Lock Coarsening: Eliminating Lock Overhead in Automatically Parallelized Object-Based Programs

Pedro Diniz[†] and Martin Rinard[‡]

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

Abstract. Atomic operations are a key primitive in parallel computing systems. The standard implementation mechanism for atomic operations uses mutual exclusion locks. In an object-based programming system the natural granularity is to give each object its own lock. Each operation can then make its execution atomic by acquiring and releasing the lock for the object that it accesses. But this fine lock granularity may have high synchronization overhead. To achieve good performance it may be necessary to reduce the overhead by coarsening the granularity at which the computation locks objects.
In this paper we describe a static analysis technique — lock coarsening — designed to automatically increase the lock granularity in object-based programs with atomic operations. We have implemented this technique in the context of a parallelizing compiler for irregular, object-based programs. Experiments show these algorithms to be effective in reducing the lock overhead to negligible levels.

1 Introduction

Atomic operations are an important primitive in the design and implementation of parallel systems. Operations are typically made atomic by associating mutual exclusion locks with the data that they access. An atomic operation first acquires the lock for the data that it manipulates, accesses the data, then releases the lock.

We have implemented a compiler designed to automatically parallelize object-based computations that manipulate irregular, pointer-based data structures. This compiler uses commutativity analysis [9] as its primary analysis paradigm. For the generated program to execute correctly, each operation in the generated parallel code must execute atomically. The automatically generated code therefore contains mutual exclusion locks and constructs that acquire and release these locks.

We have found that the granularity at which the generated parallel computation locks objects can have a significant impact on the overall performance.

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

The natural lock granularity is to give each object its own lock and generate code in which each operation acquires and releases the lock for the object that it accesses. Our experimental results indicate, however, that locking objects at this fine granularity may introduce enough overhead to significantly degrade the overall performance. To achieve good performance we have found it necessary to coarsen the lock granularity to reduce the amount of lock overhead.

This paper presents the analysis algorithms and program transformations that the compiler uses to automatically coarsen the lock granularity. We have implemented these algorithms in the context of a parallelizing compiler for object-based programs [9]. This paper also presents experimental results that characterize the performance impact of using the lock coarsening algorithms in the compiler.

The results show the algorithms to be effective in reducing the lock overhead to negligible levels. They also show that an overly aggressive lock coarsening algorithm can significantly impair the performance by artificially increasing lock contention (lock contention occurs when two processors attempt to acquire the same lock at the same time). A successful compiler must therefore negotiate a trade off between reducing lock overhead and increasing lock contention.

This paper makes the following contributions:

- It introduces two techniques for reducing lock overhead: data lock coarsening and computation lock coarsening.
- It presents novel and practical lock coarsening algorithms that a compiler can use to reduce the lock overhead.
- It presents experimental results that characterize the performance impact of the lock coarsening algorithms on several automatically parallelized applications. These performance results show that, for these applications, the algorithms can effectively reduce the lock overhead to negligible levels.

The rest of this paper is structured as follows. Section 3 presents an example that illustrates how coarsening the lock granularity can reduce the lock overhead. Section 4 describes the kinds of programs that the lock coarsening algorithms are designed to optimize. In Sections 5 and 6 we present the lock coarsening algorithms and transformations. In Section 7 we present experimental results that characterize the impact of the lock coarsening algorithms on the overall performance of two automatically parallelized applications.

2 Basic Issues in Lock Coarsening

The lock coarsening algorithms deal with two basic sources of performance loss: lock overhead and lock contention.

- **Lock Overhead:** Acquiring or releasing a lock generates overhead; the goal of the algorithms is to reduce this overhead by applying transformations that make the computation execute fewer acquire and release constructs.

– **Lock Contention:** Lock contention occurs whenever one processor attempts to acquire a lock held by another processor. In this case the first processor must wait until the second processor releases the lock; the first processor performs no useful computation during time it spends waiting for the lock to be released. Increased lock contention therefore reduces the amount of available parallelism.

