Using Atomic Data Structures for Parallel Simulation

Paul S. Barth*
M.I.T. Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139
barth@au-bon-pain.lcs.mit.edu

Abstract

Synchronizing access to shared data structures is a difficult problem for simulation programs. Frequently, synchronizing operations within and between simulation steps substantially curtails parallelism. This paper presents a general technique for performing this synchronization while sustaining parallelism. The technique combines fine-grained, exclusive locks with futures, *a write-once data structure supporting producer-consumer parallelism. The combination allows multiple operations within a simulation step to run in parallel; further, successive simulation steps can overlap without compromising serializability or requiring roll-backs. The cumulative effect of these two sources of parallelism is dramatic: the example presented in this paper shows almost 20-fold increase in parallelism over traditional synchronization mechanisms.*

1 Introduction

Simulation programs present a difficult challenge for parallel computing. Although many simulations have a large parallel component, controlling the interaction between simulation steps can become a significant bottleneck. For example, the behavior of particles in a PIC simulation are largely independent, so many particles may be simulated in parallel. However, correctly modeling particle interactions requires synchronizing particles at each time step. When the density of particles is low, this synchronization limits achievable parallelism.

This paper presents a general technique for increasing the parallelism of simulation programs. The technique combines locking techniques developed by the database community with *futures*, a write-once data structure supporting producer-consumer parallelism. This uncovers two important sources of parallelism. First, multiple operations within a simulation step can run in parallel. Second, and more important, futures allow simulation steps to overlap without compromising serializability or requiring roll-backs. The combined effect of these two sources of parallelism is dramatic: the example presented in this paper shows almost 20-fold increase in parallelism over traditional synchronization mechanisms.

2 Atomic Data Structures

Synchronizing parallel operations on shared data is a well-recognized problem. Traditional synchronization mechanisms, such as semaphores [4] and monitors [6], protect shared data by ensuring that update operations run under mutual exclusion. This type of synchronization can be a bottleneck in a parallel simulation program involving operations on a shared data structure. For example, simulating particles in a shared space may yield little parallelism, because synchronization is required to properly model particle interactions.

The database community has addressed these issues with a myriad of locking protocols that ensure atomicity while sustaining parallelism. In [2, 3], the author applies database principles to shared data structures by introducing a new class of data structures called *M-structures*. M-structure operations are *implicitly* synchronized for atomicity: reading an M-structure

*This paper describes research performed at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988. The author is supported by a fellowship from Schlumberger Technology Corporation.

element locks the element before returning the value; similarly, writing an element replaces the value and unlocks the element. This synchronization is *fine-grained*: each M-structure element has its own lock, so operations on different elements in the same data structure may operate in parallel.

Reducing the granularity of locks is a well known technique for improving concurrency. Each operation on a shared data structure acquires just the locks it needs; contention arises only when two operations require the same elements of the data structure. However, even this degree of contention can become a bottleneck. For example, if a simulation occurs in discrete steps, all updates from one step must complete before the next step begins. In this case, any local contention delays the execution of successive simulation steps. To eliminate this bottleneck, this paper describes a way to combine M-structures with another kind of atomic data structure, *futures*[5, 1].

A future is like a promissory note for a value: it represents a value that will be defined later. Futures can be passed and stored as ordinary values, but procedures requiring a future's value must wait for the value to be defined. In the examples below, the function `future` allocates a future and spawns a process that assigns it a value. The `touch` function reads the value of a future, suspending until the future's value has been assigned. Thus, futures are implicitly synchronized, like M-structures, thereby eliminating read/write races. Futures can be implemented by write-once data structures such as I-structures[1].

3 TSP and Simulated Annealing

To illustrate this technique, consider using simulated annealing to solve the traveling salesman problem (TSP). The objective of TSP is to find the shortest tour of n cities in which each city is visited exactly once, returning to starting city at the end of the tour. TSP is typically implemented as a graph algorithm, where nodes represent cities and weighted edges represent the distance between cities.

