University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Analyzable Atomic Sections: Integrating Fine-Grained Synchronization and Weak Consistency Models for Scalable Parallelism

Vivek Sarkar†
Guang R. Gao

CAPSL Technical Memo 52
February 09, 2004

†IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA. Email: vsarkar@us.ibm.com

Abstract

A key source of complexity in parallel programming arises from *fine-grained synchronizations* which appear in the form of lock/unlock or critical sections. Not only are these constructs complicated to understand and debug, but they are also often an impediment to achieving scalable parallelism because of the overhead of the underlying synchronization operations and their accompanying *data consistency* operations. In this paper, we propose the use of *analyzable atomic sections* as a parallel programming construct that can simplify the use of fine-grained synchronization, while delivering scalable parallelism by using a *weak memory consistency model*. We use OpenMP as the base programming model in this paper, and show how the OpenMP memory model can be formalized by using the *pomset* abstraction in the Location Consistency (LC) memory model. We then show how OpenMP can be extended with analyzable atomic sections, and use two examples to motivate the potential for scalable parallelism with this extension.

Contents

List of Figures

1 Introduction

One of the biggest challenges in the area of parallel programming models is defining abstractions that simplify parallel programming, while also delivering scalable performance on high-end parallel processing systems. In this paper, we focus on abstractions for *fine-grained synchronization*, and discuss issues in current synchronization constructs and their accompanying memory consistency models that result in complex programming models with limited scalability. We propose the use of *analyzable atomic sections* as a parallel programming construct that can simplify the use of fine-grained synchronization, while delivering scalable parallelism by using a *weak memory consistency model*. As a programming model abstraction, analyzable atomic sections may be realized in multiple languages. In this paper, we restrict our attention to integrating analyzable atomic sections into the OpenMP programming model [4].

Synchronization-free programs are usually simple to reason about and also amenable to scalable parallelization. The motivation for focusing our work on fine-grained synchronization is that the addition of fine-grained synchronization in the form of lock/unlock or critical sections greatly complicates the semantics and performance of parallel programs. It is usually necessary to understand implementation details of the hardware and operating system software on a parallel machine, to be able to reason about the behavior and performance impact of specific synchronization constructs.

```
C$OMP    PARALLEL PRIVATE(IX, LSCALE, LSSQ, TEMP) SHARED(SCALE, SSQ, X)
         . . .
C$OMP    DO
         DO IX = 1, 1 + (N - 1) *INCX, INCX
            . . .
         END DO
C$OMP    END DO
C$OMP    CRITICAL
         IF (SCALE .LT. LSCALE) THEN
            SSQ = ((SCALE / LSCALE) ** 2) * SSQ + LSSQ
            SCALE = LSCALE
         ELSE
            SSQ = SSQ + ((LSCALE / SCALE) ** 2) * LSSQ
         END IF
C$OMP    END CRITICAL
C$OMP    END PARALLEL
```

Figure 1: Example of OpenMP critical section from function p_dznrm2 in OpenMP 2001 benchmark, 310.wupwise_m

The OpenMP programming model [4] supports fine-grained synchronization through *critical sections* and *locks*. The example parallel region of code in Figure 1 was taken from function `p_dznrm2` in the OpenMP 2001 benchmark, `310.wupwise_m`. It consists of a parallel `DO` loop (with index `IX`), followed by a critical section. The iterations of the parallel loop update local private variables, `LSCALE` and `LSSQ`, and the critical section uses the local values to update the shared variables, `SCALE` and `SSQ`. The recommended implementation of an OpenMP critical

