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

Optimized Lock Assignment and Allocation: A Method for Exploiting Concurrency among Critical Sections

Yuan Zhang
Vugranam C. Sreedhar†
Weirong Zhu
Vivek Sarkar†
Guang R. Gao

CAPSL Technical Memo Revised 65
March, 2007

†IBM T.J.Waston Research Center, Hawthorne, NY 10532. Email:
{vugranam,vsarkar}@us.ibm.com

Abstract

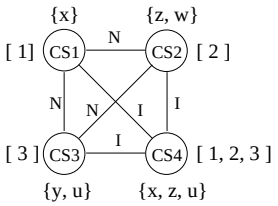
One of the major performance and productivity issues in parallel programming arises from the use of lock/unlock operations or atomic/critical sections to enforce mutual exclusion. Not only are these constructs complicated to understand and debug, but they are also often an impediment to achieving scalable parallelism. In this paper, we propose to give the programmer the convenience of critical sections combined with the scalability of fine-grained locks by solving the Minimum Lock Assignment (MLA) problem, which finds the minimum number of locks needed to enforce mutual exclusion among interfering critical sections without any loss of concurrency. We show that the MLA problem is related with the general graph coloring problem and it is NP-hard. We have proposed a heuristic to solve the MLA problem, and tested the optimality of the heuristic with the Integer Linear Programming (ILP) solver. We have also tested the efficiency of the MLA solution using scientific applications, from which we get up to 30% performance gain with respect to unoptimized programs for a full benchmark, and 47% for a benchmark subcomputation. The way we formulate and solve the MLA problem can be easily extended for other optimization problems.

Contents

List of Figures

```
main(){
   int u, w, x, y, z;
#pragma omp parallel sections
{
#pragma omp section
#pragma omp critical //CS1
{
   x ++;
}

#pragma omp section
#pragma omp critical //CS3
{
   y ++;
   u ++;
}

#pragma omp section
#pragma omp critical //CS2
{
   z ++;
   w ++;
}

#pragma omp section
#pragma omp critical //CS4
{
   x ++;
   z ++;
   u ++;
}

}
}
```

(a)



(b)

Figure 1: (a) Example program. Two sections are listed horizontally to save vertical space (b) Concurrency graph and the Minimum Lock Assignment solution (within brackets)

1 Introduction

Given that the processors in current and future computer system are becoming multi-core by default, it is important to address the performance and productivity issues in parallel programming. One of the major performance and productivity issues in parallel programming arises from the use of lock/unlock operations or critical sections to enforce mutual exclusion. 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. When programmers manage multiple fine-grained locks explicitly, they run the risk of introducing data races or creating deadlocks. When they use coarse-grained locks or critical sections, they run the risk of losing scalability in parallel performance. Ideally, we would like to give the programmers the best of both worlds – the convenience of coarse-grained locks or critical sections combined with the scalability of fine-grained locks.

In this paper, we propose to achieve this ideal by (1) letting programmers focus on the correctness of the application by using coarse-grained critical sections for mutual exclusion, and (2) letting the compiler maximize the concurrency among critical sections by selecting an assignment of compiler-managed fine-grained locks to critical sections. For (2), we introduce and solve the Minimum Lock Assignment (MLA) problem which finds the minimum number of locks needed to enforce mutual exclusion among interfering critical sections without any loss of concurrency. We will present our approach in the context of shared memory programming models, such as OpenMP and POSIX threads (Pthread).