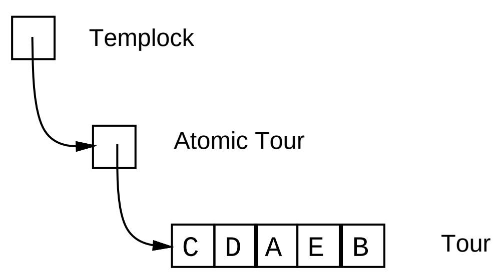
Since TSP is NP-complete, approximate techniques such as simulated annealing are used. In this approach, an initial tour of the cities is generated and then permuted to converge to a locally optimal solution. To avoid convergence on a poor local minimum, the simulation admits some permutations that increase the tour length, in hopes of entering a region containing better solutions. The frequency of such non-improving permutations is governed by the "temperature" of the simulation. Initially, the system is at a very high temperature and almost all permutations are accepted. As the simulation progresses the temperature "cools," allowing only permutations that reduce the cost of the tour.

In this version of the simulated annealing algorithm, each temperature tries several permutations of the tour in parallel. Each permutation exchanges the position of two cities in the tour. These permutations may execute in any order, as long as each updates the tour atomically. However, permutations from different temperatures must be serialized to ensure that the system stabilizes. The following program implements this algorithm:[1]

```
def tour n =  atomic M-array of cities from 1 to n
;

def anneal temp ncities nswaps =
  { templock =  atomic (tour ncities) ;
   In
    { parallel while (not_frozen temp) do
     {atomically a_tour = templock update
       { parallel for j from 1 to nswaps do
          swap two random cities } } ;
     next temp = cool temp } } ;

def swap i j a_tour temp =
 {atomically tour = a_tour update
   { old_i = tour![i] ;     %  read and lock
     old_j = tour![j] ;
     new_i, new_j = permute old_i,old_j ;
     tour![i]      = new_i; %  write and unlock
     tour![j]      = new_j } } ;
```



This program uses M-structures for synchronization and updates. The `tour` M-structure has two parts: an M-structure array (M-array) of cities, and a single M-structure cell for access control. The *atomic* function creates an access control lock for its given argument. For example, updates between temperatures is

[1] The pseudo-language used here is based on Id[7], an implicitly parallel programming language.

controlled by **templock**, another M-structure cell that references the atomic tour.

The **anneal** function takes three arguments: the initial temperature (**temp**), the number of cities in the tour (**ncities**), and the number of permutations to perform at each temperature (**nswaps**). The parallel **while** loop spawns multiple, parallel iterations until the system freezes. The **atomically** expression inside the loop is called an *atomic region.* This region uses the access control lock of the given atomic object (**templock**) to guarantee the atomicity of the region. In this case, this region consists of a parallel **for** loop that issues the swaps—atomicity means that all swaps from one temperature complete before the next temperature begins.

The **swap** function also uses an atomic region, in this case to ensure that each permutation is atomic. Without atomicity, the same city could be simultaneously moved to two different positions in the tour, yielding an invalid tour. **Swap** also illustrates Id's implicit parallelism: the statements in a block may execute in parallel unless constrained by data dependence. For example, tour positions **i** and **j** are locked in parallel, then *permute* is called, and finally **i** and **j** are unlocked in parallel. Implicit parallelism is a powerful feature of Id, but is not central to the techniques described here.

4 Synchronizing Atomic Regions

The nested atomic regions in this example highlight an important problem in parallel simulation. Although each permutation only affects a small fraction of the shared data structure, ensuring the atomicity of each permutation requires careful synchronization. The rest of this section describes three different synchronization strategies for ensuring atomicity, and compares their performance.

Barrier Synchronization: The traditional method of synchronizing atomic regions involves barriers. In this case, the body of an atomic region is surrounded by synchronization operations. These operations (typically semaphores) lock the atomic object before the atomic region begins and release the lock when the region completes. In this strategy, all parallel processes spawned inside the atomic region must complete before the lock is released.