section is to use a single lock to guard all critical sections with the same name, and to ensure that a *flush* operation is performed on entry to and exit from the critical section [4]. In general, the flush operation must ensure that local copies of all shared data in a processor's registers and caches must be flushed to main memory. This can be a severe performance overhead for small critical sections, such as the one in Figure 1. In addition, a single lock can be a significant bottleneck when multiple processors attempt to execute distinct critical sections (with the same name) in parallel. We later discuss optimization opportunities for addressing both these performance issues.

```
#pragma omp parallel default(none) shared(lambda, atomall)
private (imax, i, ii, jj, k, a1, a2, fx, fy, fz, a1fx, a1fy, a1fz, ux, uy, uz,...)
{
  imax = a_number;
#pragma omp for
  for( i= 0; i< imax; i++) {  . . .
      a1 = (*atomall)[i];      . . .
      for( ii=0; ii< jj;ii++)
        {
          a2 = a1->close[ii];
          omp_set_lock(&(a2->lock));  . . .
S2:       a2->fx -= ux*k;              . . .
          omp_unset_lock(&(a2->lock));
        }
      omp_set_lock(&(a1->lock));
S1:   a1->fx += a1fx ;  . . .
      omp_unset_lock(&(a1->lock));
    } /* for */
}/* omp parallel pragma */
```

Figure 2: Example of OpenMP lock primitives from function f_nonbon() in OpenMP 2001 benchmark, 332.ammp_m

In contrast to critical sections, *locks* in the OpenMP programming model are explicitly managed by the programmer; the programmer has the responsibility for allocating, initializing, setting and unsetting locks. The example code fragment in Figure 2 was taken from function `f_nonbon()` in the OpenMP 2001 benchmark, `332.ammp_m`. The outer `i` loop iterates in parallel through all elements of the `atomall` array. For each such element, `a1 = (*atomall)[i]`, the inner `ii` loop iterates through a set of nearby atoms, `a2 = a1->close[ii]`. Each atom has a distinct lock to enable fine-grained synchronization. The inner loop uses `a2`'s lock (`a2->lock`) to guard updates to elements of atom `a2`, such as in statement `S2`. Likewise, the outer iteration uses `a1`'s lock to guard updates to elements of atom `a1`, such as in statement `S1`. This fine grain synchronization enables iterations of the outer loop to execute in parallel, while ensuring that conflicting accesses to individual atoms (due to updates to its neighbouring atoms in the `ii` loop) are properly guarded. However, the problem of performing flush operations still remains with explicit locks.

The previous two examples serve as motivation for us to examine the underlying memory

model assumptions of OpenMP. Though there has been much past work related to OpenMP implementation, we are not aware of any prior work that attempts to formalize the OpenMP memory model for scalable parallelism.

The rest of the paper is organized as follows. Section 2 reviews the OpenMP specification of flush operations, and shows how the OpenMP memory model can be formalized by using the *pomset* abstraction in the Location Consistency (LC) memory model [2]. Section 3 describes our proposed extension of *analyzable atomic sections* to the OpenMP programming model and discusses implementation and optimization issues. Section 4 uses the two code examples from Figure 1 and Figure 2 to illustrate the optimization opportunities that arise from the use of analyzable atomic sections. Finally, Section 5 contains our conclusions.

2 Formalization of the OpenMP Memory Model using Location Consistency

In this section, we give an outline of the memory model that appears to be implicitly assumed in the OpenMP specification, and show how it can be formalized using the Location Consistency model [2]. We will focus our attention on the OpenMP `FLUSH` directive, which implicitly defines the memory consistency model assumed in OpenMP. Flushes occur frequently in OpenMP programs, because they are performed implicitly at the end of many common OpenMP constructs such as barriers, parallel loops, parallel sections, critical sections, etc.

A `FLUSH` directive at program point P requires that a certain set of shared variables, S, must be made consistent by the executing thread at point P *i.e.,* all read and write memory accesses that occur before point P must be performed/completed before the thread can advance past P, and all memory operations that occur after point P must be performed/completed after the thread advances past P. By default, S refers to all shared variables in the OpenMP program, though the user has the option of restricting the set of variables to be made consistent. This semantics can have profound implications on the software and hardware implementation of an OpenMP program. For example, the compiler must ensure that any variable in S that is allocated to a register must be spilled to memory before point P and reloaded from memory after point P. Similarly, the hardware must ensure that all store buffers and cache lines containing variables in set S are flushed before point P, and all cached values for variables in set S are invalidated after point P. Note that a single thread's execution of a flush operation only supports consistency between the thread's local state and main memory. Global consistency can only be achieved when all threads perform a flush operation.

