Data Structures for Irregular Applications *

Katherine Yelick, Soumen Chakrabarti, Etienne Deprit,
Jeff Jones, Arvind Krishnamurthy and Chih-Po Wen
Computer Science Division
University of California at Berkeley

Abstract

Parallelization of any large application can be a difficult task, but when the application contains irregular patterns of communication and control, the parallelization effort is higher and the likelihood of producing an efficient implementation is lower. The kinds of data structures that appear in irregular applications, for example, trees, graphs, and sets, do not have simple mappings onto distributed memory machines. We are building a library of such distributed data structures that use a combination of replication and partitioning to achieve high performance. Operations on these structures cannot be efficiently implemented as atomic operations, because of the latency of interprocessor communication. We propose a *relaxed* consistency model for these data structures, which is analogous to a weak consistency model on shared memory. This allows for clean, simple interfaces on the objects, but admits low latency, high throughput implementations. We demonstrate these ideas using a few data structure examples, each of which is being used in at least one application.

1 Introduction

Parallel programs may exhibit at least three different kinds of irregularity. The first kind of irregularity appears as *irregular control structures*, namely conditional statements, which make it inefficient to run on synchronous programming models such as that provided by an SIMD machine. A second kind appears in the form of *irregular data structures*, which include unbalanced trees, graphs, and unstructured grids. These data structures lead to dynamic scheduling and load balancing requirements, since it is often impossible to predict the amount of computation that will be associated with a given data structure. The third type of irregularity is *irregular communication patterns*, which lead to nondeterminism, since one cannot predict the order in which communication events will occur. Communication irregularity is typically caused by either data or control irregularity, and the three together define the most challenging class of irregular problems.

*Contact: `yelick@cs.berkeley.edu`. This work was supported in part by the Advanced Research Projects Agency of the Department of Defense monitored by the Office of Naval Research under contract DABT63-92-C-0026, by Lawrence Livermore National Laboratory, by the Semiconductor Research Consortium, by AT&T, and by a National Science Foundation Research Initiation Award and Infrastructure Grant (number CDA-8722788). The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Implementing irregular applications on distributed memory machines requires mapping their major data structures across the machine. Whether or not the programming model provides a single address space for accessing memory, the hierarchy cannot be ignored when trying to achieve high performance. We are building a *multi-ported object library*, called *Multipol*, of distributed data structures. The library hides much of the complexity of the data structures, including concurrency control and consistency management, inside the data abstractions. The library approach balances the trade-off between performance, which leads to machine specific implementations, and the conflicting goal of portability. While library implementations may be tuned to a particular architecture, the client application is portable across any machine on which the library is supported.

Two basic techniques for implementing shared distributed data structures are replication and partitioning. Replication gives high throughput for read-only operations at the cost of consistency traffic for mutating operations. Partitioning has the opposite characteristics. Our library uses a combination of both, including hybrid implementations in which an object is divided into a number of components and those are replicated on a subset of processors.

The remainder of this paper is divided as follows. Section 2 presents the programming model used in our applications, and one which is supported through the library abstractions. Section 3 describes the library interfaces, and introduces a notion of *relaxed* consistency, and Section 4 gives some examples of distributed data structures. We conclude in Section 5 with some observations and a status report on the work.

2 The Programming Model

The programming model that is used in our irregular applications is an event driven model. Each processor repeatedly executes a scheduling loop, which looks for work in one or more scheduling queues. This style, which is commonly used on shared as well as distributed memory machine, works well because the number and kind of tasks is not known until run-time [10, 8]. It adapts to input dependent load requirements, and uses fewer resources than if a new thread (complete with its own context, which means a stack) were created which each unit of parallel work. The semantics of task scheduling is that each one runs to completion, so multiple stacks are not needed.

An application may use multiple threads of control at different levels. The coarsest level threads are *anyone computes* tasks, which are units of work that can be efficiently scheduled on any processor. At the second level are *owner computes tasks* which are units of work that can be independently scheduled, but for which the communication to computation ratio is high enough that the task should be scheduled on a particular processor. Consider, for example, a column oriented sparse Cholesky factorization: the processing of a column may be done on any processor, whereas the updates to other columns (to the right) should only be done by the processor owning the updated column. Anyone computes tasks are used to increase parallelism and improve load balance, whereas anyone computes tasks are used to improve locality.

Not all tasks should be run to completion. An exceptions is a task that requires global synchronization or completion of a remote operation. For example, in a symbolic algebra problem (the Gröbner basis computation), a global set of polynomials is protected by a lock. A task that attempts to acquire the lock and fails, is de-scheduled so that another task may use the processor. This is done by hand in our current implementations, and is used only for few performance critical operations. It requires that long latency operations should be non-blocking, so that a task may test to see whether the operation can be completed, and explicitly yield its processor if not.

