EXPLOITING COMMUTING OPERATIONS IN PARALLELIZING SERIAL PROGRAMS

PEDRO DINIZ AND MARTIN RINARD

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF CALIFORNIA, SANTA BARBARA
SANTA BARBARA, CA 93106

Abstract. Two operations commute if the result of their execution is independent of the order in which they execute. Commuting operations can be executed concurrently provided they execute atomically on the objects they access. Statically recognizing commuting operations is of great interest because they increase the amount of concurrency a compiler can exploit. In this document we introduce commutativity analysis - a new technique for automatically parallelizing serial programs. We then conduct a feasibility study of existing scientific applications as to the existence and exploitability of commuting operations. We study the commuting operations present in one such application - the Barnes-Hut hierarchical N-body algorithm. We then parallelize this application using knowledge of commuting operations and present performance results of the parallel code for a shared-memory multiprocessor.

1. Introduction. Parallel machines offer the promise of a significant performance increase by enabling multiple processors to concurrently execute different parts of the computation. Programmers have traditionally developed applications for parallel machines using explicitly parallel languages. Unlike serial programming languages, explicitly parallel languages present a complex programming model characterized by such phenomena as deadlock, nondeterministic execution and, on message-passing machines, the need to directly manage the movement of data through the computation. The obvious programming advantages of the sequential imperative programming paradigm have therefore inspired the development of analysis techniques and compilers designed to automatically parallelize serial programs.

Current parallelizing compilers preserve the semantics of the original serial program by preserving the data dependencies [1, 10]. These compilers attempt to identify independent pieces of computation (two pieces of computation are independent if neither writes a piece of data that the other accesses), then generate code that executes independent pieces concurrently. A significant limitation of this approach is the difficulty of performing dependence analysis that is precise enough to expose substantial amounts of concurrency. While researchers have been able to develop reasonably effective algorithms for loop nests that manipulate dense matrices using affine access functions [13], there has been little progress towards the successful automatic analysis of programs that manipulate pointer-based data structures. Researchers have attempted to build totally automatic systems [6], but the most promising approaches require the programmer to provide annotations that specify information about the global topology of the manipulated data structures [7]. The goal of these annotations is to help the compiler to discover pieces of code that access disjoint pieces of data. A second, more fundamental limitation of data-dependence based approaches is an inability to parallelize computations that manipulate graph-like data structures. The aliases inherently present in these data structures preclude the static discovery of independent pieces of code, forcing the compiler to generate serial code. Finally, preserving the data dependences for programs that periodically update shared data structures can artificially limit the amount of exposed concurrency, since tasks must delay updates until they are sure that each update will not change the relative order of reads and writes to the shared data structure [3, 16].

A new technique, commutativity analysis [17], has been developed that eliminates many of the limitations of existing data-dependence based approaches. Instead of preserving the relative order of individual reads and writes to single words of memory, commutativity analysis aggregates both data and computation into larger grain units. It then analyzes the computation at this granularity to discover when pieces of the computation commute (i.e. generate the same result regardless of the order in which they execute). If all of the operations required to perform a given computation commute, the compiler can automatically generate parallel code. While the resulting parallel program may violate the data dependences of the original serial program, it is still guaranteed to generate the same result. We expect this approach to be most useful for computations whose parallel tasks periodically generate commuting updates to shared data structures and/or traverse recursive data structures such as lists, trees and graphs.

In this work we study issues associated with exploiting commuting operations in parallelizing sequential applications. The goals of this study are twofold. First, observe which commuting operations have been used in effectively parallelizing sequential applications. Second, assess the difficulty in automatically detecting and exploiting such commuting operations. For the purpose of this study we analyze applications included in the SPLASH benchmark [18] as they represent complete applications engineers and scientists use in practice. We then take a particular application, the Barnes-Hut hierarchical N-body algorithm, and parallelize it using the knowledge of commuting operations. We evaluate the performance of the parallel code on the Stanford DASH [11], a cache-coherent, shared-memory multiprocessor.

This document is organized as follows. The next section introduces commutativity analysis by means of a graph traversal example. Section 3 describes the analysis of SPLASH applications in light of commuting operations used. We present the commuting operations observed and discuss its fully automatic detection difficulties. In section 4 we describe the Barnes-Hut hierarchical N-body algorithm case study in greater detail. In section 5 we describe practical experiments with this application and report performance results. Finally, we present the conclusions in section 6.

2. Commutativity Analysis. We briefly describe the basic commutativity analysis approach. A full description of this technique can be found in [17].

