Atomic Section: Concept and Implementation

Yuan Zhang   Joseph Bryant Manzano Franco   Guang R. Gao
Department of Electrical & Computer Engineering, University of Delaware
Newark, Delaware 19716, U.S.A
{*zhangy,jmanzano,ggao*}*@capsl.udel.edu*

ABSTRACT

A key source of complexity in parallel programming arises from fine-grained synchronization. In the shared memory programming language OpenMP, mutual exclusion for shared data access is achieved by critical sections or locks. The critical section degrades the performance by serializing all critical section instances through a global lock, and impedes a scalable parallelism by the underlying data consistency operations; The locks, on the other hand, leave the data race detection and system resource management to programmers, thus decreases programming productivity greatly. We propose a new concept for mutual exclusion - atomic section, and its corresponding OpenMP construct - atomic_sec, which is particularly designed for both high performance and high productivity. There are three key problems on atomic section implementation in OpenMP, within which the lock assignment problem is the most challenging one. In this paper, We will first introduce the concept and semantics of atomic section. Then we will formulate the lock assignment problem, and present its implementation details. Some preliminary experimental results demonstrate the soundness of our implementation.

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 *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 and refined locks. As a programming model abstraction, atomic sections may be realized in multiple languages. In this paper, we restrict our attention to integrating atomic sections into the OpenMP programming model [1].

OpenMP is an API used to explicitly direct a multi-threaded, shared memory parallelism. It uses the fork-join model of parallel execution. All OpenMP programs begin as a single thread - master thread. The master thread executes sequentially until it encounters the first parallel region, at which point it creates a team of parallel threads. The statement block enclosed by the parallel region construct is then executed in parallel among the various team threads. When the team threads complete, they synchronize and terminate, leaving only the master thread.

```
#pragma omp parallel private(ix, lscale, lssq, temp)
                     shared(scale, ssq, x)
{
  ......
  #pragma omp for
    for(ix = 1; ix <= 1 + (n-1) * incx; ix += incx){
      ......
    }
  #pramg omp critical
  {
    if(scale < lscale){
      ssq = ((scale / lscale) ** 2) * ssq + lssq;
      scale = lscale;
    } else
      ssq = ssq + ((lscale / scale) ** 2) * lssq;
  }
}
```

Figure 1: Example of OpenMP critical section from function p_dznrm2 in OpenMP 2001 benchmark, 310.wupwise_m (equivalent C version)

OpenMP supports fine-grained synchronization through critical sections and locks. Critical sections are declared by **critical** [1] compiler construct, as shown in Figure 1, which is an equivalent C-version of Fortran code fragment taken from function `p_dznrm2` in `310.wupwise_m` of OpenMP 2001 benchmark. It consists of a parallel `for` loop (with index `ix`), followed by a critical section. The iterations of the parallel for 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 a global

[1]OpenMP supports C, C++ and Fortran. For the sake of simplicity, all the discussions in this paper are restricted to C. Readers can extend them to C++ or Fortran themselves.