Consider the simple OpenMP program shown in Figure 1(a). The main program begins as a single thread. When the *parallel sections* construct is encountered, a team of threads are generated, each (including the initial thread) executing one section. At the end of the paral-

lel sections, they synchronize and terminate, leaving only the initial thread to proceed. Four unnamed critical sections are used in this example program. The default OpenMP implementation uses a single global lock to control all unnamed critical sections, thereby introducing unnecessary serialization. For example, CS_1 and CS_2 do not access any common data, and need not be protected by a common lock. On the other hand, CS_2 and CS_4, which access the same shared data z, should be executed in mutual exclusion. In order to exploit concurrency among unnamed critical sections when it's legal to do so, we would like to partition critical sections into sets of "lock" classes such that all critical sections in a lock class are controlled by the same lock and hence run in mutual exclusion. [1] Each critical section can belong to one or more lock classes. A lock class can be implemented in a portable way by having the compiler generate the code that uses named critical constructs or explicit lock management, or in a platform-specific way by generating special hardware instructions for lock management.

Using the approach described in this paper we assign three locks to control critical sections in Figure 1(a). The results are shown in a *labeled concurrency graph* in Figure 1(b). Each vertex in this graph corresponds to an unnamed critical section in the original program. An edge (u, v) in the graph indicates that u and v are concurrent, i.e., there exists an execution of the program in which two threads may attempt to enter an instance of u and an instance of v at the same time. If an edge is labeled "N", it means that the critical sections are non-interfering and need not be executed in mutual exclusion. If an edge is labeled "I", it means that the critical sections are interfering and mutual exclusion needs to be enforced. The vertex label within brackets indicates the set of locks selected for each critical section by our approach. Three locks is the minimum number of locks needed to ensure that no parallelism is sacrificed.

Note that critical section CS_4 is controlled by multiple locks (1, 2 and 3) in Figure 1(b). The semantics of such a lock set is as follows: before a thread can execute a critical section it has to acquire all locks in the lock set, and once finished it has to release all of them [14]. In order to avoid deadlocks, the set of locks should be acquired in a predetermined order [14].

The correctness of our framework for lock assignment is based on the semantics of isolation and locking defined in the area of transaction processing and concurrency control [14]:

- **Rule 1**: *Two concurrent critical sections that access the same data are isolated if they are controlled by some common lock.*

- **Rule 2**: *Two concurrent critical sections that don't access the same data are isolated even if they are controlled by different locks.*

If all critical sections are controlled by a single global lock then at any time at most one thread can be in a critical section, which essentially guarantees the serialization among critical sections. Our lock assignment will preserve this isolation and serialization semantics, and yet exploit available parallelism among critical sections.

[1] This approach can also be used to partition atomic constructs and named critical constructs in OpenMP.

1.1 Problem Statement

Based on above two rules, we propose the **Minimum Lock Assignment** (MLA) problem to exploit the concurrency among critical sections:

Problem 1.1 (Minimum Lock Assignment). *Given a parallel program with a set of critical sections, find the minimum number of distinct locks that are needed for controlling the critical sections such that*

(a) Two critical sections are assigned disjoint sets of locks if (1) they are concurrent and (2) they do not access any common location, or if they access a common location then none of them write to the common location.

(b) Two critical sections are assigned at least one common lock if (1) they are concurrent and (2) they access some common data and at least one of them write to the common location.

Assume all shared memory locations that a critical section access can be statically identified by compiler analysis, then a simple solution to the MLA problem is to assign a distinct lock to each shared memory location, and the lock set of a critical section is the set of locks assigned to memory locations it accesses. However, this approach may use more locks than necessary. We say the number of locks required in this simple solution, i.e., the total number of memory locations accessed in a program, denoted as $|M|$, is the *upper bound* (UB) of the optimal solution.

1.2 Main Contributions

The main contributions of this paper are as follows:

- We propose and formulate the MLA problem to exploit the concurrency among critical sections in a parallel program. We show that the MLA problem is related with the classical graph coloring problem, and it is NP-complete. We then present a heuristic for solving the problem.
- We formulate the MLA problem into the Integer Linear Programming (ILP) problem, and evaluate our heuristic by comparing it with the optimal solution produced by the commercial ILP solver CPLEX. In 300 randomly generated testing cases, we observed that our MLA heuristic is optimal for 83.3% of them.
- We tested the performance of our heuristic using a 10-way Sun-Fire machine on a set of Splash2 [27] benchmarks, and obtained up to 30.17% performance speedup compared to the benchmarks controlled by single locks.
- We also implemented our heuristic approaches in the Omni [20] OpenMP compiler, and evaluated its performance on the Cyclops-64 [9] chip architecture - a state-of-the-art multicore, using the UA benchmark from NAS Parallel Benchmarks. We observed a 2× speedup compared to the default single lock implementation for critical sections in function `transfb_c_2`.