Commutativity analysis is a static analysis framework for detecting commuting operations. Two operations commute when they generate the same result regardless of the order in which they execute. Commutativity analysis is designed for programs written using a pure object-based paradigm. Such programs structure the computation as a sequence of operations on objects. Each operation consists of a receiver object, an operation name and several parameters. Each operation name identifies a method that defines the behavior of the operation; when the operation executes it invokes that method. Each object implements its state using a set of instance variables. When a method executes it can recursively invoke other operations and/or use primitive operators (such as addition and multiplication) to perform computations involving the instance variables of the receiver. We require that each method be *separable* - i.e., that the execution of the method can be decomposed into an object section followed by an invocation section. The object section performs all accesses to the receiver. The invocation section invokes other operations and does not access the receiver. It is of course possible for local variables to carry values computed in the object section into the invocation section. Separability is important for analysis purpose because it clearly separates the effects on the receiver object from the method invocations. Separability does not reduce the expressive power of the language. A compiler can conduct a simple separability test and decompose any non-separable method into separable ones by the introducing auxiliary methods.

The foundation of commutativity analysis is a set of conditions that a compiler can use to test if two operations A and B commute. The commutativity testing conditions must consider two execution orders: the execution order in which A executes first then B executes, and the execution order in which B executes first then A executes. The first commutativity testing condition examines the new values of the instance variables of the two receivers after the two operations execute. For each instance variable the condition checks that the new value of the instance variable after first A then B executes is the same as the new value after first B then A executes. The second commutativity testing condition has to do with the combined set of operations that the two operations invoke. The condition requires that when first A then B executes the two operations together invoke the same set of operations as the set of operations invoked when first B then A executes. Both commutativity testing conditions are trivially satisfied if the two operations have different receivers. We therefore focus on the case when the two operations have the same receiver.

It is possible to determine if each of the receiver's instance variables has the same new value in both execution orders symbolically executing the invoked. methods to extract two symbolic expressions. One of the symbolic expressions denotes the new value of the instance variable when first A then B executes. The other symbolic expression denotes the new value of the instance variable when first B then A executes. Given these two expressions, a compiler may be able to use algebraic reasoning to discover if they denote the same value.

It is possible to determine if A and B together invoke the same set of operations in both execution orders by analyzing the methods to extract two sets of symbolic method invocation expressions. One of the sets denotes the set of operations invoked when first A then B executes. The other set denotes the set of operations invoked when first B then A executes. Given these two sets, a compiler may again be able to use algebraic reasoning to discover that the two sets of symbolic method invocation expressions denote the same set of invoked expressions.

We now show how commutativity analysis concepts can be used to parallelize complex traversals by discussing the automatic parallelization of the marked graph traversal algorithm in Figure 1. The algorithm is written in C++.

The traversal may visit each node of the graph several times. Depending on the value of the `mark` instance the node visit updates the receiver's `val` instance variable and recursively invokes the `traverse` operation on the nodes that the `left` and `right` instance variables point to.

The way to parallelize this program is to execute the two recursive `traverse` operations concurrently. But this parallelization may violate the data dependences. The serial program executes all accesses generated by the left traversal before all accesses generated by the right traversal. If the two traversals visit the same node, in the par-

```
class graph { int mark; int val; graph *left; graph *right; }

graph::traverse(int v) {
    if (mark != m) {
        mark = true;
        val = v;
    if (left != NULL)
        left->traverse(v);
    if (right != NULL)
        right->traverse(v);
    }
}
```

Fig. 1. Serial Graph Traversal.

allel execution the right traversal may visit the node before the left traversal, changing the order of reads and writes to that node. This violation of the data dependences may generate cascading changes in the overall execution of the computation. Because of the marking algorithm, a node only generates recursive calls the first time it is visited. So, 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 the overall result of the computation. It is possible to automatically parallelize the computation even though the resulting parallel program may generate computations that differ substantially from the original serial computation. The key property that enables the parallelization is that the parallel computation generates the same set of `traverse` operations as the serial computation and the generated `traverse` operations can execute in any order without affecting the aggregate behavior of the traversal. Given this commutativity information, the compiler can automatically generate the parallel `traverse` operation. Figure 2 shows a high level pseudo-code the concurrent traversal code the compiler is able to generate. The top level `traverse` operation calls the `concurrent_traverse` operation, then invokes the `wait` routine, which blocks until all parallel tasks created by the current task or its descendant tasks finishes. The `concurrent_traverse` operation executes the recursive calls concurrently using the `spawn` construct, which creates a new task for the execution of the new traversal. The compiler also augments the `graph` data structure with a mutual exclusion lock. The `concurrent_traverse` uses this lock to ensure the atomic execution of its accesses to the receiver.

This graph traversal algorithm illustrates several aspects of the computations that commutativity analysis is designed to handle. First, the traversal manipulates a recursive, pointer-based data structure. The global topology of the data structure is irrelevant - it may be a list, a tree or a DAG, contain cycles or even change during the course of the computation. Second, it is possible to statically analyze the program and discover that all of the generated operations commute. This commutativity information enables the automatic generation of parallel code. Third, both the parallel and serial executions generate the same set of operations. Commutativity analysis is not designed for computations (such as branch and bound searches) that may generate different sets of operations in different parallel executions. Fourth, both the parallel and serial computations generate identical data structures at the end of the traversal.

```
class graph {lock key; int mark; int val; graph *left; graph *right; }

graph::traverse(int v) {
      this->concurrent_traverse(v);
      wait();
}

graph::concurrent_traverse(int v) {
      graph *l; graph *r;
      acquire(key);
      if (mark != m) {
            mark = true;
            val = v;
            l = left; r = right;
            release(key);
      if (left != NULL)
            spawn(l->concurrent_traverse(v));
      if (right != NULL)
            spawn(r->concurrent_traverse(v));
      } else {
      release(key);
      }
}
```

Fig. 2. Concurrent Graph Traversal.