The finest level of threading comes from overlapping communication with computation. Although we believe this will become increasing important on future machines, it does not play a significant role in the performance of our applications. A small amount of overlap is used by explicitly pre-fetching data before is will be used, or writing remote data without waiting for the write to complete.

Each of these kinds of threading appear in applications. In the next section we consider the affect that these considerations have on designing distributed data structure interfaces.

3 Relaxed Objects

To efficiently implement distributed data structures and still make sense of their interfaces, we use a *relaxed* consistency model. Informally, this says that an operation need not take affect simultaneously on all processors, but that every processor will see consistent versions of the data structure, in the sense that an operation will never appear to be partially performed. The analogy is to weak memory models, in which read and write operations may be reordered in the network, and guarantees about completion of the memory operations are only made at synchronization points [1, 2].

Extending this idea to arbitrary data structures, we assume that each object has an associated set of normal operations, and optionally, a set of synchronization operations. The synchronization operations are not locks, which would prevent operations from being performed, but they force outstanding normal operations to take effect. Thus, they are more like a `fence` than a lock. There are two basic varieties of the synchronization operations: local ones, which synchronize one processor's view of the shared object, and global ones, which synchronize all processors' views. For example, consider a set abstraction in which multiple processor are performing inserts. A global synchronization, which would be invoke by all processors, would ensure that every processor would have the same view of the set. A local synchronization, invoked by a single processor, only guarantee that it can see all of the performed inserts. Both types of synchronization make sense in certain contexts.

In this abstract we do give only this informal definition of relaxed object, along with a number of examples. The notion can be made more precise by extending the definition of *linearizability*, which requires that operations appear to take effect atomically, sometime during the invocation of the operations [6]. Whereas linearizability requires that there exists a global total order consistent with the invocation order, a relaxed object only requires a global partial order, since different processors may observe operations taking place in different orders.

4 Examples

As noted earlier, distributed data structures may use a combination of replication and partitioning. Some replicated data structures use consistency protocols that are similar to those found in shared memory implementations, either in hardware [1, 5] or in software [7, 2]. Programs written using shared distributed data structures have some of the programmability advantages found with shared memory. However, distributed data structures allow for the use of semantic information in the implementation: a priority queue may not strictly adhere to the priority order, or a replicated set of objects may not be kept consistent at all times. In addition, data structures may be distributed based on natural boundaries, rather than a system-defined boundary such as a page. Both of these factors lead to performance advantages.

In the remainder of this section we discuss some of the design issues in the library by considering a few concrete examples. In addition, two detailed interfaces for the multiset and task queue are given in the appendices. The key observation to make about the example is that the objects are treated semantically like a shared data structure, but because we relax the constraints on exactly *when* operations take effect, or *which* processors observe those effects, more efficient implementations are allowed.

Multiset The first example is a multiset: a container of unordered objects in which duplicates are allowed. This use used in the Gröbner basis problem, in which a key data structure is a multiset of polynomials [3, 4]. New polynomials are created and reduced with respect to the current multiset; if the reduced form of the polynomial is not zero, it is added to the multiset. (Roughly, *reduction* is the process of subtracting multiples of the polynomials in the set from the one being reduced.) Reductions often reduce a new polynomial to zero, which means that reading elements of the multiset is much more common that inserting into the set. This leads to a replicated design for the multiset. Furthermore, many reduction steps can be done using only a subset of the polynomials; the entire set is needed only to confirm that the polynomial cannot be reduced further, just before it is added to the multiset. This observation allows the replicas of the multiset to be kept only loosely synchronized, with a processor forcing the local copy to be validated just before an addition. The code, which is outlined below, uses a test-and-test-and-set style for reducing a polynomial and validating a local replica. Each of the reduction steps uses polynomials in the local multiset to reduce `p`; if `p` reduces to zero at any step, it is thrown away and the entire computation restarted with a new polynomial.

```
create a new polynomial p
reduce p by (local) multiset
while multiset is not valid
    validate
    reduce p by multiset
lock the multiset
    while multiset is invalid
        validate
        reduce p by multiset
    add p to multiset
```

Task Queue The second example of a relaxed object is a *task queue*, which is a container for holding and distributing tasks. In a sequential environment, a task queue would be a priority queue, but in a parallel environment, adhering to strict priorities leads to contention for the queue and high communication overhead. In many applications, strict priorities, or strictly FIFO behavior if there are no priorities, is unnecessary. Thus, the task queue is really an unordered collection of objects in which any priorities are used as hints; the priorities are locally, but not globally, observed. The implementation is a set of local priority queues distributed across processor.