- We show that the way we formulate and solve the MLA problem can be easily extended to solve other synchronization optimization problems.

The rest of the paper is organized as follows. In Section 2 we introduce the basic data structure, called "concurrency graph", then discuss several important components of a concurrency graph and their properties. In Section 3 we present the MLA heuristic. We report the experimental results in Section 4. In Section 5 we show how to extend the concurrency graph and the MLA formulation to solve other synchronization optimization problems. Related work and conclusions are presented in Section 6 and Section 7, respectively.

2 Concurrency Graph and Critical Sections

The concurrency and interference relation among critical sections in a parallel program is modeled as a *concurrency graph*. In this section we demonstrate how this concurrency graph helps us solve the MLA problem and the K-LA problem.

2.1 Concurrency Graph

Definition 2.1. A **Concurrency Graph** is an undirected graph $G = (V, E)$, in which: a vertex $v \in V$ denotes a textual critical section, and there is an edge $(u, v) \in E$ if instances of critical sections u and v may be concurrent.

In the above definition, if two instances of the critical section u are concurrent, we do not introduce a self-loop on u, since we will assign at least one lock to each critical section, and the isolation semantics of u with respect to itself is self preserved. We use *concurrency analysis* to compute the concurrency relation and use data flow analysis with isolation semantics to determine the data set of a critical section [26]. Figure 1(b) illustrates the concurrency graph for Figure 1(a). The location set is also listed within curly braces.

Next we define the notion of interfering and non-interfering critical sections.

Definition 2.2. Two concurrent critical sections are said to be **non-interfering** if either they do not access a common location or if they access a common location then none of them write to the common location.

Definition 2.3. Two concurrent critical sections are **interfering** if they access some common location and at least one of them write to the common location.

We extend the concurrency graph defined in Definition 2.1 by labeling an edge (u, v) with label I when critical sections u and v are interfering, and with label N when u and v are non-interfering. Note that a general concurrency graph may be a forest of connected graphs, and we analyze each connected component independently. In the following discussion, we simply assume that a concurrency graph G is a connected graph.

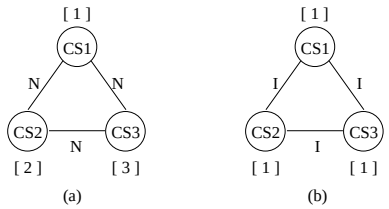


Figure 2: An example of (a) Non-interfering critical sections and (b) Interfering critical sections. In the figure, [ ] denotes the lock assignment.

2.2 Non-interfering Critical Sections

Consider a class of parallel programs P_n whose corresponding concurrency graph contains only non-interfering edges. Figure 2(a) illustrates an example of such *non-interfering concurrency graph*. We observe that all incident edges of a critical section are non-interfering, thus it cannot share any lock with its neighbors. This implies that whenever two critical sections are connected (concurrent), they require different locks. We can now restate the MLA problem (Problem 1.1) for non-interfering critical sections as follows:

Problem 2.1. Given a program with a set V_n of non-interfering critical sections, find the minimum number of locks that can be assigned to each critical section, such that if two different critical sections in V_n are concurrent then they get different locks.

The above problem is equivalent to the classical graph coloring problem — color the vertices (critical sections) of a graph using the minimum number of colors (locks) such that no two adjacent (concurrent) vertices (critical sections) are given the same color (lock). The MLA problem for this special class of programs is NP-complete[2]. The result of lock assignment for our example is shown in Figure 2(a).