The OpenMP specification does not explicitly define a memory consistency model for shared variables. Existing memory models can be classified along a spectrum of strong models (*e.g.,* Sequential Consistency [3]), relaxed models (*e.g.,* Release Consistency [1]), and weak models (*e.g.,* Location Consistency [2]). All memory models in this spectrum guarantee the same semantics for *data race free* (DRF) programs [1], but provide different semantics in the presence of data races. Stronger models provide stricter guarantees on unsynchronized accesses to shared

variables at the cost of additional burden on the implementation, especially when scaling up to large numbers of processors. For example, both strong and relaxed models include the *memory coherence* assumption, which states that all updates to the same shared variable must be observed in the same order by all processors. Since OpenMP programs provide no semantic guarantees for unsynchronized accesses to shared variables (data races) and thereby do not enforce the memory coherence assumption, we believe that it is appropriate to use a weak model such as Location Consistency (LC) to formalize the semantics of OpenMP memory model so as to place no restrictions on scalable parallelism (a relaxed model was also proposed in [5] as an interpretation of the OpenMP memory model).

In the LC model, all write and synchronization operations related to a shared variable are modeled as a *partially ordered multiset* (pomset), and the semantics of a read operation is that it can receive the value supplied by any write operation in the pomset that belongs to the *most recent write* (MRW) set implied by the read operation. The partial order in the pomset naturally follows from the ordering constraints in a program such as an OpenMP program. In the absence of data races, this set will always be a singleton returning a single determinate value. In the presence of data races, the implementation is free to choose any value in the MRW set[1], thereby enabling the use of more scalable cache consistency algorithms as well as optimizations such as register allocation of shared variables. This distinguishing feature of the LC model is a consequence of the fact that it does not rely on the memory coherence assumption. Additional details on the LC memory model and the LC cache consistency algorithm can be found in [2].

In the original LC model, a pomset for a memory locations contains elements derived from write operations and a multitude of synchronization operations. The main extension needed to support OpenMP using the LC model is to define the following two rules for inserting flush operations into a pomset. First, as with other memory operations, a flush operation, F, in thread T must obey all uniprocessor dependencies *i.e.,* all prior store and flush operations performed by T must precede F in the pomset, and F must precede all later store and flush operations performed by T. Second, all flush operations performed on a memory location are assumed to be totally ordered, even (especially) if they are performed by different threads.

For an example, see Figure 3(a). In this case, it is assumed that the flush operation by thread $T1$ occurred earlier than the the flush operation by thread $T2$. The total ordering of flush operations indirectly leads to the total ordering of write accesses, as shown at the bottom of Figure 3(a), thereby ensuring that the MRW set will never be greater in size than a singleton. Figure 3(b) illustrates what would happen if the example code fragment did not contain barrier and flush operations. In this case, the two write operations are unrelated in the partial order, leading to an MRW set of size = 2.

[1] If the underlying size of the shared location is bigger than a single machine word (*e.g.,* a *complex* data type), then the result may be $\perp$, which denotes an undefined combination of multiple values in the MRW set.

Case (a): accesses to shared variables with FLUSH operations

```
    Thread T1                Thread T2
    ---------                ---------
S1: X := val1
S2: BARRIER/FLUSH       S3: BARRIER/FLUSH
                        S4: X := val2

Pomset for variable X:
----------------------
S1:write(T1,val1) ---> S2:FLUSH(T1) ---> S3:FLUSH(T2) ---> S4:write(T2,val2)
```

Case (b): accesses to shared variables without FLUSH operations

```
    Thread T1                Thread T2
    ---------                ---------
S1: X := val1
                        S4: X := val2

Pomset for variable X:
----------------------
S1:write(T1,val1)  S4:write(T2,val2)
```

Figure 3: Simple example of modeling flush operations as pomsets in the LC model

3 Analyzable Atomic Sections

3.1 Analyzable Atomic Sections: Definition and Semantics

The scope of the OpenMP ATOMIC directive is currently limited to a single statement of the form $x = f(x, expr1, expr2, \ldots)$, which enables a read-modify-write operation to be performed atomically on a single scalar location, x. Only the load and store of x are guaranteed atomic; the evaluation of expressions, $expr1, expr2, \ldots$ is not atomic (and the expressions are not allowed to refer to x). Further, the choice of function f is limited to a small set of standard operators and intrinsic functions. With these limitations, the ATOMIC directive cannot be used for commonly-occurring scenarios that need fine-grained synchronization, such as the examples in Figures 1 and 2.