lock to guard all un-named critical sections. It ensures that an implicit *flush* operation is performed on entry to and exit from the critical section [1]. In general, this flush operation must ensure that the executing thread has a consistent view for all shared data with the main memory. This can be a severe performance overhead for small critical sections. In addition, a single lock can be a significant bottleneck when multiple processors attempt to execute distinct critical sections, which have same name but no data race, in parallel. We later discuss optimization opportunities for addressing those 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];
S2:       omp_set_lock(&(a1->lock));
          a2->fx -= ux*k;  . . .
          omp_unset_lock(&(a2->lock));
        }
S1:   omp_set_lock(&(a1->lock));
      a1->fx += a1fx ;  . . .
      omp_unset_lock(&(a1->lock));
    } /* for */
}
```

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 of allocating, initializing, setting and unsetting locks, through a set of OpenMP runtime library function calls. 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 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 neighbor atoms in the `ii` loop) are properly guarded. Although lock has no memory consistency action thus has higher performance than critical sections, it leads to severe losses in productivity because it forces the programmer to understand implementation details of the hardware and operating system of a parallel machine, deal with a complex resource management problem, and worry about the situations of excessive locking (deadlock) or inadequate locking (data race).

The previous two examples serve as motivation for us to propose a new OpenMP construct for fine-grained synchronization - *atomic section*, which is designed to achieve both high performance and high productivity. The basic idea is: the atomic section is executed *as if* it is atomic. The "atomicity" means the executing thread communicates with other threads (through reading or writing shared data) only at the beginning and at the end of the atomic section; from other concurrent threads' point of view, the atomic section is like a fat sentence which is executed in one clock cycle. Atomic section expects a high performance due to its weak memory model and refined locks for mutual exclusion. At the same time, atomic section is easier to use because the compiler takes most of work, thus release programmers from system resource management and data race analysis.

The rest of the paper is organized as follows. Section 2 first proposes the concept of the atomic section in general, then presents the syntax and semantics of its corresponding construct in OpenMP. Section 3 proposes key issues and methodology to implement the atomic section in OpenMP compiler. Lock assignment is our main focus in this paper, and we will discuss its algorithms in detail in section 3.3. In section 4 we summarize the prototype implementation by examining a benchmark program in SPEC OMP2001 benchmark suite. Finally, Section 5 summarize our work and draw the conclusion. Several possible directions for future work are listed in Section ??.

2. ATOMIC SECTION

Atomic section, by definition, is a section of code that is intended to be executed atomically, and mutually exclusively with any other conflicting atomic section instances. It satisfies the following properties:

- Atomicity: All instructions in an atomic section body appear to be executed in one step. Other threads can observe either all or none of its actions, because it can affect or be affected by other threads only at the entry or at the exit. It appears indivisible with respect to other threads.

- Mutual exclusion: Two atomic section instances are "conflicting" if they are intended to access same data simultaneously. Note that an atomic section is conflicting to itself. Two conflicting atomic section instances appear to be executed in a total order, i.e., one begins after the other is finished, there is no overlap between their lifetimes. This property is sometimes referred to as "serializability" in some other literature. Note that a total order is not necessary for two un-conflicting atomic section instances.

We assume that an atomic section is not nested within another atomic section, critical section or any other atomic operation. Although the concept of atomic section can be easily extended to cover this case, we cannot observe any useful application for it.

2.1 Syntax

In OpenMP, an atomic section is declared by an **atomic_sec** construct in the format:

#pragma omp atomic_sec [*clause clause* ...] *newline*
structured block

The clause is one of the following:

cl (*consistency list*)
on (*locks*)

The **cl** clause declares shared data which will be made consistent with the main memory at the entry and exit of the atomic section (the semantics will be explained later), and the **on** clause declares a set of locks which guard the atomic section for mutual exclusion. Lock has its own name space which is separated from any other name space in an OpenMP program. While we give programmers the option of explicitly specifying those two clauses, our default and recommended approach is to let compiler identify all shared objects accessed in the atomic section, and assign the locks.

2.2 Semantics