Although users may store any kind of object in the task queue, there are two features of that distinguish it from another kind of container. The first is that implementation attempts to keep all queues partially full, so that a processor will not have to wait for a task to become available. The second is that a termination detection protocol is built into the task queue, so that the user can query the task queue to see if all tasks have been completed. The termination detection is part of the task queue abstraction, because there are states in which a tasks may be in transit, and therefore no work is visible in the queue.

Time Warp State The notion of a *time warp system* for doing speculative simulation was introduced by Reiher and Jefferson [9]. The essential data structure is the simulation state, which is partitioned and distributed across processors. For example, in circuit simulator, the circuit is partitioned into subcircuits based on connectivity defined by direct voltage connections [11]. Although the contents of the state, its partitioning, and the algorithms for simulating a time step of a partition are specific to the simulation problem, the basic state operations are the same for any speculative simulation. The operations on the state are reading a partition state, reading the outputs of a neighboring partition, and writing a new state. In addition, the user must provide a function for speculating values and tolerance levels that determine whether a step must be redone if the actual values do not match the predicted ones.

Hidden within that state is a system for saving multiple time step states, rolling back states as needed, and garbage collecting old states when rollbacks to that point are no longer possible. One rollback can lead to a chain of others, so the problem of determining whether a state can be thrown away, and the implementation of the rollbacks for simulations at various stages is non-trivial. In this example, the relaxed nature of the data structure comes from the speculation: the current value of the state is not necessarily seen when the object is read, but rather some approximate is used. In the end, only those steps based on legitimate read value will have any effect on the result.

The simulation algorithm is sketched below.

```
pick a local subcircuit L to be simulated at time T
    read inputs to L at T
    evaluate L at T
    update state of L and outputs
```

In the second line, the inputs to `L` may have been supplied by the neighboring subcircuits, or it may have to be speculated. Even if the inputs are the result of a a simulation step, that step may itself be based on speculation. The performance of the simulation depends on prioritizing less speculative executions over more speculative ones; this prioritization is implicit in the choice of a subcircuit. The machinery for speculation and rollbacks is hidden, since speculation happens automatically when the desired inputs are not available, and rollbacks happens when the update changes a previously predicted value.

Bipartite Graph A final example of a relaxed object comes from an electromagnetics problem, in which the main data structure is a bipartite graph. The computation works in phases, reading one half of the graph while updating the other half, synchronizing, and then switching to the opposite side of the graph. Nodes of the graph are spread across processors; the user does not have to ask about the specific layout, since any communication is done within the data structure. Operations on the graph allow for iterating over the right or left nodes, writing to individual nodes, and forcing outstanding writes on either half of the graph to take effect. Thus, updates may happen any time between their initiation and the synchronization point at which the writes are forced.

5 Summary

The Multipol library is an ongoing project. To date, we have a small set of data structures and complete applications that use them. The implementation have all been done on a CM5 multiprocessor and performance of the applications is very good. The multiset is used in the Gröbner basis problem and speedups on sufficiently large problems scale to the maximum number of processors available. Similarly, the speculative timing simulator, PARSWEC, show nearly linear speeds up on large circuits. The task queue is used in a number of applications, including a search problem, an eigenvalue computation, the Gröbner basis compuation, and a sparse Cholesky factorization.

One of the goals of the project is code reuse between applications and within the library. A common abstraction is replicated memory blocks that have an associated consistency protocol. This is useful as part of the multiset implementation, and also appears to be part of a distributed hash table, with cached entries, and a distributed tree, in which the top levels are replicated to avoid contention. The library will grow as new data structures are identified within applications. An important part of any data structure design is understanding the operation load placed on it by an application. With distributed data structures, even more than sequential ones, there are many engineering trade-offs that can only be determined by evaluating both the application needs and the architectural parameters.

References

[1] Sarita V. Adve and Mark D. Hill. Weak ordering–a new definition. In *17th International Symposium on Computer Architecture*, April 1990.

[2] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 7(4):152–164, November 1989.

[3] Soumen Chakrabarti. A distributed memory gröbner basis algorithm. Master's thesis, University of California, Berkeley, Berkeley, CA, 1992.

[4] Soumen Chakrabarti and Katherine Yelick. Implementing an irregular application on a distributed memory multiprocessor. In *Principles and Practice of Parallel Programming*, San Diego, CA, 1993. to appear.

[5] Kaourosh Gharachorloo, Daniel Lenoski, James Laudon, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th International Symposium on Computer Architecture*, pages 15–26, 1990.

[6] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, pages 463–492, July 1990.