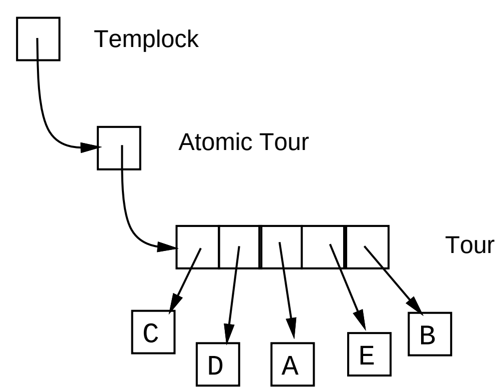
Lock-Coupling: A technique developed by the database community, called *lock-coupling*, ensures atomicity but unleashes more parallelism than barrier synchronization. In lock-coupling, the shared data is structured as a tree of locks. Every operation on the data acquires the locks in tree order, releasing the parent lock after the child is acquired.

To implement lock-coupling in this example, a lock is associated with every position in the tour array. Under lock-coupling, **swap** locks the **tour**, acquires the locks of the two cities to be updated, then releases the lock on **tour**. This allows other swaps to begin while the two cities are being updated. Similarly, **anneal** releases the temperature lock as soon as the last **swap** process has locked the **tour**.

Futures: Parallelism can be further increased by using futures. A future is a "placeholder" for a value that is being computed by some process. If a process tries to read the value of a future before it is defined, that process is suspended; when the value is written, any suspended processes are resumed.

In the TSP example, each element of the tour is a future containing a city. The **swap** function calls *touch* and *future* to read the cities and update the positions with new futures, as follows (lock-coupling operations are excluded for brevity):

```
def swap i j a_tour temp =
 {atomically tour = a_tour update
   { old_i = touch tour![i] ;
     old_j = touch tour![j] ;
     new_i, new_j = permute old_i,old_j ;
     tour![i] = future new_i ;
     tour![j] = future new_j } } ;
```



In this version of **swap**, the current cities in positions **i** and **j** are read by locking these positions and touching the futures; *permute* is called and, *in parallel*, new futures are placed in positions **i** and **j**. (Recall that Id statements execute in parallel, and the *future* function returns future before its value has been assigned.) Thus, all locks are released well before the new values for the cities are computed. In turn, the **tour** is locked only long enough for new futures to be allocated—even when **swap** operations contend for the same position! If some position becomes a "hot-spot," swaps involving other positions can run in parallel. Futures allow significant parallelism between swaps, even between swaps in different temperatures.

Experimental Results

The behavior of the TSP program was tested on a parallel simulator called GITA[8]. This simulator measures potential parallelism under many conditions. Using a graph of 100 cities, the following data reflect the performance of annealing for five temperatures, where each temperature spawns 40 swaps:

Synchronization method	Total instructions executed	Critical path	Average parallelism
Barrier	237,119	105,973	2.2
Lock-Coupling	239,424	19,240	12.4
Futures	241,424	6,417	37.6

The first column gives the total number of instructions executed, which is comparable for all three methods. Note that GITA measures *idealized* performance to demonstrate the maximum achievable parallelism. One important idealization is that memory allocation (including allocating futures) takes a single instruction. Although very low overhead allocation schemes have been developed[9], real systems will not realize this assumption. However, increasing the cost of future allocation does not significantly curtail parallelism; rather, the additional overhead is masked by parallel computation.

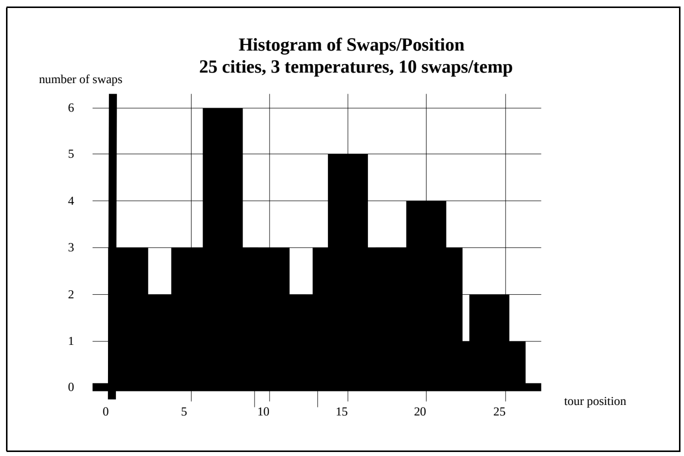
Clearly, traditional barrier synchronization yields the least parallelism of the three. Lock-coupling improves on this, allowing permutations on different elements to proceed in parallel; however, when there is contention, the access lock of the tour remains locked until the contention is resolved. Futures eliminate this bottleneck, unleashing the most parallelism.