From programmers point of view, OpenMP is a collection of threads operating on an uniformly accessed, globally addressable shared memory, and each thread also has its private memory, as described in Figure 3. The shared space and private space are exclusive to each other. In order to support atomic section, the private memory is divided into two parts. One part, called private space, contains "traditional" private data which is only visible to the associated thread. The other part, called "scratch-pad space", contains thread private copies of shared data.

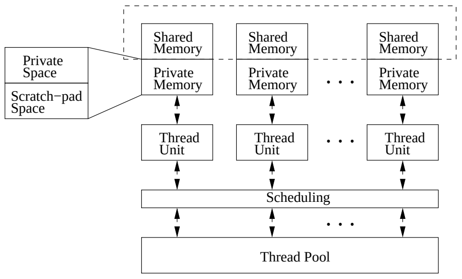

Figure 3: OpenMP Execution Model

Atomic section is intended to be executed as if it follows a five-step process:

1. Acquire the locks ("acquire" step);
2. Initially, make a copy of the shared data (as specified in the consistency list) into the scratch-pad space ("refresh" step);
3. Start the operations in the structured block, and all memory accesses are made to the thread private copy of the shared data ("computation" step);
4. Finally, when the execution of the code is finished, the private copy of the shared data will be copied back to the shared address space ("write_back" step);
5. After write_back finishes, release the locks ("release" step);

The "refresh" and "write_back" operations specify two sequence points at which the implementation is required to ensure that the executing thread has a consistency view of shared objects in the consistency list with the main memory. In other words, any other concurrent activities, either in another atomic section instant, or directly access the main memory, cannot affect, or be affected by, the computations on the private copy of the executing thread, except at two sequence points specified by "refresh" and "write_back". These two operations guarantee the *atomicity* of the atomic section, and make it behave like a "fat sentence" from other threads' point of view.

An atomic section is guarded by one or more "symbolic", "refined" locks. The "symbolic" lock differs from the real lock used in lock/unlock runtime library in such a way that it is not associated with any system resource. It is the compiler that allocates the resource for it. This improves the programming language productivity and make atomic sections easier to use. On the other hand, the lock is "refined" since each lock guarantees the mutual exclusion of this atomic section with *some of* (not all) concurrent atomic sections. We will discuss one of implementation schemes for the refined lock in section 3.3.2.

3. IMPLEMENTATION

In this section, we outline a concept-proof implementation of atomic sections in OpenMP compiler. Since it strictly follows the atomic section semantics, it may not achieve optimal performance on a specific system. Algirithms discussed here therefore should be tailored and optimized to fit into the real system. Note that although we give users the ability to explicitly specify the consistency list and lock set, at current discussion we assume the implicit case. We will discuss three key steps performed at compiler's backend:

1. **Consistency List Analysis:** This steps examines the atomic section and collects all shared data which might be read or written into the consistency list.
2. **Lock Refinement and Assignment:** This step assigns one or more locks to guard the entrance to the atomic section.
3. **Generation of Consistency Actions:** As indicated in the semantics, shared data will be refreshed into the thread private space before their first reference, and will be written back to the shared memory after their last update. Refresh and write_back operations are added into atomic section body in this step.

Among those three steps, lock refinement and assignment is our concentration in this paper, and we will discuss it in detail in section 3.3. Before that, we will introduce the implementation infrastructure in section 3.1. The other two steps, consistency list analysis and generation of consistency actions will be introduced, but not discussed in detail, in section 3.2 and 3.4, respective.

3.1 Infrastructure

We choose Omni OpenMP compiler [2] as our implementation platform. Its framework is shown as Figure 4. Omni is a source-to-source compiler, which translates the OpenMP (C/C++ or Fortran) program into C program with Omni runtime library function calls. A general C compiler then compile it into parallel executable. All those three steps (consistency list analysis, lock assignment and consistency actions generation) are implemented in Omni compiler's backend. The Omni runtime systems is composed of three parts. The runtime library API provides functions for each OpenMP constructs and directives to the general C compiler. The execution framework part executes the parallel executable generated by the compiler in a fork-join model in the target platforms, with the help of scheduling and resource management part.

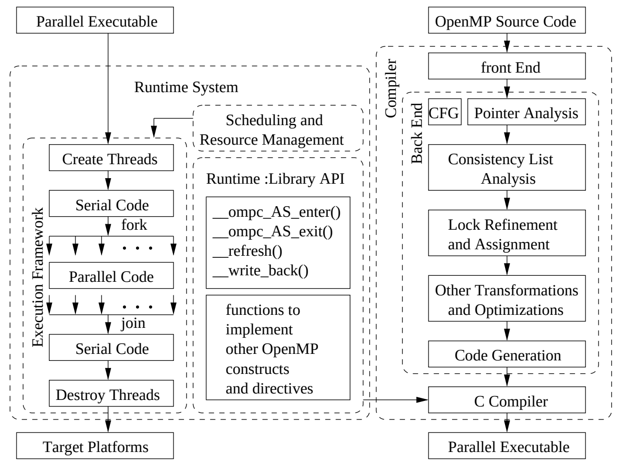

Figure 4: Omni OpenMP Compiler and Runtime System Infrastructure

3.2 Analysis

The consistency list of an atomic section contains shared data which will be kept consistent with the main memory at the entry and exit. As stated in section 2.2, in order to avoid un-necessary consistency actions and fault interference, the consistency list contains a subset of the declared shared data which are referenced (read or written) within the atomic section. The purpose of this step is to analyze the atomic section and collect those referenced shared data into the consistency list.

Besides directly referenced shared data, we have to recognize shared data indirectly referenced by a pointer. For instance, for the atomic section shown in Figure 5, the implementation is required to recognize not only the directly referenced pointer p, but also the indirectly referenced (through p) shared data x.