All of the transformations that the algorithms apply to reduce the lock overhead have the potential to increase the lock contention. The algorithms must therefore negotiate a tradeoff between the lock overhead and the lock contention.

The algorithms apply two lock coarsening techniques: data lock coarsening and computation lock coarsening.

– **Data Lock Coarsening:** Data lock coarsening is a technique in which the compiler associates one lock with multiple objects that tend to be accessed together. The compiler then transforms computations that manipulate one or more of the objects. Each transformed computation acquires the lock, performs the manipulations, then releases the lock. The original computation, of course, acquired and released a lock every time it manipulated any one of the objects.

  Data lock coarsening may improve the computation in two ways. First, it may reduce the number of executed acquire and release constructs — it enables computations to access multiple objects with the overhead of only a single acquire construct and a single release construct. Second, it may reduce the number of locks that the computation requires to execute successfully — giving multiple objects the same lock may reduce the number of allocated locks.

  An overly aggressive data lock coarsening algorithm may introduce *false contention*. False contention occurs when two operations attempt to acquire the same lock even though they access different objects.
– **Computation Lock Coarsening:** Consider a computation that repeatedly acquires and releases the same lock. This may happen, for example, if a computation performs multiple operations on the same object or on objects that have all been given the same lock by the data lock coarsening algorithm. The computation lock coarsening algorithm analyzes the program to find such computations. It then transforms the computations to acquire the lock, perform the operations with no additional synchronization, then release the lock. This transformation may significantly reduce the number of executed acquire and release constructs.

  An overly aggressive computation lock coarsening algorithm may introduce *false exclusion*. False exclusion may occur when a computation holds a lock for an extended period of time during which it does not access one of the lock's objects. If another computation attempts to acquire the lock (so that it may access one of the lock's objects), it must wait for the first computation to release the lock even though the first computation is not actively accessing any of the lock's objects. False exclusion may therefore reduce the performance by decreasing the amount of available concurrency.

There is a potential interaction between lock coarsening and concurrency generation. To ensure that all of the operations within a given coarsened mutual exclusion region execute atomically with respect to each other, the algorithms require that the entire computation within the coarsened region execute sequentially. There are two options: refusing to apply the lock coarsening transformation if the coarsened region would contain a concurrency generation construct, or removing all of the concurrency generation constructs within the coarsened region. The current transformations apply the first option.

There may be a concern that the transformations will introduce deadlock. As explained in Sections 5 and 6, the lock coarsening transformations never cause a program to deadlock.

3 Example

In this section we provide an example, inspired by the Barnes-Hut application in Section 7, that illustrates both kinds of lock coarsening. The example computation manipulates an array of pointers to nodes; each node has a vector and a count. There is also a set of values stored in a binary search tree. The computation scales every node by all of the values in the tree that fall within a certain range. It finds all of these values by traversing the binary search tree.

Figure 1 contains the parallel C++ code for this example. Each class is augmented with a mutual exclusion lock; the parallel code uses this lock to make operations on objects of that class atomic. If an operation modifies its receiver object[1], it first acquires the receiver's lock, performs the modification, then releases the lock.

The computation starts at the nodeSet::scaleNodeSet method. This method invokes the node::traverse method in parallel for each node in the array; the parallel for loop makes the loop iterations execute concurrently. Note that all of the invocations of the node::traverse method may not be independent — if two array elements point to the same node, the corresponding loop iterations will modify the same node. The operations in the loop iterations must therefore execute atomically for the computation to execute correctly.

The node::traverse method traverses the binary search tree to find all of the values in the range [min,max]. Whenever it finds a value inside the range, it invokes the node::scaleNode method to scale the node by the value. The node::scaleNode method scales a node by incrementing the count of applied scale operations, then invoking the vector::scaleVector method to scale the vector stored in the node.