4.1 The Critical Path of Shared Data Structures

The examples above demonstrate that atomicity barriers and futures enable a program to tolerate contention. But how close is this to optimal? Can more concurrency be wrung from this program?

To address these questions, consider the *critical path* of the tour M-array. The annealing process posts $t \times n$ swaps against the tour, where t is the number of temperatures and n is the number of swaps per temperature. The critical path of the tour is the time it takes for all swaps to complete. For the TSP program, the tour position involved in the most number of swaps gives a lower bound on the critical path of the tour.

For example, consider annealing a tour of 25 cities in three steps, where each temperature attempts 10 swaps. Thus, a total of 30 swaps are performed against the tour. The histogram below shows the distribution of swaps against the tour positions. Position 7 is swapped 6 times, the most of any position in the tour. Since each swap takes 500 cycles to update the positions, the minimum critical path for the tour is 3000 cycles (500 cycles $\times$ 6 swaps).



Figures 1 through 3 depict the history of swaps for each of the three algorithms. In these diagrams, the x axis represents time (cycles), while the y axis represents each of the 25 tour positions. For each position, horizontal lines indicate update intervals. The start of an interval indicates when a swap has acquired the position's old value, *e.g.,* when a position has been locked. The interval ends when a new value put back into the position. The three different line patterns associate the intervals with the three annealing temperatures.

Figure 1 shows the swap history when barriers are used. Note that there is no overlap between swaps: each locks and updates a pair of positions before the next swap begins. Given this serialization, the critical path of the tour cannot be less than 30×500 cycles = 15,000 cycles. The actual critical path is