```
int main(){
  int *p, x;
  ......
  p = &x;
  #pragma omp parallel
  {
    #pragma omp atomic_sec
      *p = 1;
  }
  return 0;
}
```

Figure 5: Example of Necessity of Pointer Analysis

Static pointer analysis is applied to accomplish this task, and we chose Rugina and Rinard's data flow method [3] as the basic algorithm. We extended it to work on OpenMP programs by changing the control flow graph (CFG) for OpenMP working sharing constructs (**for**, **sections**, and **single**), and modifying data flow equations for different OpenMP data-sharing attributes (**private**, **firstprivate**, **lastprivate**, **shared**, etc.).

3.3 Lock Assignment

This phase assigns one or more locks to guard the entrance of the atomic section, so that any pair of concurrent atomic sections that might access the same shared data will be guarded by a same lock. 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 atomic section. Also, while some "over-locking" may be permitted for convenience and simplicity, the goal of the lock assignment phase is to satisfy the semantics 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.

The implementation of lock assignment at compiler side involves two steps:

1. Concurrency analysis, *i.e.*, statically identify pairs of atomic sections which might occur concurrently;

2. Build up the atomic section "Interference Graph", and assign locks to each atomic section;

The lock acquisition, release, and deadlock avoidance are handled in runtime library.

3.3.1 *Concurrency Analysis*

Consistency analysis addresses the problem of deciding whether two atomic sections in a concurrent program can be executed in parallel, suppose they are not synchronized. Since the problem of precisely determining whether two atomic sections can be executed in parallel is undecidable, we try to make a "conservative" estimation in the sense that if there is one real execution in which two atomic sections a_1 and a_2 from different thread of control happen in parallel, then this concurrency relationship a_1, a_2 must be included in our estimation.

Concurrency analysis, or its complement problem - non-concurrency analysis, is a widely studied research topic, and a variety of approaches have been proposed, each working on some different programming model. Callahan and Subhlok [4] propose a data flow algorithm, based on a synchronized control flow graph, that computes for each statement s a set of statements that must be executed before s. Their analysis is mainly focused on event variable synchronization, *i.e.*, **post** and **wait**. Masticola and Ryder [5] present a framework for non-concurrency analysis of Ada tasks. A sync graph is used to represent the program's synchronization behavior. It first assumes any pair of statements are concurrent, then improves this estimation by an iterative refinement process until a fixed point is reached. Jeremiassen and Eggers [6] propose a concurrency analysis method for course-grained, explicitly parallel program with **barrier** synchronization. The programs conform to an SPMD model of parallel programming. Its basic idea is to divide the program into a set of phases, and compute the control flow between them. Each phase consists of one or more sequences of statements that are delimited by barrier and can execute concurrently. The barrier is "global" in the sense that all the threads in the program must reach the barrier before any of them can proceed. We will discuss later in this section that

Jeremiassen and Eggers' method is not optimal, and it may generate some un-necessary concurrency relationships.

Concurrency Analysis in our implementation is to identify concurrent atomic sections, instead of concurrent statements. In this sense our analysis is relatively "coarse-grained". In OpenMP, there are two main factors that determine whether two atomic sections are concurrent or not. One is the control flow between them, and the other is the **barrier** synchronization. When the control flow of a thread reaches a barrier, it must wait until all of the other threads in the team have reach the same point. Thus any pair of atomic sections which are delimited within the same pair of barriers are concurrent.

In OpenMP, a barrier is either explicitly declared by the **barrier** construct, or optionally implied at the exit from the parallel region (specified by **parallel** construct) and various work sharing regions (specified by **for** construct, **sections** construct, and **single** construct [1]).

Besides control flow and barrier synchronization, some elaborations are required when we analyze concurrency under OpenMP work sharing constructs. Intuitively, in a general parallel region without work sharing partition and a parallel region with loop work sharing partition (specified by **for** construct), all threads in the team execute the identical code, but need not take the same path through the program. In this case, any pair of atomic sections delimited by the same barriers are concurrent. However, in **sections** construct, the whole section is executed by only one thread, thus all atomic sections in such section is not concurrent with each other (but they might be concurrent with atomic sections from another section). Same thing for **single** construct.

To combine the consideration on control flow, barrier synchronization and work sharing together, we introduce the notion of a *barrier synchronization and work-sharing aware control flow graph* (BW-CFG) $G = (V, E)$ for an OpenMP program. It is defined as follows:

1. A vertex $v \in V$ could be:
   (a) An ordinary basic block without OpenMP directive;
   (b) A single **barrier** construct, either explicitly declared or implied;
   (c) An atomic section, declared by atomic section directive and the subsequent structured block;
   (d) A par-begin block, a par-end block, a section-begin block or a section-end block;
   (e) Atomic section set block, containing a set of atomic sections in one work section of a **sections** construct, or in a **single** construct;
2. A parallel region begins from a par-begin block, and ends to a par-end block;
3. There exists a directed edge $(u, v) \in E$ if there is a control flow path from u to v;
4. The **sections** construct is transformed into a switch-case structure. It begins from a section-begin block and ends to a section-end block. Each path is an atomic section set block;

As an example, Figure 7 demonstrates the BW-CFG for the OpenMP program shown in Figure 6. Specifically, "barrier 1" is explicitly declared within the for loop, while "barrier 2" and "barrier 3" are **for** construct and **parallel** construct implied, respectively. Note that there is no implicit barrier for **sections** construct because of the **nowait** clause. The "ASset1" in Figure 7 contains AS_3 and AS_4 in the first section.