3.1 Data Lock Coarsening in the Example

An examination of the parallel code in Figure 1 reveals that the computation acquires and releases two locks every time it scales a node: the lock in the node

[1] Programs that use the object-based programming paradigm structure the computation as operations on objects. Each operation has a single receiver object; as described in Section 4 this object is the object that the operation manipulates.

```
const int NDIM 3;
class vector {
  lock mutex;
  double value[NDIM];
public:
  void scaleVector(double s){
    mutex.acquire();
    for(int i=0; i < NDIM; i++)
      value[i] *= s;
    mutex.release();
  }
};
class tree {
public:
  double x;
  tree *left;
  tree *right;
};
class node {
  lock mutex;
public:
  int count;
  vector value;
  void scaleNode(double s);
  void traverse(tree *t,
    double min, double max);
};
class nodeSet {
  int size;
  node **elements;
public:
  void scaleNodeSet(tree *t,
    double min, double max);
};
```

```
node::scaleNode(double s){
  mutex.acquire();
  count++;
  mutex.release();
  value.scaleVector(s);
}
node::traverse(tree *t,
    double min, double max){
  if((min ≤ t->x) && (t->x < max))
    scaleNode(t->x);
  if(min ≤ t->x)
    traverse(t->left,min,max);
  if(t->x ≤ max)
    traverse(t->right,min,max);
}
nodeSet::scaleNodeSet(tree *t,
    double min, double max){
  scaleNodeSet(t, min, max);
  wait();
}
nodeSet::scaleNodeSet(tree *t,
    double min, double max){
  parallel for(int i = 0; i < size; i++){
    elements[i]->traverse(t, min, max);
  }
}
```

Fig. 1. Parallel Node Scaling Example

object (the node::scaleNode method acquires and releases this lock) and the lock in the nested vector object inside the node object (the vector::scaleVector method acquires and releases this lock).

It is possible to eliminate the acquisition and release of the lock in the nested vector object by coarsening the lock granularity as follows. Instead of giving each nested vector object its own lock, the compiler can use the lock in the enclosing node object to make operations on the nested vector object atomic. Figure 2 contains the transformed code that locks the objects at this granularity. The compiler generates a new version of the vector::scaleVector method (this new version is called vector::syncFree_scaleVector) that does not acquire the

lock. It invokes this new version from within the node::scaleNode method and transforms the code so that it holds the node's lock during the execution of the vector::syncFree_scaleVector method.

```
vector::syncFree_scaleVector(double s){
 for(int i=0; i < NDIM; i++)
   value[i] *= s;
}
node::scaleNode(double s){
 mutex.acquire();
 count++;
 vector.syncFree_scaleVector(s);
 mutex.release();
}
```

Fig. 2. Data Lock Coarsening Example

```
node::syncFree_scaleNode(double s){
 count++;
 vector.syncFree_scaleVector(s);
}
node::syncFree_traverse(tree *t,
 double min, double max){
 if((min ≤ t->x) && (t->x < max))
   syncFree_scaleNode(t->x);
 if(min ≤ t->x)
   syncFree_traverse(t->left,min,max);
 if(t->x ≤ max)
   syncFree_traverse(t->right,min,max);
}
node::traverse(tree *t,
   double min, double max){
 mutex.acquire();
 syncFree_traverse(t, min, max);
 mutex.release();
}
```

Fig. 3. Computation Lock Coarsening Example

To legally perform this transformation, the compiler must ensure that every thread that executes a vector operation acquires the corresponding node lock before it executes the operation. An examination of the code shows that it satisfies this constraint.

This transformation illustrates the utility of data lock coarsening. It reduces the number of executed locking constructs by a factor of two because it eliminates the acquire/release pair in the vector::scaleVector method. The compiler can also omit the mutual exclusion lock declaration in the vector class because none of the methods in the parallel computation acquire or release the lock.

