.

DINIZ, P. AND RINARD, M. 1996. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, San Jose, CA, 285–299.

DOUGLAS, J. AND KEMMERER, R. 1994. Aslantest: A symbolic execution tool for testing Aslan formal specifications. In *1994 International Symposium on Software Testing and Analysis*. Seattle, WA, 15–27.

EIGENMANN, R., HOEFLINGER, J., LI, Z., AND PADUA, D. 1991. Experience in the automatic parallelization of four Perfect Benchmark programs. In *Languages and Compilers for Parallel Computing, Fourth International Workshop*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Springer-Verlag, Santa Clara, CA.

EMAMI, M., GHIYA, R., AND HENDREN, L. 1994. 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*. ACM Press, Orlando, FL, 242–256.

FISHER, A. AND GHULOUM, A. 1994. Parallelizing complex scans and reductions. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*. ACM Press, Orlando, FL, 135–144.

GHULOUM, A. AND FISHER, A. 1995. Flattening and parallelizing irregular, recurrent loop nests. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, Santa Barbara, CA, 58–67.

GIFFORD, D., JOUVELOT, P., LUCASSEN, J., AND SHELDON, M. 1987. FX-87 Reference Manual. Tech. Rep. MIT/LCS/TR-407, MIT. Sept.

GRAHAM, S., KESSLER, P., AND MCKUSICK, M. 1982. gprof: A call graph execution profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*. Boston, MA.

HALL, M., AMARASINGHE, S., MURPHY, B., LIAO, S., AND LAM, M. 1995. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*. IEEE Computer Society Press, San Diego, CA.

HAMMEL, R. AND GIFFORD, D. 1988. FX-87 Performance Measurements: Dataflow Implementation. Tech. Rep. MIT/LCS/TR-421, MIT. November.

HARRIS, J., LAZARATOS, S., AND MICHELENA, R. 1990. Tomographic string inversion. In *60th Annual International Meeting, Society of Exploration and Geophysics, Extended Abstracts*. 82–85.

HENDREN, L., HUMMEL, J., AND NICOLAU, A. 1992. 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*. ACM Press, San Francisco, CA.

HENDREN, L., HUMMEL, J., AND NICOLAU, A. 1994. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*. ACM Press, Orlando, FL.

HIRANANDANI, S., KENNEDY, K., AND TSENG, C. 1992. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM 35*, 8 (Aug.), 66–80.

IBARRA, O., DINIZ, P., AND RINARD, M. 1996. On the complexity of commutativity analysis. In *Proceedings of the 2nd Annual International Computing and Combinatorics Conference*. Vol. 1090. Springer-Verlag, Hong Kong, 323–332.

KEMMERER, R. AND ECKMANN, S. 1985. UNISEX: a UNIx-based Symbolic EXecutor for Pascal. *Software—Practice and Experience 15*, 5 (May), 439–458.

KING, J. 1976. Symbolic execution and program testing. *Communications of the ACM 19*, 7 (July), 385–394.

KING, J. 1981. Program reduction using symbolic execution. *ACM SIGPLAN Notices 6*, 1 (Jan.), 9–14.

KNUTH, D. 1971. An empirical study of FORTRAN programs. *Software—Practice and Experience 1*, 105–133.

LAM, M. AND RINARD, M. 1991. Coarse-grain parallel programming in Jade. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, Williamsburg, VA, 94–105.

LAMPSON, B. W. AND REDELL, D. D. 1980. Experience with processes and monitors in Mesa. *Communications of the ACM 23*, 2 (February), 105–117.

LANDI, W., RYDER, B., AND ZHANG, S. 1993. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN '93 Conference on Program Language Design and Implementation*. ACM Press, Albuquerque, NM.

LARUS, J. AND HILFINGER, P. 1988. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Program Language Design and Implementation*. ACM Press, Atlanta, GA.

LENGAUER, C. AND HEHNER, E. 1982. A methodology for programming with concurrency: An informal presentation. *Science of Computer Programming 2*, 1 (Oct.), 1–18.

LENOSKI, D. 1992. The design and analysis of DASH: A scalable directory-based multiprocessor. Ph.D. thesis, Stanford, CA.

LUSK, E., OVERBEEK, R., BOYLE, J., BUTLER, R., DISZ, T., GLICKFIELD, B., PATTERSON, J., AND STEVENS, R. 1987. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc.

MOHR, E., KRANZ, D., AND HALSTEAD, R. 1990. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*. 185–197.

PINTER, S. AND PINTER, R. 1991. Program optimization and parallelization using idioms. In *Proceedings of the Eighteenth Annual ACM Symposium on the Principles of Programming Languages*. ACM Press, Orlando, FL, 79–92.