Computations such as fully parallel tree insertions that may generate different data structures in different parallel executions are beyond the reach of this and all other published automatic techniques.

3. Commuting operations in application codes. The applications we have analyzed are included in the Stanford SPLASH benchmark set of applications for shared-memory multiprocessors. Details of both sequential codes and the technical issues pertaining the parallelization process for the DASH machine can be found in the SPLASH report [18] and references within.

In this study we have observed the parallelized applications' source codes. No sequential codes were available. With the available description of the applications we were able to reason in the cases described in this section about the sequential program's semantics. For each of the applications we briefly describe both the parallelization opportunities that were exploited and point out which ones took advantage of commuting operations. We conclude with a discussion of the opportunities exploited and assess the difficulty in detecting these opportunities by statically analyzing the source sequential code for each application.

3.1. Ocean. The Ocean application consists of a spatial partial differential equation solver using second order finite-differencing over a fixed size grid. It exploits mostly data-parallelism opportunities as a static distribution of the grid work-load is performed. Commuting operations are explored, however, to achieved better locality by using blocking strategies in the computation of the boundary cells accumulation steps as well as global convergence tests. These approaches clearly violate the original sequential program semantics and are based on the numeric algebraic properties.

3.2. Water. The Water application simulates the trajectories of a set of liquid particles in a cuboidal box under Newtonian forces. It computes all-pairs interaction forces using a predictor-corrector scheme. Work-load is statically distributed in the data-parallelism style. Interprocessor communication and processor data locality are optimized by reordering the particle interactions' computations. Instead of computing for each local particle what its influence on all the remaining particles is, effects of non-local particle are "blocked" and computed for all the local particles. This scheme is possible by the exploitation of numeric algebraic properties.

3.3. LocusRoute. LocusRoute is a VLSI standard-cell routing application code. It attempts to find a good routing layout for the VLSI cells pins by evaluating and choosing the best path for each pair of connections among a couple of possible path permutations. The parallel version implemented clearly violates the sequential program semantics as the order of pairs selected is relevant for the cost of subsequent paths evaluation and hence for the definition of the final configuration.

3.4. Raytracer. This application code renders a scene with geometric shaped objects by tracing-out from each frame pixel position in the observator location-point the reflections and refraction of a given ray. Multiple rays with distinct directions are traced being their contribution, in terms of the three fundamental colors (RGB), added up in the final image. These contributions are evaluated by following the rays trajectories to the light sources. Due to its high irregularity and input dependent work-load distribution per frame pixel element, the parallel application uses a queue of rays to be processed. Rays either resulting from refractions or reflections are stored in this global queue for further processing. Parallelization takes advantage of the rays evaluation independence and the algebraic commuting operations on the frame pixels.

3.5. Discussion. Parallelization of the numeric applications studied is performed using a data-parallelism paradigm defined by a static domain decomposition. Computations are interleaved with global synchronization steps. Commuting operations are often exploited in global reduction steps as is the case of convergence testing. The Ocean case also takes advantage of a specific algorithmic properties: independent relaxation of the subdomains is guaranteed to convergence.

Optimization and non-numeric intensive applications as the LocusRoute or the Raytracer reveal two interesting aspects. In the case of the LocusRoute and due to its inherently sequential semantics the only effective way to parallelize it is to change the semantics of the program. For some given inputs parallel and sequential program will yields distinct answers. This is a particular case in which relaxing the sequential program semantics yields acceptable answers. The Raytracer, on the other hand, exhibits another aspect. While it possesses vast amounts of parallelism as rays may be independently evaluated, its manipulation using a simple link-based data structures precludes the exposure of additional concurrency.

In conclusion and informally we observe the existence of "classes"[1] of applications exhibiting particular kinds of properties that can be exploited in their parallelization process. As an example we have applications, e.g. the LocusRoute, that tolerate different results for the same input, i.e. allow the relaxation of the deterministic sequential program semantics. Other applications, as for example the RayTracer, perform collection-based computations. While it is clear that none of the existing

[1] No formal definition is attempted here as this is a mere informal exposition.

techniques are able to cope with the kinds of application properties, some might still benefit from commutativity analysis techniques, as is the case of the Barnes-Hut N-body algorithm described in the next section.