2.3 Interfering Critical Sections

Consider a class of programs P_i, for which the concurrency graph contains only interfering edges. Figure 2(b) illustrates such an example. In this case, two critical sections are either concurrent and interfering, or are not concurrent (i.e., they are not connected). If they are concurrent and interfering, they should share at least one common lock to preserve the isolation semantics, which implies that they must be serialized. If they are not concurrent, they are already serialized. Therefore, in this interfering special case, there is no inherent parallelism, so we can use a single lock without introducing any performance penalty.

[2]For certain classes of graphs, such as the interval graphs, the graph coloring problem can be solved in polynomial time. However, the general concurrency graphs are not necessarily interval graphs.

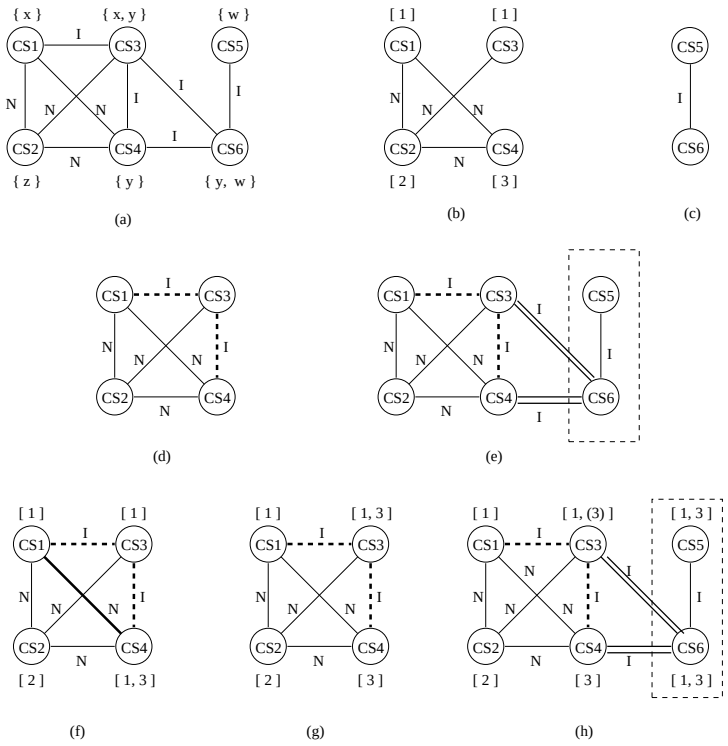


Figure 3: (a) A general concurrency graph (b) The non-interfering subgraph G_n (c) The interfering subgraph G_i (d) The SNIG G_n^s (e) The crossing edges (double lines), serializing interfering edges (dotted lines), and the interfering subgraph (in dotted box) (f) A un-safe borrowing from CS_3 to CS_4 (g) A safe borrowing from CS_4 to CS_3 (h) Final lock assignment result

2.4 Concurrency Graph Partition

Given a concurrency graph $G = (V, E)$, let E_n denote the set of non-interfering edges and E_i denote the set of interfering edges in G, such that $E = E_n \cup E_i$ and $E_n \cap E_i = \emptyset$. Let $G_n = (V_n, E_n)$ be the *non-interfering subgraph* induced by E_n, where $V_n \subseteq V$ such that a vertex $v_n \in V_n$ has at least one non-interfering edge incident on it. Figure 3(b) illustrates the G_n subgraph of Figure 3(a). Let $G_i = (V_i, E_i')$ be the *interfering subgraph* induced by vertices V_i, where $V_i = V - V_n$ and $E_i' \subseteq E_i$ is a set of interfering edges (x_i, y_i) such that $x_i, y_i \in V_i$. Figure 3(c) illustrates the interfering subgraph for Figure 3(a). Finally, let $E_i'' = E_i - E_i'$ be a set of interfering edges that are not in G_i. Some of interfering edges in E_i'' connect vertices of the non-interfering subgraph, for example, edges (CS_1, CS_3) and (CS_3, CS_4), as illustrated as bold dashed lines in Figure 3(c). We call such interfering edges that occur inside a non-interfering subgraph as *serializing* interfering edges E_s, because they could "serialize" inherent parallelism that exists within non-interfering subgraph. The remaining interfering edges $E_{ci} = E_i'' - E_s$ are *crossing edges* between vertices in G_n and G_i. In the example shown in Figure 3(a), $E_{ci} = \{(CS_3, CS_6), (CS_4, CS_6)\}$, illustrated as double solid lines in (e). Besides the non-interfering subgraph G_n and the interfering subgraph G_i, we introduce the notion of