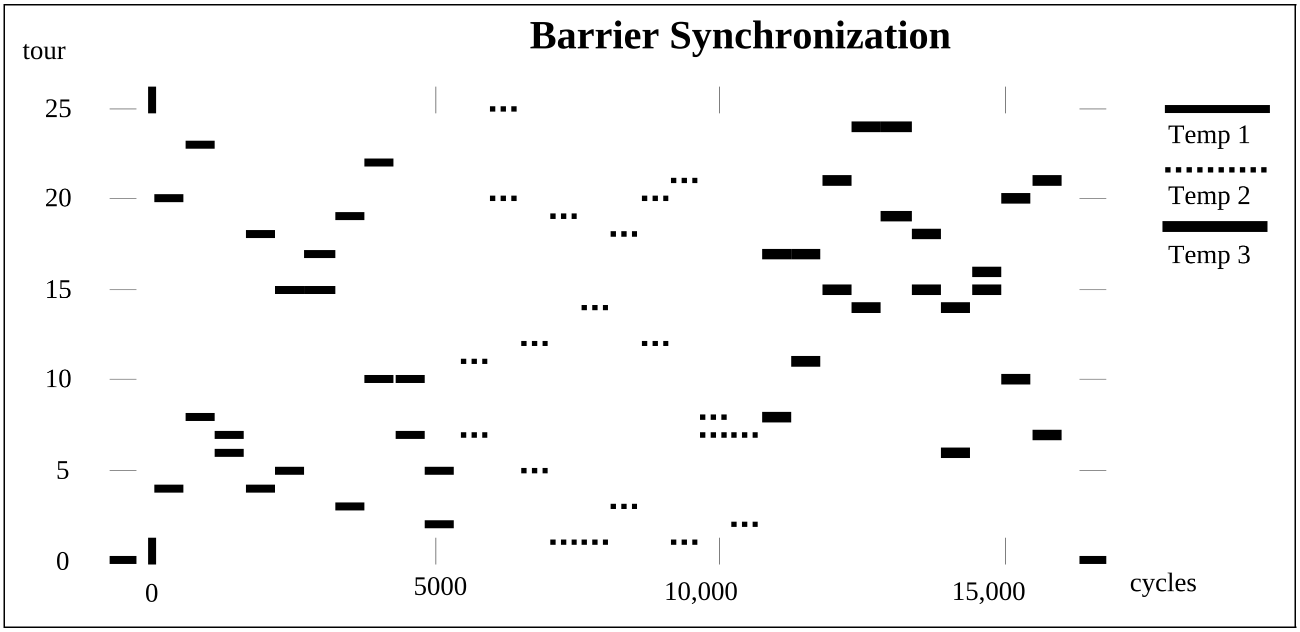


Figure 1: Swap history of the tour using barriers.

slightly longer, due to synchronization overhead between swaps.

Figure 2 shows the swap history when lock-coupling is used to synchronize atomic regions. Here, many swaps within a temperature execute in parallel. Note that unlike Figure 1, these intervals are of different lengths. Long intervals indicate contention: for example, consider positions 20, 18, and 4. The first positions swapped are 20 and 4, from interval 0 to 500. At time step 10, the program attempts to swap positions 18 and 4. Position 18 is successfully locked, but position 4 is deferred until the first swap updates its value. Note that while position 4 is deferred, the tour remains locked and no other swaps begin. The first swap unlocks position 4 at time 500; after this, the swap on positions 18 and 4 begins computing their new values, and other swaps on the tour begin.

Parallelism between temperatures is quite minor. This is because the next temperature cannot begin until the last *take* in the previous temperature completes. Thus, the maximum overlap between temperatures is 500 cycles, as occurs between cycles 3800 to 4300.

The critical path of the tour when atomicity barriers are used is 5400 cycles. This is a dramatic improvement over completion barriers, but still 80% longer than the lower bound of 3000 cycles. Note that all positions have "gaps" between update intervals; in particular, position 7 has 3 gaps totalling more than 1500 cycles, and one contention delay (as described above) from cycle 1100 to 1700.

Figure 3 illustrates how futures allow these gaps to be filled and delays to be masked. By cycle 200, all swaps for the first temperature have replaced their tour positions with futures. Similarly, the second temperature inserts all its futures by cycle 400, and the third temperature by cycle 600. In this figure, intervals indicate the time between touching an old future and filling a new one. Note that there are no gaps in the intervals, because all the swaps are initiated by time step 600. In particular, position 7 has no gaps or delays, and is finished with all 6 updates by cycle 3000.

However, delays are not eliminated; rather, they are masked. For example, the conflict between positions 20, 18, and 4 cited above is still present. Position 18 is again delayed until position 4 is updated by its swap with 20. However, no other swap is affected by this delay (unless it involves positions 18 and 4). In fact, while position 18 is delayed, swaps in temperatures 2 and 3 are proceeding!

The critical path when futures are used is 3400 cycles, just 13% above the lower bound. It is interesting that the critical path does not involve position 7. Rather, position 15 incurs enough delay to make it the