4. The Barnes-Hut Hierarchical N-body algorithm. The Barnes-Hut Hierarchical algorithm [2] is an important application code for several areas of science and engineering such as astrophysics and chemistry. The Barnes-Hut algorithm simulates the trajectories of a set of interacting bodies, for example galaxies or particles, under Newtonian forces from which system macro level properties can be computed. The Barnes-Hut approach builds on the fact that a body interaction with a cluster of distant bodies can be approximated by a single interaction between the body and the combined center of mass of the cluster. This interaction between bodies and cluster are named body-cell interaction opposed to the direct body-body interactions. A user defined parameter quantifies the threshold distance beyond which a cluster is said to be distant. This strategy allows the complexity of the algorithm to be reduced from the naive $O(n^2)$ to $O(nlogn)$, a significant improvement for large systems. The Barnes-Hut algorithm uses a space subdivision tree to record the relative distances between bodies and cells. This tree is constructed from scratch at each system iteration. The internal data structure nodes are either cells or leaves, each of which represents a square spatial domain where bodies are located. Each leaf holds a maximum constant capacity of bodies. Under suitable cut-off distance selection the error incurred in the algorithm approximation can be made negligible and consistent with the finite-arithmetic round-off error.

Figure 3 below outline the sequential algorithm. Detailed description of either the sequential algorithm or its parallelization for shared-memory or distributed-memory machines can be found in [24, 20].

```
read simulation parameters
while (simulated time < final time) do {
    /* construct BH tree */
    for all bodies do
        insert body b_i in the BH tree

    /* compute center of mass */
    recursively traverse tree and compute center-of-mass

    /* compute gravitational forces */
    for all bodies do
        compute gravitational force on body b_i

    /* update bodies velocities */
    for all bodies do
        update position and velocity of body b_i
}
```

Fig. 3. Barnes-Hut Hierarchical N-body algorithm.

4.1. Commuting Operations. We now describe the commuting operations that can be exploited in in three of the four main steps of this algorithm.

4.1.1. Tree construction. Approximating the interaction between a body and a cluster of bodies by a single body-cell interactions is the basis of the whole design of the Barnes-Hut algorithm. A space subdivision tree (a Quad-tree in a 2D problem, and an Oct-tree for the 3D case) records the relative distances between bodies and hence between cells and bodies and is constructed from scratch at every system step. [2] In this tree data structure each internal node represents a spatial domain hierarchical and spatial related. Terminal nodes – the leaves – hold a maximum constant capacity of bodies. Once this capacity is exceeded the leaf is promoted to a cell, the corresponding spatial domain subdivided, by halving each dimension, and bodies are re-distributed according to their spatial location. Figure 4 illustrates a Quad-tree and the spatial domain it models.

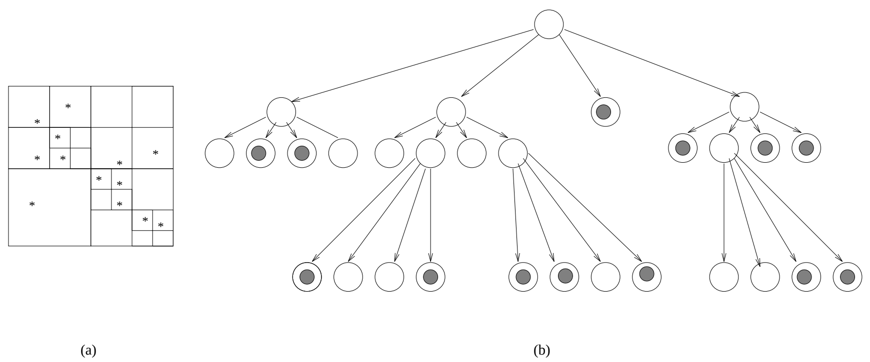

Fig. 4. Barnes-Hut Algorithm Quad-tree Data Structure. Spatial distribution (a) and the internal representation (b) for a unary leaf capacity.

The key property of the tree insertion process is that the final tree, as far as the overall computation is concerned, is oblivious of the bodies insertion order. Two key properties contribute to this fact. First, the leaf, a given body will end up in, depends only on its spatial coordinates and a global function of all bodies positions that define the bounding spatial box, and neither on the sequence of insertions (as in a pure tree) nor on the relative position each body. Second, and inside each leaf the relative insertion order is irrelevant. Thus, and although distinct insertion order yield distinct intermediate and final trees all final possible tree have the same set of bodies on the same leaves. Implementations based on arrays for the set of bodies in each leaf make possible to obtain distinct bodies ordering for each leaf under distinct insertion sequences. The algorithm however is insensitive to the fact the algorithm uses all bodies in every leaf to perform commuting updates of the bodies accelerations and potentials.

4.1.2. Center of Mass computation. During this step the tree is recursively traversed being the center of mass of both leaves and cells computed during the bottom-up phase as the weighted-mean of the contribution of the cell/leaves below as shown in the Figure 5.

At first glance it seems that recursive function invocations commute as their effects on the receive all use numeric commuting operations. However, the last assignment instruction on the `center_of_mass` make invocations non-commutative.