3.2 Computation Lock Coarsening in the Example

The example also contains an opportunity for computation lock coarsening. Consider the subcomputation generated as a result of executing a node::traverse method. This subcomputation periodically executes node::scaleNode methods, which acquire and release the node's mutual exclusion lock. All of these executions acquire and release the same mutual exclusion lock. In fact, all of the operations in the entire subcomputation that acquire any lock acquire the same

lock: the lock in the receiver object of the original node::traverse operation. It is therefore possible to coarsen the lock granularity by acquiring the lock once at the beginning of the subcomputation, then holding it until the subcomputation finishes. This transformation eliminates all of the lock constructs except the initial acquire and the final release. Figure 3 shows the transformed code.

This example also illustrates the potential for false exclusion. Consider the original program in Figure 1. This program only holds the node's lock when it is actually updating the node. The transformed code in Figure 3 holds the lock for the entire traversal. If two traversals on the same node seldom update the node, they can execute mostly in parallel in the original version of the code. In the coarsened version they will execute serially. As we will see in Section 7, this kind of serialization may significantly impair the performance of the parallel computation. The compiler must therefore ensure that its lock coarsening policy does not introduce a significant amount of false exclusion.

4 Model of Computation

Before presenting the lock coarsening algorithms, we discuss the kinds of programs that they are designed to optimize. First, the algorithms are designed for pure object-based programs. Such programs structure the computation as operations on objects. Each object implements its state using a set of instance variables. Each instance variable can be either a nested object or a primitive type from the underlying language such as an integer, a double, or a pointer to an object. Each object has a mutual exclusion lock that exports an acquire construct and a release construct. Once a processor has successfully executed an acquire construct on a given lock, all other processors that attempt to acquire that lock block until the first processor executes a release construct. Operations on the object use its lock to ensure that they execute atomically.

Programmers define operations by writing methods. Each operation corresponds to a method invocation: to execute an operation, the machine executes the code in the corresponding method. Each operation has a receiver object and several parameters. When an operation executes it can access the parameters, invoke other operations or access the instance variables of the receiver. There are several restrictions on instance variable access. An operation cannot directly access an instance variable of a nested object — it can only access the variable indirectly by invoking an operation that has the nested object as the receiver. If an instance variable is declared in a parent class from which the receiver's class inherits, the operation can not directly access the instance variable — it can only access the variable indirectly by invoking an operation whose receiver's class is the parent class.

The computation consists of a sequence of alternating serial and parallel phases. Within a parallel phase the computation uses constructs such as parallel loops to create operations that execute concurrently. The only synchronization consists of the mutual exclusion synchronization required to make the operations atomic and the barrier synchronization at the end of a parallel phase.

If an operation accesses an instance variable that may be modified during the parallel phase, it uses the lock in the instance variable's object to make its access atomic. Before the operation executes its first access to the instance variable, it acquires the object's lock. It releases the lock after it completes the last access.

We extend the model for read-only data as follows. If no operation in the parallel phase modifies an instance variable, any operation in the phase (including operations whose receiver is not the object containing the instance variable) can directly access the variable without synchronization.

There may be a concern that the model of computation imposes overhead in the form of excessive method invocations. We have found that inlined methods and the extension described in the previous paragraph, eliminate virtually all of this overhead. The serial C++ versions of our two benchmark applications, for example, perform slightly better than the C versions from the SPLASH benchmark set [9].

5 Data Lock Coarsening

The data lock coarsening algorithm starts with a computation in which each object has its own mutual exclusion lock. The basic idea is to increase the lock granularity by giving multiple objects the same lock. Before the computation accesses any one of these objects, it first acquires the lock. Once it has acquired the lock it can access any other object that has the same lock with no further synchronization.

The algorithm must first choose which objects should have the same lock. The current policy attempts to give nested objects the same lock as their enclosing object. The algorithm must then determine if it can transform the entire parallel computation to lock objects at the new granularity. The key issues are to determine statically the new lock that each operation must hold, to make sure that in the generated code each operation actually holds that lock, and to ensure that no part of the computation locks objects at the old granularity.