Our proposal extends the current ATOMIC and CRITICAL constructs in OpenMP as follows. An *analyzable atomic section* (AAS) is a region of code that is intended to be executed atomically *i.e.,* such that any concurrent AAS either observes all or none of the execution of this AAS. We require that the addresses of all shared locations that are read or written in the AAS be computable on entry to the AAS. While we give users the option of explicitly specifying this *consistency list* of shared locations, our default approach is that the shared locations be identified automatically through compiler analysis — hence the name, "analyzable atomic section". For example, if we replace the critical section in Figure 1 by an AAS, we obtain the following code:

```
C$OMP    ATOMIC
C        CONSISTENCY LIST OBTAINED BY AUTOMATIC ANALYSIS = (SSQ, SCALE)
```

```
         IF (SCALE .LT. LSCALE) THEN
            SSQ = ((SCALE / LSCALE) ** 2) * SSQ + LSSQ
            SCALE = LSCALE
         ELSE
            SSQ = SSQ + ((LSCALE / SCALE) ** 2) * LSSQ
         END IF
C$OMP    END ATOMIC
```

By automatically identifying `SSQ` and `SCALE` as the two shared variables in the AAS, the consistency actions can be limited to these two variables. This effect is analogous to an OpenMP `FLUSH` directive with a `(SSQ, SCALE)` variable list, except that the programmer did not have to identify the variables explicitly. Another difference between a `FLUSH` directive and an AAS, is that an AAS allows the list of shared locations to include array elements and pointer-dereferenced regions as illustrated in the next example.

Now, if we replace the regions of code between the `omp_set_lock()` and `omp_unset_lock()` function calls in Figure 2 by AAS's, we obtain the following code:

```
#pragma omp atomic /* consistency list = (*a2) */
. . .  a2->fx -= ux*k;  . . .
#pragma omp end atomic
. . .
#pragma omp atomic /* consistency list = (*a1) */
. . .  a1->fx += a1fx;  . . .
#pragma omp end atomic
```

The lists of shared locations for the two atomic sections, *a2 and *a1 respectively, can be easily established by automatic analysis.

The semantics of AAS's can be established by using the Location Consistency (LC) model. Specifically, the consistency operations associated with the entry to and exit from an AAS are identical to those associated with *acquire* and *release* operations performed on the set of locations associated with the AAS. Note that this semantics enforces a weak memory consistency model. The consistency actions are limited to only the shared locations accessed within the atomic section. For accesses outside atomic sections, there is no assumption of memory coherence as is typically assumed in strong and relaxed models.

3.2 Implementation Techniques

In this section, we outline the three key issues that need to be addressed in the implementation of Analyzable Atomic Sections.

Analysis. The analysis phase examines each read and write access to shared locations in the AAS, and checks if the address is computable on entry to the atomic section. If so, the location is added to the consistency list for the AAS; otherwise, an error message to indicate that the atomic section is unanalyzable. As an option, we can give users the ability to specify the consistency list explicitly, but we will focus on the implicit case in this paper.

Lock assignment. The lock assignment phases assigns one or more locks to be used to guard the entrance to the AAS. The lock assignment should be semantically correct. There should be no "under-locking" *i.e.,* it should not be possible for another atomic section with an overlapping consistency list to execute concurrently with the current AAS. Also, while some "over-locking" may be permitted for convenience, the goal of the lock assignment phase is to satisfy the semantic requirements while exposing as much parallelism as possible. Finally, the lock assignment should guarantee that deadlock cannot occur in any execution of the OpenMP program.

Consistency actions. As indicated earlier, it is sufficient to perform a flush operation on all locations in the AAS's consistency list at the start and end of the AAS to ensure memory consistency. However, this approach can be inefficient because the consistency list represents an upper bound on the shared locations accessed in the AAS, and a specific AAS may not necessarily access all locations. Also, it is usually preferable on most shared-memory machines to spread consistency actions across other computations, rather than performing them in large bursts.