the *serializing non-interference graphs* (SNIG) as the non-interfering subgraph with serializing edges, $G_n^s = (V_n, E_n \cup E_s)$. Figure 3(d) illustrates an example of SNIG. SNIGs have some interesting properties that will influence the lock assignment.

2.5 Serializing Non-Interference Graph

Let us consider a class of concurrency graphs called *Serializing Non-Interfering Graphs* (SNIGs). A SNIG is made of only non-interference edges and serializing interfering edges (as defined in the previous section). Serializing interfering edges constrain the inherent parallelism in a non-interfering concurrency graph. They also constrain the minimum number of locks for a SNIG. We can restate the MLA problem (Problem 1.1) for SNIGs as an extension of the graph coloring problem.

Problem 2.2. Given a SNIG $G_n^s = (V_n, E_n \cup E_s)$ find the minimum number of colors that are needed to color G_n^s such that

1. If two vertices x_n and y_n are connected by a non-interfering edge then x_n and y_n are given different colors.
2. If two vertices x_n and y_n are connected by a serializing interfering edge then x_n and y_n are given the same color(s).

The following observation states that sometimes it is impossible to color a SNIG if a vertex can be assigned with at most one color.

Observation 2.1. It is impossible to color an arbitrary SNIG with the following conflicting constraints:

1. Each vertex gets only one color,
2. If two vertices x_n and y_n are connected by a non-interfering edge then they are given two different colors, and
3. If two vertices x_n and y_n are connected by a serializing interfering edge then they are given the same color.

Consider the SNIG in Figure 4. Assume we satisfy all above constraints, then all critical sections get the same lock, because they are connected by serializing interfering edges (CS_1, CS_3), (CS_3, CS_4) and (CS_4, CS_2). However, the constraint (2) requires that CS_1 and CS_2 are given two different colors, a contradiction. Therefore Figure 4 cannot satisfy all three constraints.

There are two ways to deal with the above impossibility: (i) relax constraint (1) in the above observation, or (ii) relax constraint (2). By relaxing constraint (1) we are allowed to assign multiple colors to each vertex. By relaxing constraint (2) we will reduce the parallelism. Constraint (3) must be satisfied since otherwise the isolation semantics will be violated. In the

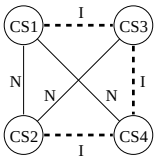


Figure 4: Example SNIG for Observation 2.1

rest of the paper we will take the approach of assigning multiple locks so as to maximize the parallelism. Let $C(x)$ be the set of colors that are assigned to a vertex x, then Problem 2.2 is restated as:

Problem 2.3. Given a SNIG $G_n^s = (V_n, E_n \cup E_s)$ find the minimum number of colors that are needed to color G_n^s such that:

1. If two vertices u and v are connected by a non-interfering edge then $C(u) \cap C(v) = \emptyset$ and
2. If two vertices u and v are connected by a serializing edge then $C(u) \cap C(v) \neq \emptyset$.

Let G be an arbitrary concurrency graph, and let G_n^s be the SNIG of G. We will show in Section 3.1 that the minimum number of locks required by G is no more than the minimum number of locks required by G_n^s.