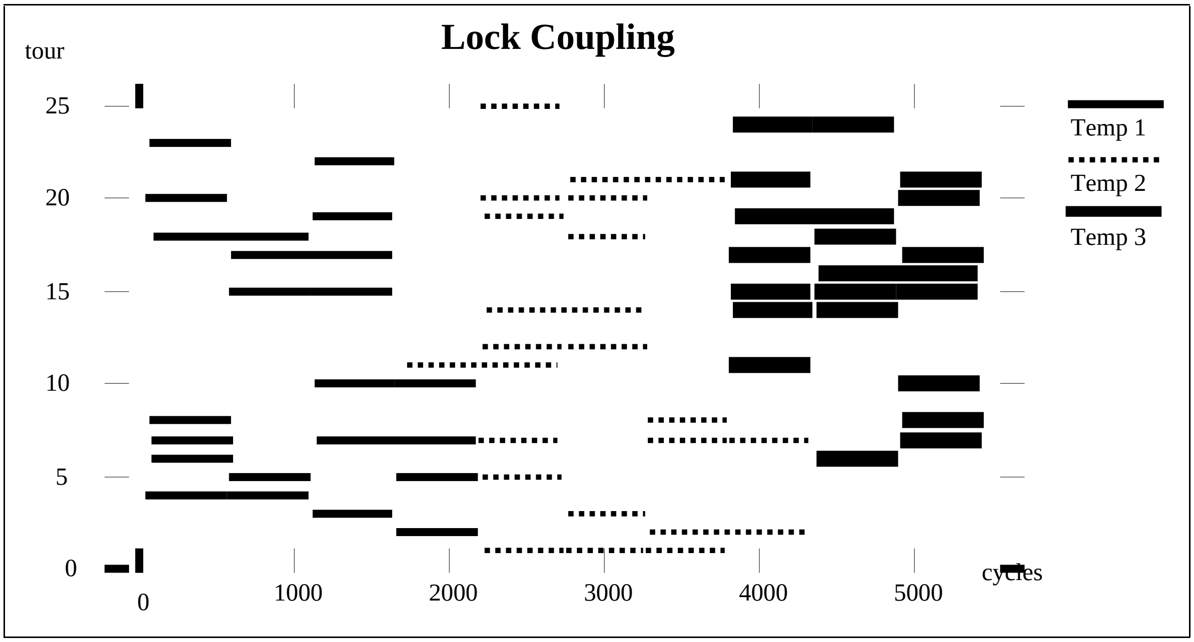


Figure 2: Swap history of the tour using lock coupling.

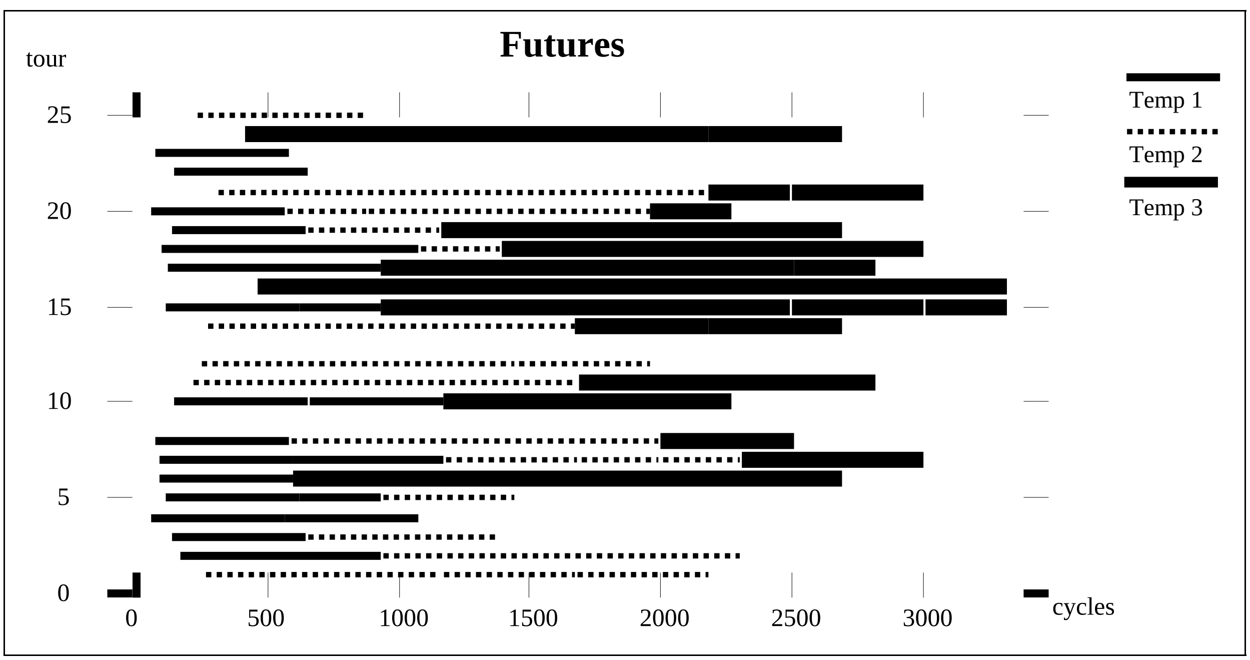


Figure 3: Swap history of the tour using futures.