```
main()
{
   int i, x, array[10]={0}, ID;
   x = 0;
   #pragma omp parallel private(ID)
   {
      ID = omp_get_thread_num();
      #pragma omp for
      for(i=0; i<10; i++)
      {
         #pragma omp barrier
         if(i<5)
         #pragma omp atomic_sec /* AS1 */
            x += array[i];
         else
         #pragma omp atomic_sec /* AS2 */
            x -= array[i];
      }
      #pragma omp sections nowait
      #pragma omp section
      {
         if(ID==0)
         #pragma omp atomic_sec /* AS3 */
            array[0] ++;
         else
         #pragma omp atomic_sec /* AS4 */
            array[0] --;
      }
      #pragma omp section
      {
         #pragma omp atomic_sec /* AS5 */
            array[0] = x;
      }
   }
}
```

Figure 6:

Based on BW-CFG, our algorithm for concurrency analysis is very simple and straightforward. It is summaried as the following theorem. In this paper we assert its correctness, and leave the proof to the future work.

Theorem I: Two atomic sections AS_1 and AS_2 are concurrent if any of the following conditions holds:

1. AS_1 and AS_2 are connected, and there is a barrier-free control flow path between them;
2. (a) There is no control flow path between AS_1 and AS_2, and
   (b) Denote the "immediate" common pre-dominator of AS_1 and AS_2 as $preD$, and their "immediate" common post-dominator as $postD$, then there is a barrier-free path between $preD$ and $postD$ through AS_1, and there is a barrier-free path between $preD$ and $postD$ through AS_2;
3. AS_1 is in atomic section set $ASset_1$, AS_2 is in atomic

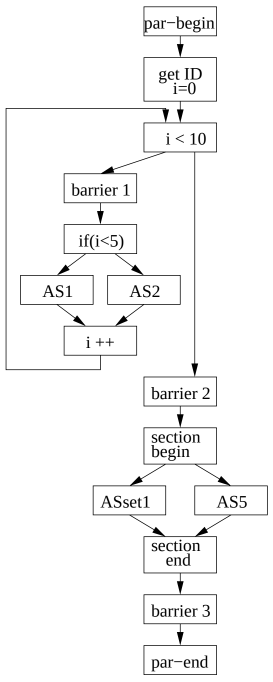

Figure 7:

section set $ASset_2$, and $ASset_1$ and $ASset_2$ are concurrent according to the preceding two conditions;

Soundness of **Theorem I** can be illustrated by some examples shown in Figure 8. Figure 8(a) is the simplest case. AS_1 and AS_3 are concurrent since the control flow path between them is barrier free (condition 1); same thing for AS_2 and AS_3. Besides, AS_1 and AS_2 are also concurrent since different threads may execute on different branches concurrently (condition 2). There is some subtlety in Figure 8(b), where barrier occurs in one of the if-then-else branches. Note that in any valid execution, threads either all pass the path $AS_1 \rightarrow AS_2 \rightarrow AS_4$, or all pass the path $AS_1 \rightarrow AS_3 \rightarrow barrier \rightarrow AS_4$. Hence AS_2 and AS_3 are not concurrent, neither are AS_3 and AS_4. Figure 8(c) shows a natural loop case, in which AS_1, AS_2 and barrier are in the loop body. AS_1 and AS_3 are not concurrent, while AS_2 and AS_3 are concurrent (condition 2).

Our concurrency analysis algorithm generates more optimal results than Jeremiassen and Eggers' method since they didn't consider the fact that each section will be executed sequentially by only one thread, thus atomic sections within that section cannot be concurrent with each other. In the example shown in Figure 6, their method will wrongfully find AS_3 and AS_4 are concurrent, while our algorithm will not.

3.3.2 Lock Assignment

As mentioned before, **critical** construct in OpenMP reduces the parallelism by serializing all critical section instances through a global lock. As a contrast, atomic section tries to maximize the parallelism by refining the locks. The intuition is, two atomic section instances must be executed in a total order only if they might access same shared data concurrently. If they are non-concurrent, or they will never touch same shared data even if they are concurrent, they

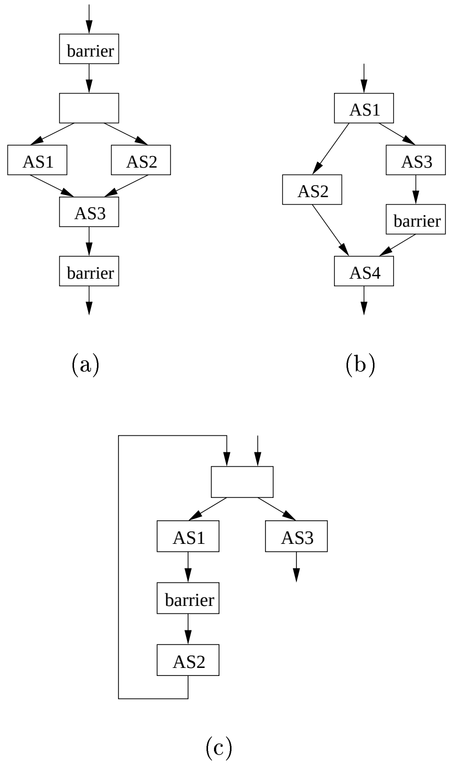

Figure 8: Examples of Concurrency Analysis Algorithm

are not required to execute one by one. Our lock assignment algorithm attempts to maximize the parallelism by elaboratively differentiating these cases. It is based on the consistency list and concurrency information of atomic sections, summarized by the *interference graph* $G_I = (V_I, E_I)$ of a parallel region, which is defined as follow:

1. Each vertex $v \in V_I$ corresponds to an atomic section in this parallel region, and it is associated with a consistency list CL_v;
2. There exists an un-directed edge $(u, v) \in E_I$, if atomic sections u and v are concurrent;

Based on this interference graph, the sufficient and necessary condition for two atomic sections u and v sharing same lock can be formulated as:

$$((u, v) \in E_I) \wedge (CL_u \cap CL_v \neq \phi)$$

In this case we call u and v are *interfered.*

The lock assignment problem can be formulated as follow:

Problem Formulation: Given an interference graph $G_I = (V_I, E_I)$, assign a lock set L_v to every node $v \in V_I$, so that:

1. If $u \in V_I$ and $v \in V_I$ are interfered, $L_u \cap L_v \neq \phi$;
2. If $u \in V_I$ and $v \in V_I$ are not interfered, $L_u \cap L_v = \phi$;
3. Let $L_G = \bigcup_{v \in V_I} L_v$, $|L_G|$ is minimized, where $|S|$ represents the cardinality of set S;

Solutions which satisfy only condition 1 are *correct* solutions, and solutions which satisfy all conditions are *optimal*.

The heuristic solution we use in our current implementation, is outlined in Figure 9. The basic idea is to find the *seeds* from atomic sections' consistency list which might cause data race, and then assign locks to guard the seeds. The final lock set of an atomic section is the union set of seed locks in it.

A *seed* is a set of shared data which either happen together, or never happen in any atomic section's consistency list, and are always shared by two atomic sections together. In other words, if two data d_1 and d_2 are in different atomic sections, or if d_1 is shared by atomic sections AS_1 and AS_2, while d_2 is shared by another two atomic sections AS_3 and AS_4, then d_1 and d_2 must in two different seeds.

A *seedset* is the set of seeds in G_I. Initially the seedset is empty. The seed is obtained by computing the intersection of consistency lists of two concurrent atomic sections, and then using this intersection to refine the existing seeds.

After seeding, each seed is assigned a unique lock, and all shared data in such seed share this lock. $datalock[s]$ is the vector which contains the lock assigned to every single data. The output of the algorithm, $aslock[i]$, which is a vector containing each atomic section's lock set, is obtained by collecting seed locks in atomic section i.

Figure 10 shows an example interference graph. Three cases are concerned:

case 1: Two atomic sections are concurrent, and they share data, as AS_1 and AS_2;

case 2: Two atomic sections are concurrent, but they don't share data, as AS_1 and AS_3;

case 3: Two atomic sections are non-concurrent, while they share data, as AS_3 and AS_4;

Table 1 illustrates how the algorithm works on Figure 10, and the result are shown in Table 2. For case 1, two atomic sections share some locks to guarantee the mutual exclusion; while in case 2, they don't share any lock, thus can be executed simultaneously. In case 3, they don't share any lock either.

Initially seedset = ϕ

Edge	Intersection	seedset
(AS_1, AS_2)	{x, y}	{x, y}
(AS_2, AS_3)	{z}	{x, y}, {z}
(AS_1, AS_4)	{y}	{x}, {y}, {z}

Final seedset = {{x}, {y}, {z}}

datalock[x] = 0, datalock[y] = 1, datalock[z] = 2

Table 1: Lock Assignment Process for Figure 10

Atomic Section	Seeds	locks
AS_1	{x}, {y}	{0, 1}
AS_2	{x}, {y}, {z}	{0, 1, 2}
AS_3	{z}	{2}
AS_4	{y}	{1}

Table 2: Lock Assignment Result for Figure 10

Algorithm: Lock assignment