3 Minimum Lock Assignment Solution

The MLA problem for arbitrary concurrency graphs is NP-complete [3]. In this section we present a heuristic approach for solving MLA. We also formulate it as an integer linear programming (ILP) problem, and use this ILP formulation to quantitatively evaluate our heuristic.

3.1 MLA Heuristic

Our MLA heuristic consists of three main steps (see Figure 5):

1. Assign locks to non-interfering subgraph G_n using graph coloring heuristic (Line 6).
2. Ensure that the serializing interfering edges in SNIG are correctly handled (Line 7).
3. Finally propagate the locks to the interfering subgraph G_i (Line 8).

The first step is straightforward. We use a heuristic graph coloring algorithm [4] to color G_n, and the result for our example is shown in Figure 3(b).

Next, we must ensure that critical sections connected by serializing interfering edges in SNIG are correctly serialized. The details of this step is given by the function HandleSerializingEdges

[3]The MLA problem is NP-complete because one of its special cases – graph coloring is NP-complete.

```
LockAssignment(G)

1.      Initialize Lock(u) for all u ∈ V as empty
2.      Partition the graph G
3.      if G_n = φ
4.         assign a global lock to each critical section
5.      else
6.         HLB = GraphColoring(G_n)
7.         HandleSerializingEdges(G_n^s)
8.         LockPropagation(E_ci, G_i)
9.      end if
10.     if HLB > |M| then
11.        for each v ∈ V
12.           Lock(v) = ⋃_{i∈LS(v)} Lock(i)
13.        end for
14.     end if

HandleSerializingEdges (G_n^s)

15.     for each serializing interfering edges (u, v)
16.        if Lock(u) ∩ Lock(v) = ∅
17.           if borrow(u ← v) is safe
18.              Lock(u) = Lock(u) ∪ Lock(v)
19.           else if borrow(v ← u) is safe
20.              Lock(v) = Lock(v) ∪ Lock(u)
21.           else
22.              HLB = HLB + 1
23.              add a new lock to u and v's lock sets
24.           end if
25.        end if
26.     end for

LockPropagation(E_ci, G_i)

27.     for each (v_i, v_n) ∈ E_ci
28.        sequence = BreadthFirstSearch(G_i, v_n)
29.        for each v in sequence
30.           Lock(v) = ⋃_{p∈Pred(v)} Lock(p)
31.        end for
32.     end for
```

Figure 5: Lock Assignment Heuristic

in Figure 5. In Figure 3(d), CS_1, CS_3 and CS_4 are in G_n and each of them has obtained a lock from the graph coloring. Interfering critical sections CS_1 and CS_3 are automatically serialized by sharing lock 1, but CS_3 and CS_4 are not. A straightforward method to solve this is let one of them "borrow" the lock from the other. For a serializing interfering edge (u, v), we say vertex u borrows the lock from v, denoted as $borrow(u \leftarrow v)$, if u adds v's lock to its lock set, $Lock(u) = Lock(u) \cup Lock(v)$. Denote the set of locks from u's non-interfering neighbors as $NIN(u)$, $NIN(u) = \bigcup_{(u,w)\in G_n} Lock(w)$. Before the borrowing, u has a disjoint

set of locks with all its non-interfering neighbors, i.e., $Lock(u) \cap NIN(u) = \emptyset$. This implies that the parallelism between u and its non-interfering neighbors is maximized. After the borrowing, we also require u not share any lock with its non-interfering neighbors. This is satisfied if $Lock(v) \cap NIN(u) = \emptyset$, that is, none of u's non-interfering neighbors has u's borrowed lock from v. In this case we say the borrowing is "safe", which means it does not reduce parallelism among non-interfering critical sections.

In our example, in order to isolate CS_3 and CS_4, we first let CS_4 borrow the lock from CS_3, then $Lock(CS_4) = \{1, 3\}$. This is shown in Figure 3(f). However, this borrowing is not safe, because one of CS_4's non-interfering neighbor CS_1 would share lock 1 with it. Then we try the alternative way. We let CS_3 borrow the lock from CS_4. This is illustrated in (g). This borrowing is safe because $Lock(CS_4) \cap NIU(CS_3) = \emptyset$, where $NIU(CS_3) = \{2\}$. Note that if neither borrowing is safe, we will introduce a new lock and add it to both end vertices' lock sets. The procedure of lock borrowing is summarized in Figure 5.