[2] In a sequential implementation this step consumes no more than 2% of the total running time.

```
class treenode {
    real Mass; // Aggregate cell Mass
    vector CofM; // Aggregate cell Center of Mass
}

treenode::center_of_mass(){
    for all subnodes subptr do
        subptr->center_of_mass();
        CofM += subptr->CofM * subptr->Mass;
        Mass += subptr->Mass;
    CofM = CofM/Mass;
}
```

Fig. 5. Center of Mass computation traversal code.

The computations generated by this method cannot be therefore parallelized by the application of commutativity analysis. In practice, however, and using the knowledge that the underlying data structure is a tree, hence all tree nodes constitute disjoint pieces of data, parallelization is possible. Each method invocation locks the tree node object of the computation, concurrently spawns the recursive invocations and waits for them to terminate to compute the node's `CofM` new value.

The problem of statically recognizing the shape of a recursively defined pointer-based data structure is a difficult one. General techniques [6] have been proposed but their effectiveness in dealing with existing applications not yet demonstrated. Related techniques [8] exist that can be used to verify properties of data structure, such as trees or acyclic graphs. It would require a significant engineering effort to combine this technique with data-flow analysis to be able to build a compiler capable of automatically detecting the shape of a given data structure and verifying it through a whole application.

4.1.3. Gravity Forces computation. The force computation step consists of a simple loop that computes, for every body, the experienced combined gravitational force due to interactions with all remaining bodies. These interactions can be of two types. For "close" bodies a *body-body* interaction is computed. For clusters of of bodies sufficiently "distant" the interaction is computed using an synthesized body located at the center-of-mass of the cluster. A user defined parameter defines the threshold distance beyond which a body and a cluster of bodies are considered sufficiently apart. The algorithm performs a recursive traversal of the tree, checking at each spatial cell (holding a cluster of bodies) the validity of the center-of-mass interaction approximation (body-cell interaction). Each interaction performs an update of both the acceleration (`Acc`) and the potential (`Phi`) vectors, using commutative operatos, of the body that is object of the computation. Figure 6 presents a high level description of the code and includes the relevant class declarations. A high level description of the code is shown in Figure 6.

Figure 6 shows the code for the `body_interaction` method that computes the *body-body* interaction. The code for the method `cell_interaction` is similar. The `body_interaction` method performs an update operation on both the body's acceleration (`Acc`) and potential (`Phi`) fields. It is clear that any two invocations of this method on a given object do commute provided the invocations execute atomically.

```
class vector { double val[NDIM]; };

class body {
    real Mass; // mass
    vector Pos; // position
    vector Vel; // velocity
    vector Acc; // acceleration
    real Phi; // potential
}

void compute_forces(){
    for all bodies do
    body->compute_grav(Quad_tree,bounding_box_size);
}

body::compute_grav(treenode,size){
    if need to open cell(treenode,size)
        if treenode is a cell
            for all subcells do
                body->compute_grav(subcell,size/4.0)
        else
            for all bodies p in leaf do
                body->body_interaction(p);
    else
        body->cell_interaction(treenode);
}
body::body_interaction(p){
    real drabs, phii, mor3, drsq; vector ai, dr;
    dr = this->Pos - p->Pos;
    drsq = dr * dr;
    drsq += epssq;
    drabs = sqrt((double) drsq);
    phii = p->Mass / drabs;
    Phi = Phi - phii;
    mor3 = phii / drsq;
    ai = dr * mor3;
    Acc = Acc + ai;
}
```

Fig. 6. Barnes-Hut sample code.

The final result on the body's updated fields is independent of the order in which the two updates are performed. The fact that all invocations of `cell_interaction` and `body_interaction` commute follows by a similar reasoning.

Given the knowledge that both interaction methods commute it is possible to reason about whether or not any two invocations of the `compute_grav` method commute. The `compute_grav` method recursively traverses the data structure along multiple paths invoking the interaction methods on body objects. Because all such interaction methods commute, we are guaranteed that any two invocations of `compute_grav` commute, making it possible to parallelize all invocations of `compute_grav` at the loop level. The ability to detect this loop-level parallelism is particularly attractive because each `compute_grav` invocation generates a significant amount of computational work by operating on double precision vectors at the leaf nodes of the computation.

4.2. Discussion. This Barnes-Hut hierarchical N-body algorithm exhibits commuting operations at different level of difficulty for automatic detection. On one end we have the almost trivial detecting commuting operations (`compute_forces`) where all receiver update are commutative and data structure operations consist only of reads. On the other side we have the most complex and problem specific commuting operations - the Quad-tree insertions. Here there are two challenges. First, to be able to recognize set operations as a first step to make leafs insertions independent. Second to realise global inter-body tree insertion independence.
Automatic recognition of set operations is possible by the combined usage usage of techniques for data structure topological extraction properties and data flow analysis "def-use" chains. No system to date detects such high-level operations. However, they are of crucial importance in the effective parallelization of symbolic and numeric sparse codes as they heavily rely on linked-list data structures either with pointer-based or array-based implementation.