PLEVYAK, J., KARAMCHETI, V., AND CHIEN, A. 1993. Analysis of dynamic structures for efficient parallel execution. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, Portland, OR.

POLYCHRONOPOULOS, C. AND KUCK, D. 1987. Guided self-scheduling: A practical scheduling scheme for parallel computers. *IEEE Transactions on Computers 36*, 12 (Dec.), 1425–1439.

PUGH, W. AND WONNACOTT, D. 1992. Eliminating false data dependences using the Omega test. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*. ACM Press, San Francisco, CA.

RINARD, M. 1994a. The design, implementation and evaluation of Jade, a portable, implicitly parallel programming language. Ph.D. thesis, Stanford, CA.

RINARD, M. 1994b. Implicitly synchronized abstract data types: Data structures for modular parallel programming. In *Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software and Applications*, M. Furnari, Ed. World Scientific Publishing, Capri, Italy, 259–274.

RINARD, M. 1995. Communication optimizations for parallel computing using data access information. In *Proceedings of Supercomputing '95*. IEEE Computer Society Press, San Diego, CA.

RINARD, M. AND DINIZ, P. 1996. Commutativity analysis: A technique for automatically parallelizing pointer-based computations. In *Proceedings of the 10th International Parallel Processing Symposium*. IEEE Computer Society Press, Honolulu, HI, 14–22.

SALMON, J. K. 1990. Parallel hierarchical N-body methods. Ph.D. thesis, California Institute of Technology.

SCALES, D. AND LAM, M. S. 1994. 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*. ACM Press, Monterey, CA.

SINGH, J. 1993. Parallel hierarchical N-body methods and their implications for multiprocessors. Ph.D. thesis, Stanford University.

SINGH, J., WEBER, W., AND GUPTA, A. 1992. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News 20*, 1 (March), 5–44.

STEELE, G. 1990. Making asynchronous parallelism safe for the world. In *Proceedings of the Seventeenth Annual ACM Symposium on the Principles of Programming Languages*. ACM Press, San Francisco, CA, 218–231.

TSENG, C. 1995. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, Santa Barbara, CA, 144–155.

URSCHLER, G. 1974. Complete redundant expression elimination in flow diagrams. Research Report RC 4965, IBM Yorktown Heights. Aug.

WEIHL, W. 1988. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers 37*, 12 (Dec.), 1488–1505.

WILSON, R. AND LAM, M. 1995. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*. ACM Press, La Jolla, CA.

WOO, S., OHARA, M., TORRIE, E., SINGH, J., AND GUPTA, A. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*. ACM Press, Santa Margherita Ligure, Italy.

YONEZAWA, A., BRIOT, J.-P., AND SHIBAYAMA, E. 1986. Object oriented concurrent programming in ABCL/1. In *Proceedings of the OOPSLA-86 Conference*. ACM Press, Portland OR, 258–268.

Received May 1996; revised November 1996; accepted February 1997

APPENDIX

An appendix to this article is available in electronic form (PostScript™). Any of the following methods may be used to obtain it; or see the inside back cover of a current issue for up-to-date instructions.

—By anonymous ftp from **acm.org**, file **[pubs.journals.toplas.append]p1813.ps**

—Send electronic mail to **mailserve@acm.org** containing the line
send [anonymous.pubs.journals.toplas.append]p1813.ps

—By *Gopher* from **acm.org**

—By anonymous ftp from **ftp.cs.princeton.edu**, file **pub/toplas/append/p1813.ps**

—Hardcopy from *Article Express*, for a fee: phone 800-238-3458, fax +1-516-997-0890, or write 469 Union Avenue, Westbury NY 11550; and request ACM-TOPLAS-APPENDIX-1813.

THIS DOCUMENT IS THE APPENDIX TO THE FOLLOWING PAPER:

Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers

MARTIN C. RINARD
Massachusetts Institute of Technology
and
PEDRO C. DINIZ
University of Southern California / Information Sciences Institute

---

This appendix contains information on how to generate parallel executable programs for the three applications studied in this article, `barnes`, `water` and `string`. We describe how to run each application, what files are generated, and how to interpret the output files. For comparison purposes we have also provided, for each application, a sample of output files we have obtained while running these parallel applications on the Stanford DASH shared-memory multiprocessor.

This appendix folder is divided into two main folders, respectively `rts` and `apps`. The `rts` folder contains the C source and include files for the run time system that supports the execution of our compiler generated parallel programs. The `apps` folder has one folders for each of the applications, respectively, `barnes`, `water` and `string`. For each of these applications we have a folder with the source files, input files and output files for the DASH multiprocessor.

Please read the `README` file in each of the folders for further directions on how to build the run-time system and application executables.