The first two steps together color the SNIG G_n^s. Finally, in function LockPropagation, we propagate the SNIG lock assignment result to color the interfering subgraphs G_i . The interfering subgraph G_i is connected to the non-interfering subgraph G_n through a set of crossing edges (v_n, v_i), where $v_n \in G_n$, and $v_i \in G_i$. Each (v_n, v_i) is an interfering edge, that means v_i should share v_n's locks obtained from the graph coloring. We say v_n "propagate" its locks (colors) to v_i. If v_i has more than one incident crossing edges, then it should inherit locks from all its neighbors in G_n, that is, $Lock(v_i) = \bigcup_{(v_i,v_n)\in E_{ci}} Lock(v_n)$. Subsequently, v_i propagates its lock set to its neighbors in G_i. This propagation continues until every vertex in G_i inherits locks from its neighbors. This procedure can be simply implemented as a set of breath first searches, with each v_n at a crossing edge as the source vertex. The algorithm is shown in Figure 5. The propagation result of our example is shown in Figure 3(h). An important property of this lock propagation is that it doesn't introduce any new lock, therefore the number of locks required to color G_i cannot exceed the number of locks required to color the SNIG G_n^s.

The final lock assignment result is shown in Figure 3(h). We refer to the number of locks required to color G as the Heuristic Lock Bound (HLB). We have mentioned in Section 1.1 that the upper bound UB of the required locks is the number of memory locations accessed in the concurrency graph G, i.e., $UB = |M| = \bigcup_{v\in V} LS(v)$, where $LS(v)$ denotes the set of data accesses in v. In some cases HLB might exceed UB (see Section 3.3, and we need to choose the smaller one from HLB and UB for lock assignment. The MLA heuristic algorithm is summarized in Figure 5.

The following two theorems show that the lock assignment heuristic is correct, and it can maximize the parallelism.

Theorem 3.1. *When the algorithm LockAssignment (G) terminates, any pair of interfering critical sections in G share at least one common lock.*

Proof: Let (u, v) be an interfering edge. There are three cases to consider:

Case 1: $(u, v) \in G_i$. Since one of u and v propagates its locks to the other, they share some locks.

Case 2: (u, v) is a crossing edge. Without loss of generality, assume $v \in G_n$, and $u \in G_i$. Then v propagates its locks to u.

Case 3: (u, v) is a serializing interfering edge. If a borrow, say $borrow(u \leftarrow v)$, is safe, u and v share v's lock. Otherwise, they share the new added lock.

Theorem 3.2. *When the heuristic LockAssignment (G) terminates, any pair of non-interfering critical sections do not share any lock.*

Proof: For the edge $(u, v) \in G_n$, the theorem is trivially true due to the graph coloring of G_n and the safe borrowing strategy.

The following lemma relates the minimum lock assignment for G and its SNIG G_n^s.

Lemma 3.1. *Let $MinLock(G)$ be the minimum number of locks that is needed for assigning a concurrency graph G. Then $MinLock(G) = MinLock(G_n^s)$.*

Proof: This is immediately comes from the fact that the lock propagation doesn't introduce any new lock.

The following theorem states that a lock assignment on G is optimal if and only if the lock assignment on G_n^s is optimal.

Theorem 3.3. *Lock assignment on a concurrency graph G is optimal if and only if the lock assignment on its SNIG G_n^s is optimal.*

Proof: 1: Optimal lock assignment on $G_n^s \rightarrow$ optimal lock assignment on G.