5. Experiments and Results. To obtain an initial measure of the feasibility of exploiting commuting operations, we parallelized and evaluated the performance of the Barnes-Hut algorithm for a variety of number of bodies and number of processors involved. We have derived the code from the `barnes` application code included in the SPLASH2 benchmark set [25] and modified it to an object-based style suitable for analysis by a compiler using commutativity analysis techniques. We then parallelized by hand this serial program taking advantage of the fact that the methods `comp_grav`, `body_interaction` and `cell_interaction` commute. We expect a compiler to automatically extract the commutativity information for these methods and to generate the same parallel code. The code generation follows several code generation *templates* as shown in the Appendix A. We evaluate the performance of this parallel code and also compare the results against results from the highly tuned parallel `barnes` code included in the SPLASH2 benchmark.

To support these experiments we developed a run-time support layer, with lightweight dynamic task spawning and synchronization capabilities. Additional functionality includes features to allow dynamic load balancing and computation coarsening in order to match the application-level concurrency to the machine-level available parallelism. In the next subsections we describe the multiprocessor characterization, run-time environment used and the parallel code generation *templates*.

5.1. The Stanford DASH Machine. The Stanford DASH machine[11] is a cache-coherent shared-memory multiprocessor. It uses a distributed directory-based protocol to provide cache coherence. It is organized as a group of processing clusters connected by a mesh interconnection network. Each of the clusters is a Silicon Graphics 4D/340 bus-based multiprocessor. The 4D/340 system has four processing nodes, each of which contains a 33MHz R3000 processor, a R3010 floating-point co-processor, a 64KByte instruction cache, 64KByte first-level write-through data cache, and a 256KByte second-level write-back data cache. Each node has a peak performance of 25 VAX MIPS and 10 double-precision MFLOPS. Cache coherence within a cluster is maintained at the level of 16-byte lines via a bus-based snoopy protocol. Each cluster also includes a directory processor that snoops on the bus and handles references to and from other clusters. The directory processor maintains directory information on the cacheable main memory within that cluster that indicates which clusters, if any, currently cache each line.

5.2. Run-Time Support. We have developed a user-level run-time support to allow dynamic light-weight task creation and synchronization under a Work Crew model paradigm. Relevant issues in the design of such run-time support are both load-balancing and task grain size. For its simplicity we use a variant of the simple load-based task inlining mechanism despite some of the drawbacks mentioned by Mohr [14]. Each process may spawn as many task as desired, constructing and putting in a globally accessible work list a *task descriptor*. This descriptor holds the address and the arguments of the function to be executed. Task descriptors are seized and executed by idle processes filling the appropriate return value slots. We have implemented a very simple task spawning policy based on local and global load information. Rather then examining system load by the length of its task queue, as is the case in [21], we use a unitary length queue per processor, i.e. a table, as suggested by Weening [22] and a global shared variable indicating if there idle processes based on which either spawning or task inlining is performed. The implementation is in spirit close to Leapfroging [21]. However, we have implemented more task spawning/synchronization schemes to efficiently meet the diversity of applications' computation paradigms.

5.2.1. Recursive computations. At each invocation multiple tasks are potentially spawned and added to a local queue in the computations' activation frame[3] (see code generation) upon which synchronization might be enforced. Partial and terminal synchronization is implicit enforced by *task descriptor* synchronization. In the case of no-value-returning recursive computations a simpler scheme is used that avoids the burden of dynamic *task descriptor* allocation. In this last case a global initially allocated table is used. Each processor fills its own designed entry in this table with the description of the spawned task. Termination synchronization is now performed by a global shared counter. In the Barnes-Hut case study this recursive computation paradigm was used in the computation of the center of mass.

5.2.2. Loop computations. For the cases of the simple loop level parallelism a more efficient scheme can be devised. Instead of filling *task descriptors* a self-scheduling [9, 12, 15] strategy might be used where each processor "steals" a set of loop iterations which executes in a irrevocable fashion. The current code generation scheme allows for the implementation of any of the above mentioned iteration-space partition.

[3] The queue local declarations and manipulation is automatically generated.

A more challenging scheme (not supported in the current system implementation) would call for tentative assignment of loop iteration based on any of the self-scheduling strategies, but allowing processors to dynamically release previously assigned iterations as a mean to attain better load balancing.

5.3. Code generation. The code used was generated following parallelization *templates* depending on the specific computation scheme at hand. These *templates* generate, for each parallelizable function, a set of acompaining functions, i.e. wrappers, responsible for constructing and interpreting the computation arguments from a *task descriptor*. For the sakeness of this experiment we have "hand-generated" the parallel program code given the information on the commuting operations. This generation process can be easily automated as suggested by the complete example of one of these *templates* presented in Appendix A.