```
Input: Interference graph $G_I = (V_I, E_I)$
Output: Lock set assigned to each $v \in V_I$

/* seeding */
seedset = $\phi$
for $(u, v) \in E_I$ do
  intersect = $CL_u \cap CL_v$
  if(intersect not $\in$ seedset) then
    found = false
    for seed $\in$ seedset do
      if intersect $\cap$ seed $\neq \phi$ then
        newseed1 = intersect $\cap$ seed
        newseed2 = seed - newseed1
        intersect = intersect - newseed1
        seedset = $seed^C \cup newseed1 \cup newseed2$
        found = true
      endif
    endfor
    if found = false then
      seedset = seedset $\cup$ intersect
    endif
  endif
endfor
/* assign lock to each data in a seed */
lock = 0
for $seed \in seedset$ do
  for $s \in seed$ do
  datalock[s] = lock
  endfor
  lock = lock + 1
endfor
/* assign locks to each atomic section */
for $v \in V_I$ do
  for $cl \in CL_v$ do
    aslock[v] = aslock[v] $\cup$ datalock[cl]
  endfor
endfor
for $v \in V_I$ do
  if aslock[v] = $\phi$ do
    aslock[v] = 0
  endif
endfor
```

Figure 9: Lock Assignment Algorithm

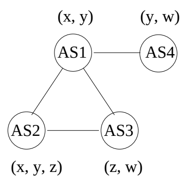

Figure 10: Lock Assignment Example

3.3.3 *Deadlock Avoidance*

Lock acquisition and release are implemented in Omni runtime system. Since an atomic section is assignmed a set of refined locks, instead of a single lock, deadlock may happen at execution time when each of two threads holds some locks and tries to claim those held by the other. We make two protocols, in both compiler and runtime system, to avoid such deadlock:

1. At compiler side, assign lowered numbered locks to shared data which is referenced by an earlier executed atomic section. This, in some extent , orders the lock acquisition, and reduces the possibility of deadlock.

2. At runtime system side, maintain a global lock *gl* to synchronize all attempts to claim locks. Only when a thread gets *gl*, can it claim an available lock. More important, once a thread holds *gl*, it does not release it unless it has got ALL locks it is looking for. Otherwise, it blocks itself, waiting for all it wants to become available.

3.4 Consistency Actions Generation

The purpose of this step is to refresh each shared data before its use, and write it back to main memory after its update. Our current implementation strictly follows the semantics given in section 2.2: refresh all data in the consistency list at the beginning, and a write them back together at the end.

Three functions in runtime system accomplish this work: refresh(), write_back(), and update_memory(). The scratchpad space is simulated by a dynamic allocated linked list, with each shared data as an element in such list. The function refresh() copies data from main memory into the list and marks it as clean. The function write_back(), which is also called at the beginning of atomic section, copies data and marks it as dirty. Finally, upon exiting the atomic section, function update_memory() copies all dirty data from the list back to the memory.

4. CASE STUDY

Ammp [7] is a molecular mechanics, dynamics and modeling program found in SPEC OMPM2001 benchmark suite. The OpenMP version runs a molecular dynamics on a protein-inhibitor complex which is embedded in water. The energy is approximated by a classical potential or "force field". Computation is dominated by the loop nest, which accounts for more than 96% of execution time of the entire program, as demonstrated in [8]. The loop nest computes the non-bounded forces that simulated atoms exert on on another according to the following algorithm [9]:

```
foreach atom A
   foreach node N
      if N contains A or is one of 26 nestest neighbors
         foreach atom B in node N
            compute forces between A and B
      else
         compute forces between A and node N
```

The main data structure in ammp is shown in Figure 11. Atoms are stored as a linked list. Each atom contains a "close" array field, with each element pointing to its neighbors. "atomall" is an array collecting all atoms' entry pointers for purpose of easier accesses.

The OpenMP implementation parallelizes the outer loop, using a **for** construct, as shown in Figure 12. Note that three

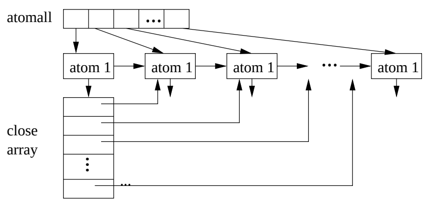

Figure 11: Atom Data Structure in ammp

lock functions in the original code have been substituted by implicit atomic sections.

The pointer analysis algorithm mentioned in section 3.2 are applied to the entire program, and the resulting consistency lists are summaried in Table 3. The analysis is conservative in such a way that the entire shared "atomall" array is put into each atomic section's consistency list.

```
#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];     . . .
AS1:  #pragma omp atomic_sec
      { ...
        a1->fx = ... ;
        a1->fy = ... ;
        a1->fz = ... ;
        ...
      }
      ...
for1: for( jj=0; jj<NCLOSE; jj++)
        { if( a1->close[jj] == NULL) break; }

      for( ii=0; ii< jj; ii++)
        {
          a2 = a1->close[ii] ;
AS2:      #pragma omp atomic_sec
          { ...
            ux = a1->dx;
            ...
            a2->fx -= ux*k ;
            a2->fy -= uy*k ;
            a2->fz -= uz*k ;
          }
        }
AS3:  #pragma omp atomic_sec
      {
        a1->fx += a1fx ;
        a1->fy += a1fy ;
        a1->fz += a1fz ;
      }
    } /* for */
}
```

Figure 12: Loop Nest with Atomic Sections in ammp

The BW-CFG graph for the parallel region is demonstrated in Figure 13. According to the concurrency analysis theorem in section 3.3, AS_1 and AS_2 are concurrent because there is a barrier-free control flow path between them. AS_1 is also concurrent with AS_3 due to the same reason. AS_2 and AS_3 are concurrent because AS_2 can reach AS_3 through the back edge of the nested loop without encountering any

Atomic Section	Consistency List
AS_1	atomall
AS_2	atomall
AS_3	atomall

Table 3: Consistency Lists for Atomic Sections in Figure 12

barrier.

Combining the consistency list and concurrency information together, Figure 14 demonstrates the interference graph. Since AS_1, AS_2 and AS_3 might access same data simultaneously, a global lock is assigned to guarantee the mutual exclusion among them.

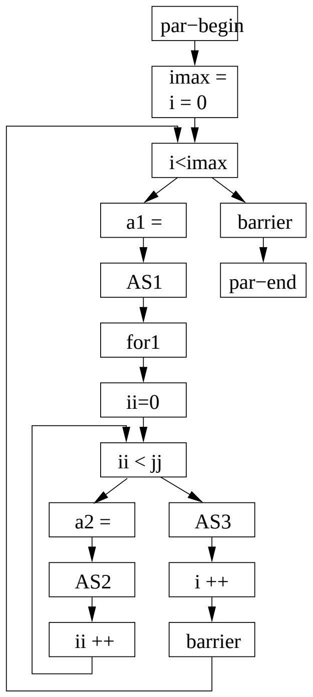

Figure 13: BW-CFG of ammp Parallel Region

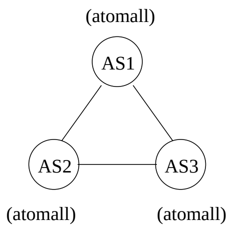

Figure 14: Interference Graph of ammp Parallel Region

The compiler generated C code is listed in Figure 15. The whole parallel region is translated into a function call, with parameters as shared data accessed in such parallel region. At the beginning of each atomic section, a runtime library function "__ompc_enter_AS()" is called, in which locks are acquired, as discussed in section 3.3. In the body of an atomic section, each shared data is refreshed, by calling runtime library function "refresh()", before its read, and is written back before its write, by calling function "write_back". Those two functions simulate the operations on scratch-pad memory shown in Figure 3. At the end of an atomic section, function "__ompc_exit_AS()" is called, in which updated shared data, by calling "write_back(), will be written back to the shared memory, and locks are released.