5.1 The Data Lock Coarsening Algorithm

The primary responsibility of the data lock coarsening algorithm is to ensure that every time the computation executes an operation on a nested object, it holds the lock in the nested object's enclosing object. The algorithm checks that the computation satisfies this constraint by computing the set of methods that may have a nested object as the receiver. It then verifies that all of these methods are invoked only from within methods that have the nested object's enclosing object as their receiver. In this case, the algorithm can generate code that holds the enclosing object's lock for the duration of all methods that execute on nested objects. Because the lock in the enclosing object ensures the atomic execution of all methods that execute with nested objects as the receiver, the compiler can eliminate the lock constructs in these methods. Figure 4 outlines this algorithm.

The algorithm records the results of the data lock coarsening in the variable `lockClass`. Given a method m, `lockClass`$[m]$ is the class whose lock ensures

Primitive Operations:
`receiverClass`(*method* m) : the class that m is defined in.
`methods`(*class* c) : set of potentially invoked methods in the computation that have class c as receiver.
`closed`(*method* m) : true if the entire computation generated as a result of executing m only accesses the receiver object of m or nested objects of the receiver object of m. The compiler computes this by traversing the call graph.
`invokedMethods`(*method* m) : set of methods that may be directly or indirectly invoked as a result of executing m.
`generatesConcurrency`(*method* m) : true if m contains any constructs that generate parallel execution.
method set m_1 `dominates` *method set* m_2 : true if for every method $m \in ms_2$, every path from the root of the call graph to m contains a method in ms_1.
`CoarsenGranularity`(*method* m) : true if m satisfies the false exclusion policy in Section 6.

global `lockClass`;
void `DataLockCoarseningClass`(*class* c)
  $ms_1 = \{m : m \in$ `methods`(c) and `closed`$(m)\}$;
  $ms_2 = \cup\{$`invokedMethods`$(m) : m \in ms_1\} - ms_1$;
  $ms_3 = \{m : m \in ms_1$ and `invokedMethods`$(m) \subseteq ms_2\}$
  $cs \quad = \{$`receiverClass`$(m) : m \in ms_2\}$;
  $ms_4 = \cup\{$`methods`$(c') : c' \in cs\}$;
  if(ms_3 `dominates` ms_4) then
    for all methods $m \in ms_3 \cup ms_4$ do
      if(`generatesConcurrency`(m) or (not `CoarsenGranularity`(m))) return;
    for all methods $m \in ms_3$ do
      Make the generated parallel version of m invoke the synchronization-free version of each method that it invokes. This synchronization-free version contains no lock constructs and invokes the synchronization-free version of all methods that it invokes. Also make the first statement of m acquire its receiver's lock and the last statement of m release the lock.
    for all methods $m \in ms_4$ do
      `lockClass`$[m] = c$;

Fig. 4. Data Lock Coarsening Algorithm

the atomic execution of m. The computation lock coarsening algorithm presented below in Section 6 uses `lockClass` to determine if can legally apply the computation lock coarsening transformation.

To generate code for the transformed computation, the compiler generates a new *synchronization-free* version of all methods whose receivers are nested objects — in other words, all of the methods in the set ms_4. The synchronization-free version is the same as the original version except that it omits any synchronization constructs present in the original version and invokes the synchronization-

free version of all of the methods that it executes. The compiler also modifes all of the call sites of the methods in the set ms_3 to ensure that they invoke the synchronization-free version of each invoked method.

Finally, we briefly note that the data lock coarsening algorithm can never introduce deadlock. The model of computation in Section 4 ensures that the processor holds no locks when it enters the transformed version of one of the methods in ms_3. Because the entire computation of the transformed method only acquires and releases the lock in its receiver object, there is no possibility of deadlock.

6 Computation Lock Coarsening

Th computation lock coarsening algorithm traverses the call graph, attempting to identify methods whose computation repeatedly acquires and releases the same lock. At each node of the program call graph the computation lock coarsening algorithm uses the `ComputationLockCoarseningMethod`(m) algorithm to determine if it should coarsen the granularity at the execution of the corresponding method m of the call graph node.