A preliminary version appeared in the proceedings of the 14th ACM Symposium on Principles of Programming Languages, 1987, under the title: *Axioms for concurrent objects.*

[7] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[8] Steven Lucco. A dynamic scheduling method for irregular parallel programs. In *Proceedings of the Conference on Programming Language Design and Implementation.* ACM Sigplan, 1992.

[9] Peter Reiher and David Jefferson. Dynamic load management in the time warp operating system. *Transactions of the Society for Computer Simulation*, 7(2):91–120, June 1990.

[10] Eric Roberts and Mark Vandevoorde. Work crews: An abstraction for controlling parallelism. Technical Report 42, Digital Equipment Corporation Systems Research Center, Palo Alto, California, 1989.

[11] Chih-Po Wen. Parallel timing simulation on a distributed memory multiprocessor. Master's thesis, University of California, Berkeley, Berkeley, CA, 1992. Master's Report.

Appendix A: Multiset Interface

`int MultiSet_create (void (*freeEltProc)(), int (*sizeofEltProc)(),`
`void * (*packEltProc)(), void * (*unpackEltProc)());`

Creates a new multiset that will use the given procedures for packing, unpacking and deallocating elements, and for testing their size. These operations show through the interface, because we would like to arbitray objects, which may be linked structures, to be contained in a multiset.

`int MultiSet_insert (MultiSet *s, void *elt);`

Inserts a new element into the multiset.

`int MultiSet_delete (MultiSet *s, void *elt);`

Deletes a given element from the multiset.

`int MultiSet_modify (MultiSet *s, void *oldelt, void *newelt);`

Modify an element of the multiset. Because replication and consistency are handled as part of the multiset abstraction, the multiset must be notified when an element is changed. An alternate interface would allow the user to pass a mutating procedure, rather than a new object. The only constraint is that the modifications must be done atomically, since the multiset itself does not ensure atomicity of the element operations.

`int MultiSet_isValid (MultiSet *s);`

Check to see whether the current view of the set is valid, i.e., whether the local copy contains all the elements.

`int MultiSet_validate (MultiSet *s);`

Forces the current view to become up to date. This guarantees that the local view will have more of the up-to-date elements after invocation than before, but does not guarantee that the view will be valid.

`MultiSet_for_elements (MultiSet *s, MultiSet_indexer *i)`

Iterate over a subset of the elements in the multiset. If the local copy is valid (as indicated by the `is_valid` operation, then all elements of the multiset will be covered. Because we are working in C, the iterator is really a macro, which must be invoked with a variable index of type `*MultiSet_indexer`, e.g.,

```
MultiSet_for_elements (s,i)
    { code for loop body }
```

Appendix B: Task Queue Interface

`TaskQ *TaskQ_create(int task_size);`

Creates and returns a `TaskQ`. All tasks in a given queue must have the same size, which is given by `task_size`.

`TaskQ_status TaskQ_enq(TaskQ *q, void *data [, int priority, int compute, int communicate ]);`

Enqueue a task. Assumes that the size of data matches the size given when `q` was created. Returns OK (0) if the enqueue succeeds, FAILED (1) if there is no space available, and TERMINATED (-1) if q has already terminated. Task are scheduled by preferring local tasks over remote ones.

There are three optional parameters:

- Priority is a hint for ordering dequeues; higher priority tasks are preferred over lower priority ones. Priorities are not strictly observed, since a processor will always dequeue a task that is local before one that is remote.
- Compute is a hint of the task granularity (e.g., in milliseconds).
- Communicate is the penalty incurred by migrating this task in milliseconds. A special constant `DONT_MOVE` can be supplied to ensure that the task will not be migrated.

`TaskQ_status TaskQ_deq(TaskQ *q, void *data);`

Dequeue a task, setting the `data` parameter to the dequeued tasks and returning a status: OK if the enqueue succeeds, and TERMINATED if q has already terminated.

```
TaskQ_status TaskQ_enq_atP(TaskQ *q, int proc,
    void *data [, int comp, int priority, int comm] );
```

Similar to `TaskQ_enq`, but the task is enqueued at a specified processor.

`TaskQ_status TaskQ_terminated(TaskQ *q);`

Returns TERMINATED if the TaskQ is empty, OK otherwise.

`TaskQ_status working_locally(TaskQ *q);`

Called when the processor is going to be doing task work locally. This is used to keep the TaskQ from terminating if it becomes empty. For termination to take place, any processor that has called `working_locally` must then call `done_locally`.

`TaskQ_status done_locally(TaskQ *q);`

Called when the processor is idle, i.e., it is clear of any task in its local work space. This ensure that the processor is ready to terminate.

`TaskQ_status TaskQ_load(TaskQ *q, int *tasks, int *comp);`

Query the remaining amount of local computation left on this processor, based on the comp amounts given in the enqueues. This can be used to do granularity adjustments in the application.