```
static void __ompc_func_3 (void **__ompc_args)
{
  struct atoms **_pp_atomall;
  struct atoms * __addr_0;
  struct atoms * __addr_1;
  struct atoms * __addr_2;
  int __Lock1[1];
  int __Lock2[1];
  int __Lock3[1];

  _pp_atomall = (struct atoms **)(*__ompc_args);

  imax = a_number;
  _ompc_static_bsched(0, imax, 1);
  for(i= 0; i< imax; i++) {  . . .
      a1 = (*atomall)[i];    . . .
AS1:  __Lock1[0] = 0;
      __ompc_enter_AS(1, __Lock1, 1);
      { ...
        __addr_0 = (struct atom *)write_back(a1, 1032);
        ...
        __addr_0->fx = ... ;
        __addr_0->fy = ... ;
        __addr_0->fz = ... ;
        ...
      }
      __ompc_exit_AS(1, __Lock1, 1);
      ...
      for( jj=0; jj<NCLOSE; jj++)
        { if( a1->close[jj] == NULL) break; }

      for( ii=0; ii< jj; ii++)
        {
          a2 = a1->close[ii] ;
AS2:      __Lock2[0] = 0;
          __ompc_enter_AS(1, __Lock2, 1);
          { ...
            __addr_1 = (struct atom *)write_back(a2, 1032);
            __addr_2 = (struct atom *)refresh(a1, 1032);
            ...
            ux = __addr_2->dx;
            ...
            __addr_1->fx -= ux*k ;
            _addr_1->fy -= uy*k ;
            __addr_1->fz -= uz*k ;
          }
          __ompc_exit_AS(1, __Lock2, 1);
        }
AS3:  __Lock3[0] = 0;
      __ompc_enter_AS(1, __Lock1, 1);
      { ...
        __addr_3 = (struct atom *)write_back(a2, 1032);
        ...
        __addr_3->fx += a1fx ;
        __addr_3->fy += a1fy ;
        __addr_3->fz += a1fz ;
      }
      __ompc_exit_AS(1, __Lock3, 1);
    } /* for */
}
```

Figure 15: Output C file with Runtime Library Function Calls from Compiler

Compile the C code shown in Figure 15 with a general C compile, such as cc or gcc, we get the parallel binary file. Running this binary file at Sun UntraSparc 4 CPUs machine, we obtained the same results as the original OpenMP program with lock functions. It shows the soundness of our analysis and translation.

5. CONCLUSIONS

In this paper we proposed a new synchronization construct - atomic section. An atomic section is a set of operations that is intended to be executed atomically, and mutually exclusively with the conflicting atomic section instances. Atomic section is designed to overcome the disadvantages of the current fine-grained synchronization mechanism - critical section and lock function, and provide both high performance and high productivity to parallel programmer.

We proposed a realization of atomic section in OpenMP, declared by **atomic_sec** construct. We presented its syntax, semantics, and a concept-proof implementation strategy. Three key problems in implementation are discussed: consistency list analysis, lock refinement and assignment, and consistency action generation. Among them, lock refinement and assignment is our main concern in this paper, and we discussed two main phases in detail: concurrency analysis and lock assignment.

A case study about ammp - a molecular dynamics program from SPEC OMP2001 benchmark suite - shows the soundness of our design and implementation.

6. REFERENCES

[1] OpenMP C/C++ Manual. http://www.openmp.org/specs/.

[2] Omni OpenMP Compiler Project. http://phase.hpcc.jp/Omni/home.html.

[3] Radu Rugina and Martin C. Rinard. Pointer analysis for structured parallel programs. *ACM Trans. Program. Lang. Syst.*, 25(1):70–116, 2003.

[4] David Callahan and Jaspal Sublok. Static analysis of low-level synchronization. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 100–111. ACM Press, 1988.

[5] Stephen P. Masticola and Barbara G. Ryder. Non-concurrency analysis. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 129–138, San Diego, California, May 1993.

[6] Tor E. Jeremiassen and Susan J. Eggers. Static analysis of barrier synchronization in explicitly parallel systems. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '94*, pages 171–180, Montréal, Québec, August 1994. North-Holland Publishing Company.

[7] Ammp Home Page. http://www.cs.gsu.edu/ cscrwh/ammp/ammp.html.

[8] Vishal Aslot and Rudolf Eigenmann. Performance characteristics of the spec omp2001 benchmarks. *SIGARCH Comput. Archit. News*, 29(5):31–40, 2001.

[9] Collin McCurdy and Charles Fischer. User-controllable coherence for high performance shared memory multiprocessors. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2003. ACM Press.