Figure 5 presents `ComputationLockCoarseningMethod`(m) algorithm. It first checks that m is closed. It then checks that none of the methods that m's computation may execute acquire and release different locks. It also checks to make sure that none of these methods contain any concurrency generation constructs. If m passes all of these tests, it is legal for the compiler to apply the computation lock coarsening transformation.

```
global lockClass;
void ComputationLockCoarseningMethod(method m)
  if(generatesConcurrency(m)) return;
  if(m is closed)
    c = receiverClass(m);
    for all m' ∈ invokedMethods(m) do
      if(lockClass[receiverClass(m)] ≠ c) or (generatesConcurrency(m)) return;
    if(CoarsenGranularity(m))
      Make the generated parallel version of m invoke the synchronization-free
      version of each method that it invokes. This synchronization-free version
      contains no lock constructs and invokes the synchronization-free version
      of all methods that it invokes. Also make the first statement of m
      acquire its receiver's lock and the last statement of m release the lock.
```

Fig. 5. Computation Lock Coarsening Algorithm

The remaining question is whether coarsening the granularity will generate an excessive amount of false exclusion. The compiler currently uses one of three policies to determine if it should apply the transformation:

- **Original:** Never apply the transformation — use the original granularity.
- **Bounded:** Increase the granularity only if the transformation will not cause the computation to hold a lock for a statically unbounded number of method executions. The compiler implements this policy by testing for cycles in the call graph of the set of methods that may execute while the computation holds the lock. It also checks to make sure that none of these methods contain loops that invoke methods. The idea is to limit the potential severity of any false exclusion by limiting the amount of time the computation holds any given lock.
- **Aggressive:** Always increase the granularity if it is legal to do so.

This policy choice is encapsulated inside the `CoarsenGranularity`(m) algorithm. If the algorithm determines that it should apply the transformation, the compiler generates code for m that acquires the lock, invokes the synchronization-free versions of all of the methods that it invokes, then releases the lock.

The computation lock coarsening algorithm can never introduce deadlock. It simply replaces computations that acquire and release the same lock with computations that acquire and release the lock only once. If the original version does not deadlock, the transformed version can not deadlock.

7 Experimental Results

We have implemented the lock coarsening algorithms described in Sections 5 and 6 and integrated them into a prototype compiler that uses commutativity analysis [9] as its primary analysis paradigm. In this section we present experimental results that characterize the performance impact of using the lock coarsening algorithms in a parallelizing compiler. We report performance results for two automatically parallelized scientific applications: the Barnes-Hut hierarchical N-body solver and the Water code [11].

7.1 Methodology

To evaluate the impact of the lock coarsening policy on the overall performance, we implemented the three lock coarsening policies described in Section 6. We then built three versions of the prototype compiler. The versions are identical except that each uses a different lock coarsening policy. We then used the three versions of the compiler to automatically parallelize the benchmark applications. We obtained three automatically parallelized versions of each application — one from each version of the compiler. The generated code for each application differs only in the lock coarsening policy used to reduce the lock overhead. [2] We evaluated the performance of each version by running it on a 32-processor Stanford DASH machine [6]. Because the prototype compiler is a source-to-source translator, we use a standard C++ compiler to generate object code for the automatically generated parallel programs.

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

7.2 Barnes-Hut

The Barnes-Hut application simulates the trajectories of a set of interacting bodies under Newtonian forces [1]. It uses a sophisticated pointer-based data structure: a space subdivision tree that dramatically improves the efficiency of a key phase in the algorithm. The application consists of approximately 1500 lines of serial C++ code. The compiler is able to automatically parallelize phases of the application that together account for over 95% of the execution time.

Figure 6 presents the speedup curves for this application. The speedups are calculated relative to the serial version of the code, which executes with no lock or parallelization overhead. All versions scale well, which indicates that the compiler was able to effectively parallelize the application. Although the absolute performance varies with the lock coarsening policy, the performance of the different parallel versions scales at approximately the same rate. This indicates that the lock coarsening algorithms introduced no significant contention.