5.4. Results. Tables 1 and 2 present experimental results for several number of processors and number of bodies in terms of total execution time for 4 iterations. Both tables have wall-clock timings and the Speedup relative to each's code sequential version. Figure 7 plots the parallel Speedup and Efficiency for the code parallelized using commutativity knowledge of the force computation step. Results are with respect to the object-based sequential version on the code.

The overall execution time of the parallel code generated using commutativity information is always faster than the execution time for the SPLASH2 code except for the 16K particles case. Even at 16K particles the SPLASH2 version is at most 20% faster. This is a very encouraging result considering the amount of effort and sophistication of the optimizations in the SPLASH2 version[20].

Nbody	Processors									
	1		2		4		8		16	
	Time	Spdup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1024	7.94	1.0	4.05	1.96	2.19	3.63	1.38	5.75	0.92	8.63
2048	20.46	1.0	10.70	1.90	5.72	3.57	3.37	6.08	2.18	9.39
4096	51.23	1.0	26.82	1.91	14.20	3.61	8.09	6.33	5.16	9.93
8192	131.2	1.0	64.10	2.05	33.88	3.87	15.59	6.69	11.56	11.35
16384	301.4	1.0	146.9	2.05	77.56	3.88	45.60	6.61	26.13	11.54

Table 1. Data results on the DASH multiprocessor for the "hand-generated" parallel code for the force computation step. (Time in seconds)

Nbody	Processors									
	1		2		4		8		16	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1024	9.15	1.0	4.58	1.998	2.46	3.720	2.39	3.828	2.07	4.420
2048	22.36	1.0	11.78	1.898	6.04	3.702	3.65	6.126	4.08	5.481
4096	55.51	1.0	27.40	2.026	14.95	3.713	8.06	6.887	5.90	9.408
8192	127.1	1.0	65.67	1.935	35.79	3.550	19.73	6.439	11.60	10.952
16384	277.1	1.0	140.5	1.972	76.70	3.613	39.63	6.992	22.69	12.212

Table 2. Data results on the DASH multiprocessor for the barnes code in the SPLASH2 benchmark. (Time in seconds)

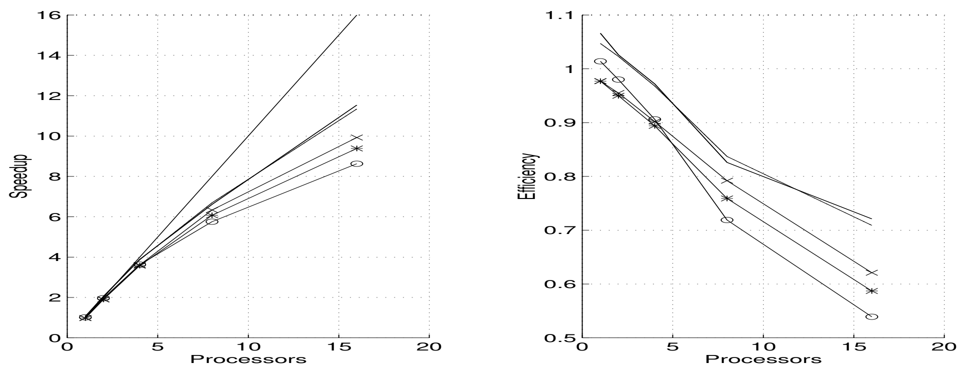

Fig. 7. Barnes-Hut Algorithm Speedup and Efficiency.

6. Conclusions. Programmers do exploit commuting operations to effectively parallelize existing sequential application codes. Observed transformations are beyond the reach of existing parallelizing compiler as they violate the sequential programs' data dependencies. We have parallelized an existing and complex application for a shared-memory multiprocessor exclusively using information derived from the commutativity analysis of its sequential program. Experimental results do show the practical importance of exploiting these opportunities making commutativity analysis a promising technique.

Acknoledgements. We would like to thank Dr. Klaus Schauser for valuable comments on an early version of this report.

REFERENCES

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

[2] J. Barnes and P. Hut. A hierarchical O(NlogN) force-calculation algorithm. Nature, 324(4), pp 446–449, December 1986.

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

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

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

[6] W .L. Harrison. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, October 1989.

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

[8] L. Hendren, and A. Nicolau. Parallelizing programs with recursive data structures In *IEEE Transacions on Parallel and Distributed Systems, 1(1),1990*

[9] S. Hummel, E. Schonberg and L. Flynn. Factoring: A Practical and Robust Method for Scheduling Parallel Loops *Technical Report* 74172, IBM Research Divison, 1987.

[10] D. Kuck, Y. Muraoka, and S. Chen. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Transactions on Computers*, C-21(12):1293–1310, December 1972.

[11] D. Lenoski. The Design and Analysis of DASH: A Scalable Direcoty-based Multiprocessor. Department of Computer Science, Stanford University, Technical Report CSL-TR-92-507, February 1992.