critical path. This points out that futures do not guarantee that updates occur in the optimal order; rather, futures opportunistically fill gaps in the swap history. If parallel swaps fill these gaps in a different order, a shorter critical path may be achieved. Scheduling updates to achieve an optimal critical path is analogous to bin-packing, and is not considered further.

4.2 Implicit Futures

Using futures in M-structures can be viewed as another form of queuing. In this view, the return of the future is an acknowledgment that an update has been enqueued, while touching a future indicates that an update has completed. Since the updates are seen in the order received, futures implement a FIFO queue.

Given this perspective, it is natural to ask whether M-structures should provide the functionality of futures implicitly. That is, each M-structure read can implicitly touch a future for the value, and every write can replace the old future with a new one. Although technically possible, this strategy places important limitations on how M-structures are implemented.

Consider the changes required to incorporate futures into M-structure operations. First, each M-structure read must return a signal indicating that it has been enqueued on a list of processes waiting for an element. Second, reads and writes must respect a FIFO queuing protocol. With these two changes, atomic regions could synchronize on the *enqueuing* of M-structure reads, rather than their completion, thus yielding the behavior of futures. FIFO ordering ensures that reads from different atomic regions occur in order to avoid deadlock. Thus, availability is increased without rewriting a program with futures.

Unfortunately, this design places some important restrictions on the implementation of M-structure operations. First, requiring a FIFO order on reads and writes precludes simple implementations such as spin-locks and stacks. Also, the additional acknowledgment returned by M-structure reads induces additional memory traffic, or requires special capabilities of the network. These restrictions make an efficient implementation more difficult and costly, and so futures are coded explicitly.

5 Conclusions

Atomic data structures provide a clean model for synchronizing processes that share data. By coupling two types of atomic data structures—M-structures and futures—simulation programs can exploit substantial amount of parallelism. This technique is widely applicable, enabling updates to a shared data structure to be posted in constant time (the time allocate the futures), even in the presence of conflicts. The strategy maintains serializability, so algorithms requiring this property can also benefit.

The examples in this paper demonstrate the benefit of futures in the presence of cheap, fine-grained locking. Generalizing this mechanism for coarse-grained locking over large data structures is a topic of further research.

References

[1] Arvind and R. E. Thomas. I-structures: An Efficient Data Type for Parallel Machines. Technical Report TM 178, Computation Structures Group, MIT Lab. for Computer Science, Cambridge, MA 02139, September 1980.

[2] P. S. Barth. *Atomic Data Structures for Parallel Computing.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, January 1992. LCS Technical Report 532.

[3] P. S. Barth, R. S. 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, Springer-Verlag Lecture Notes In Computer Science, Volume 523*, pages 538–568, August 1991.

[4] E. W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1:115–138, 1971.

[5] R. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *Transactions on Programming Languages and Systems*, October 1986.

[6] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 10(10):549–557, October 1974.

[7] R. S. Nikhil and Arvind. *Programming in Id: a parallel programming language.* 1990. Textbook on implicit parallel programming. In preparation.

[8] R. S. Nikhil, P. R. Fenstermacher, J. E. Hicks, and R. P. Johnson. *Id World Reference Manual.* Computation Structures Group Memo, M.I.T. Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, Nov. 1989.

[9] K. M. Steele. Implementation of an I-Structure Memory Controller. Technical Report TR-471, M.I.T. Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, May 1990. (MS Thesis, Dept. of EECS, MIT).