7.3 Water

Water uses a $O(n^2)$ algorithm to simulate a set of n water molecules in the liquid state. The application consists of approximately 1850 lines of serial C++ code. The compiler is able to automatically parallelize phases of the application that together account for over 98% of the execution time.

Figure 7 presents the corresponding speedup curves. [3] The Original and Bounded versions initially perform well (the speedup over the sequential C++ version at sixteen processors is approximately 5.2). But both versions fail to scale beyond eight processors. The Aggressive version fails to scale well at all — the maximum speedup for this version is only 2.0.

A further investigation into the source of the lack of scalability reveals that the application suffers from high lock contention. For this application, the Bounded policy yields the best results and the corresponding parallel code attains respectable speedups. Although the Aggressive policy dramatically reduces the number of executed locking operations, the introduced lock contention almost completely serializes the execution.

8 Related Work

8.1 Automatically Parallelized Scientific Computations

Previous parallelizing compiler research in the area of synchronization optimization has focused almost exclusively on synchronization optimizations for parallel loops in scientific computations [8]. The natural implementation of a parallel

[3] The speedup curves are given relative to the sequential C++ version. We have also obtained sequential C and sequential Fortran versions of this application. The running times for these versions on an input size of 343 molecules are: 65 seconds (Fortran), 68 seconds (C) and 73 seconds (C++).

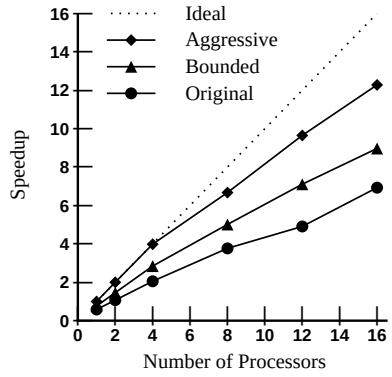

Fig. 6. Speedup for Barnes-Hut (16384 bodies)

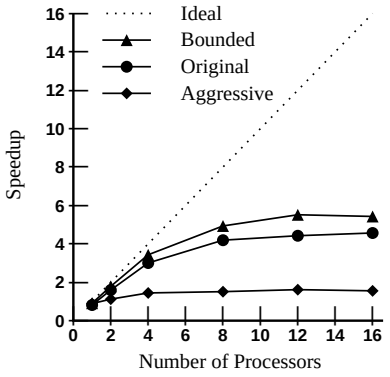

Fig. 7. Speedup for Water (512 molecules)

loop requires two synchronization constructs: an initiation construct to start all processors executing loop iterations, and a barrier construct at the end of the loop. The majority of synchronization optimization research has concentrated on removing barriers or converting barrier synchronization constructs to more efficient synchronization constructs such as counters [10]. Several researchers have also explored optimizations geared towards exploiting more fine grained concurrency available within loops [3]. These optimizations automatically insert one-way synchronization constructs such as post and wait to implement loop-carried data dependences.

The research presented in this paper investigates synchronization optimizations for a compiler designed to parallelize object-based programs, not loop nests that manipulate dense arrays using affine access functions. The problem that our compiler faces is the efficient implementation of atomic operations, not the efficient implementation of data dependence constraints.

8.2 Database Concurrency Control

A goal of research in the area of database concurrency control is to develop efficient locking algorithms for atomic transactions. This goal is similar to our goal of efficiently implementing atomic operations in parallel programs. In fact, database researchers have identified lock granularity as a key issue in the implementation of atomic transactions, and found that excessive lock overhead can be a significant problem if the lock granularity is too fine [2, 4].

The proposed solution to the problem of excessive lock overhead in the context of database concurrency control is to dynamically coarsen the lock granularity using a technique called lock escalation. The idea is that the lock manager (which is responsible for granting locks to transactions) may coarsen the lock granularity by dynamically locking a large section of the database on behalf of a given transaction. If the transaction requests a lock on any object in that

section, the lock manager simply checks that the transaction holds the coarser granularity lock.