In light of these considerations, our recommended approach is to insert a *refresh* operation prior to each read access and a *writeback* operation after each write access in the AAS, analogous to the approach proposed in [2]. A *sync-writeback* operation is inserted at the end of the AAS to ensure that all pending *writeback* operations are completed before exiting the AAS. After these operations are inserted, a *partial redundancy elimination* phase can be performed to eliminate redundant *refresh* and *writeback* operations.

4 Examples of Optimization of Analyzable Atomic Sections

In this section, we use the two code examples from Figure 1 and Figure 2 to illustrate the optimization opportunities that arise from the use of analyzable atomic sections.

Figures 4(a) and 4(b) shows the result of inserting and optimizing consistency actions for the example code from Figure 1. Note that the redundant *refresh* operations on `SCALE` in the true branch of the `IF` construct has been eliminated. Also, a *writeback* operation on shared variable `SCALE` is only performed when the `IF` condition is true. Finally, if no *writeback* operation was necessary for `SSQ` in the false case, even the *sync-writeback* operation could have been moved to only the true case of the `IF` construct. Further optimization may be possible based on the target architecture. For example, by allocating `SCALE` and `SSQ` in the same cache line, it may be beneficial to combine their *refresh* and *writeback* operations.

In our second code example from Figure 2, our experiments indicate that the time spend in executing function `f_nonbon()` accounts for over 93 percent of the total time of the entire benchmark. Our measurement also indicates that an average of 118 and 68 bytes are read and written in an analyzable section (for this program), which is much smaller than the total volume of data that would be made consistent using the default OpenMP flush operation.

(a) After inserting consistency operations:

```
ATOMIC
refresh(SCALE)
IF (SCALE .LT. LSCALE) THEN
   refresh(SCALE)
   refresh(SSQ)
   SSQ=((SCALE/LSCALE)**2)*SSQ+LSSQ
   writeback(SSQ)
   SCALE = LSCALE
   writeback(SCALE)
ELSE
   refresh(SCALE)
   refresh(SSQ)
   SSQ=SSQ+((LSCALE/SCALE)**2)*LSSQ
   writeback(SSQ)
END IF
sync-writeback
END ATOMIC
```

(b) After optimization:

```
ATOMIC
refresh(SCALE)
IF (SCALE .LT. LSCALE) THEN
    refresh(SSQ)
    SSQ=((SCALE/LSCALE)**2)*SSQ+LSSQ
    writeback(SSQ)
    SCALE = LSCALE
    writeback(SCALE)
ELSE
    refresh(SSQ)
    SSQ=SSQ+((LSCALE/SCALE)**2)*LSSQ
    writeback(SSQ)
 END IF
 sync-writeback
END ATOMIC
```

Figure 4: Insertion and optimization of consistency operations in function p_dznrm2

5 Conclusions

In this paper, we introduced *analyzable atomic sections* as a parallel programming construct that can simplify the use of fine-grained synchronization, while supporting a *weak memory consistency model*. We used OpenMP as the base programming model in this paper, and showed how the OpenMP memory model can be formalized by using the *pomset* abstraction in the Location Consistency (LC) memory model. We then showed how OpenMP can be extended with analyzable atomic sections, and used two examples to motivate the potential for scalable parallelism with this extension.

Acknowledgments

This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCHC020056. We would like to thank members of the IBM PERCS project for their feedback. Zhang Yuan has directly helped in the preparation of this manuscript and performed the measurements in collaboration with Juan Cuvillo and Joseph Bryant - all are pursuing their graduate study in Gao's group at University of Delaware.

References

[1] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, pages 613–624, June 1993.

[2] Guang R. Gao and Vivek Sarkar. Location consistency - a new memory model and cache consistency protocol. *IEEE Trans. on Computers*, 49(8):798–813, August 2000.

[3] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[4] OpenMP Fortran Manual. http://www.openmp.org/specs/.

[5] Vijay S. Pai Sarita V. Adve and Parthasarathy Ranganathan. Recent advances in memory consistency models for hardware shared memory systems. *Proceedings of the IEEE*, 87(3):445–455, Mar 1999.