[12] S. Lucco. A Dynamic Scheduling Method for Irregular Parallel Programs. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992

[13] D. Maydan, J. Hennessy, and M. Lam. Efficient and exact data dependence analysis. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[14] E. Mohr, D. Kranz and R. Halstead. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs In *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2 (3), pp 264–280, July, 1991.

[15] C. Polychronopoulos and D. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439, December 1987.

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

[17] M. Rinard and P. Diniz. *Commutativity Analysis : A New Technique for Automatically Parallelizing Serial Programs*. Tech. Rep. TRCS94-16, University of California, Santa Barbara, CA, August 1994.

[18] J. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. Tech. Rep. Computer Science Laboratory, CSL-TR-91-469, April 91.

[19] J. Singh and J. Hennessy, *Finding and Exploiting Parallelism in an Ocean Simulation Program: Experience, Results and Implications*, In Journal of Parallel and Distributed Computing, vol. 15, no. 1, (1992), pp. 27–58.

[20] J. Singh. Parallel Hierarchical N-Body Methods and their Implications for Multiprocessors. PhD thesis, Tech. Report No. CSL-TR-93-565, Stanford, CA, 1993.

[21] D. Wagner and B. Calder. Leapfrogging: A Portable Technique for Implementing Efficient Futures. In *Proceedings of the SIGPLAN '93 Conference on Principles and Practive of Parallel Programing*, July, 1993.
[22] I. A. Mason, J. D. Pehoushek, C. Talcott and J. Weening. Programming in QLisp. Department of Computer Science, Stanford University, Technical Report CS-90-1340, 1990.
[23] W. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.
[24] M. S. Warren and J. K. Salmon. Astrophysical N-Body Simulations Using Hierarchical Tree Data Structures. In *Proceedings of SuperComputing '92 IEEE Comp. Soc.* pp 570–576, 1992.
[25] S. Woo, M. Ohara, E. Torrie, J. Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. June 1995.

Appendix A - Code generation templates. In this appendix we present a complete example of the code generation template for the case of a value-returning recursive computation. The original source code is shown in Figure 8 and consists of the recursive traversal of a graph. The user defined data structure consists of structure nodes with an integer value field `val` and `CHILD_NUM` children nodes. The program performs a depth-first traversal of the data structure.

```
/* Function "fct_name" source body */
int "fct_name"(nodeptr p)
{ int i, ret;
     if(p == NULL)
          return 0;
     ret = p->val;
     for(i=0; i < CHILD_NUM; i++)
          ret += "fct_name"(p->field[i]);
     return ret;
}
```

Fig. 8. Source code of a recursive interger-value-returning computation.

The analysis of this code reveals that all recursive computations can be concurrently executed.[4] In this particular case no synchronization at the user-level data structure is need as all accesses perform read operations. The Figure 9 below illustrates the corresponding generated code.

The generated function `"fct_name"_spawn` is responsible for allocating a task descriptor and filling the appropriate computation information - the arguments and the function pointer. An idle process seizes a task from a globally available table and executes the function indicated in the task descriptor. This function, which is by construction the `"fct_name"_wrapper` is invoked having as argument the task decriptor iteself, and is responsible call the `"fct_name"` function with the arguments extracted from the task descriptor. Synchronization is achieved by depositing a value in the first task descriptor argument slot and signaling in the `sync` flag. The supporting functions `add_task` and `remove_task` implement simple linked-list task descriptor insertions and removals. Also used is the `steal_task` internal function which allows a process to look for work to perform while waiting for its spawned tasks to complete.

[4] Notice that in the presence of cycles in the user data structure both the sequential and the concurrent codes presented will enter an infinite-loop.

```
void "fct_name"_wait(tasklist list)
{ task tp;
    while((tp = (*list))!=NULL){
    while(tp->sync == FALSE)
        steal_task();
    remove_task(list,tp);
}

void "fct_name"_wrapper(task p)
{ /* function call extracting args from task descriptor */
    p->args[0].ival = "fct_name"((nodeptr)(p->args[0].pval);
    p->sync = TRUE;
}

void "fct_name"_spawn(tasklist list, char *p)
{ task ptr;
    ptr = new_task(); /* allocates new task descriptor */
    ptr->sync = FALSE;
    ptr->fct_addr = "fct_name"_wrapper;
    ptr->args[0].pval = (char*)p; /* wraps args */
    add_task(list,ptr); /* adds to local synch list */
    work_pool[this_proc] = ptr;
}

/* Function "fct_name" new body */
int "fct_name"(nodeptr p)
{ int i, ret; task list = NULL;
    if(p == NULL)
        return 0;
    ret = p->val;
    for(i=0; i < CHILD_NUM; i++){
        while((idle_counter) AND (work_pool[this_proc]));
        if(work_pool[this_proc] == NULL)
            "fct_name"_spawn(&list,p->field[i]);
        else
            ret += "fct_name"(p->field); // inline
    }

/* synchronization on task list */
    if(list != NULL)
        ret += "fct_name"_wait(&list);
    return ret;
}
```

Fig. 9. Code generation template for recursive interger-value-returning computations.