There are several key differences between the lock manager algorithm and the lock coarsening algorithms presented in this paper. The lock manager algorithm only attempts to increase the data lock granularity — there is no attempt to increase the computation lock granularity. This paper presents algorithms that coarsen both the data and the computation lock granularities.

Several other differences stem from the fact that the lock manager algorithm takes place dynamically, which means that it can not change the program generating the lock requests. The programs therefore continue to execute lock acquire and release constructs at the fine granularity of individual items in the database. The goal of the lock manager algorithm is to make it possible to implement the fine grain lock requests more efficiently in cases when the lock manager has granted the transaction a coarse grain lock on a section of the database, not to change the granularity of the lock requests themselves.

Because the transactions always generate lock requests at the granularity of items in the database, the lock manager must deal with the possibility that a transaction may attempt to lock an individual item even though it does not hold a lock on the section of the database that includes the item. The locking algorithm must therefore keep track of correspondence between different locks that control access to the same objects. Tracking this correspondence complicates the locking algorithm, which makes each individual lock acquire and release less efficient in cases when the lock manager has not already granted the transaction a coarse grain lock.

The lock coarsening algorithms presented in this paper, on the other hand, transform the program so that it always generates lock requests at the coarser granularity. The fine grain lock acquire and release operations are completely eliminated and generate no overhead whatsoever. Furthermore, computations that access the same object always execute acquire and release constructs on the same lock. This property makes it possible for the implementation to use an extremely efficient lock implementation. Modern processors have synchronization instructions that make it possible to implement the required lock acquire and release constructs efficiently [5].

8.3 Efficient Synchronization Algorithms

Other researchers have addressed the issue of synchronization overhead reduction. This work has concentrated on the development of more efficient implementations of synchronization primitives using various protocols and waiting mechanisms [7]. The research presented in this paper is orthogonal to and synergistic with this work. Lock coarsening reduces the lock overhead by reducing the frequency with which the generated parallel code acquires and releases locks, and not by providing a more efficient implementation of the locking constructs.

9 Conclusions

This paper addresses a fundamental issue in the implementation of atomic operations: the granularity at which the computation locks the data that atomic operations access. We have found that using the natural lock granularity for object-based programs (giving each object its own lock and having each operation lock the object that it accesses) may significantly degrade the performance. We have presented algorithms that can effectively reduce the lock overhead by automatically increasingly the granularity at which the computation locks data. We have implemented these algorithms and integrated them into a parallelizing compiler for object-based languages. We present experimental results that characterize the performance impact of using the lock coarsening algorithms in this context. These results show that the algorithms can effectively reduce the lock overhead to negligible levels.

References

1. J. Barnes and P. Hut. A hierarchical O(NlogN) force-calculation algorithm. *Nature*, pages 446–449, December 1976.
2. P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
3. R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, St. Charles, IL, August 1986.
4. U. Herrmann, P. Dadam, K. Kuspert, E. Roman, and G Schlageter. A lock technique for disjoint and non-disjoint complex objects. In *Proceedings of the International Conference on Extending Database Technology (EDBT'90)*, pages 219–235, Venice, Italy, March 1990.
5. G. Kane and J. Heinrich. *MIPS Risc Architecture*. Prentice-Hall, 1992.
6. D. Lenoski. *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor*. PhD thesis, Stanford, CA, February 1992.
7. B-H. Lim and A. Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1994.
8. S. Midkiff and D. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, 36(12):1485–1495, December 1987.
9. M. Rinard and P. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Proceedings of the SIGPLAN '96 Conference on Program Language Design and Implementation*, Philadelphia, PA, May 1996. (http://www.cs.ucsb.edu/~martin/pldi96.ps).
10. C. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 144–155, Santa Barbara, CA, July 1995.
11. S. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995.

This article was processed using the LaTeX macro package with LLNCS style