The proof proceeds by contradiction. From Lemma 3.1 we know $MinLock(G) = MinLock(G_n^s) = k$. Assume there is another lock assignment on G that needs k' locks, where $k' < k$, then coloring G_n^s also needs k'. Then we find an even more optimal solution for G_n^s coloring. Contradiction. Therefore an optimal graph coloring algorithm implies an optimal lock assignment result.

2: Optimal lock assignment on $G \rightarrow$ optimal lock assignment on G_n^s.

This proof proceeds by contradiction too. If an optimal lock assignment on G requires k locks, then G_n^s also needs k colors. If there is an even more optimal assignment on G_n^s which requires k' colors, where $k' < k$, then perform the lock propagation, and we will have an even better lock assignment for G. This contradicts the fact that the original lock assignment is optimal.

Therefore, Lock assignment on a concurrency graph G is optimal if and only if the lock assignment on its G_n^s is optimal.

The concurrency graph partitioning runs in $\mathcal{O}(V+E)$ time, where V and E are number of vertices and edges in the graph, respectively. The graph coloring's time complexity is $\mathcal{O}(V^2)$. At the worst case, HandleSerializingEdges takes $\mathcal{O}(E*V)$ time, and LockPropagation takes $\mathcal{O}(E^2+V*E)$ time. Therefore, at the worse case, the total time complexity of LockAssignment is $\mathcal{O}(E^2+V*E)$.

3.2 ILP Formulation

In this section, we formulate the MLA problem as an integer linear programming (ILP) problem. Given a concurrency graph $G=(V,E)$, we introduce 0-1 variables $f_{u,i}$ to indicate whether lock i is assigned to node u in G, $1 \le u \le |V|$, and $1 \le i \le |M|$, where M is the set of shared memory locations that are accessed in all critical sections. Recall that the number of locks given by an optimal solution cannot exceed $|M|$. Since each critical section must be assigned at least one lock, we have the following constraint:

$$f_{u,1} + f_{u,2} + \cdots + f_{u,|M|} \ge 1 \quad \text{for all } u \in G \tag{1}$$

We use 0-1 variables l_i to indicate whether lock i is assigned to any critical section, $l_i = f_{1,i} \vee f_{2,i} \vee \cdots \vee f_{|V|,i}$. This condition is represented by the following constraints:

$$f_{1,i} + \cdots + f_{|V|,i} \ge l_i \tag{2}$$

$$f_{1,i} + \cdots + f_{|V|,i} \le |V| \times l_i \tag{3}$$

Next we derive conditions that ensure the lock assignment is correct and maximizes the parallelism. Recall that a lock assignment solution is correct if interfering critical sections u and v ($(u,v) \in E$ and $LS(u) \cap LS(v) \neq \phi$) share some lock, and parallelism is maximized if non-interfering critical sections ($(u,v) \in E$ and $LS(u) \cap LS(v) = \phi$) are assigned two disjoint sets of locks. Let 0-1 variable $s_{u,v,i}$ indicate whether u and v share lock i, then $s_{u,v,i} = f_{u,i} \wedge f_{v,i}$. This condition is imposed by the following constraints:

$$f_{u,i} + f_{v,i} \ge 2 \times s_{u,v,i} \tag{4}$$

$$f_{u,i} + f_{v,i} \le 2 \times s_{u,v,i} + 1 \tag{5}$$

We use 0-1 variable $s_{u,v}$ to indicate whether u and v share any lock. Then $s_{u,v} = s_{u,v,1} \vee \cdots \vee s_{u,v,|M|}$. The following two constraints represent this condition:

$$s_{u,v,1} + \cdots + s_{u,v,|M|} \ge s_{u,v} \tag{6}$$

$$s_{u,v,1} + \cdots + s_{u,v,|M|} \le |M| \times s_{u,v} \tag{7}$$

Then

$$s_{u,v} = 1 \qquad \text{for interfering edge } (u,v) \tag{8}$$

$$s_{u,v} = 0 \qquad \text{for non-interfering edge } (u,v) \tag{9}$$

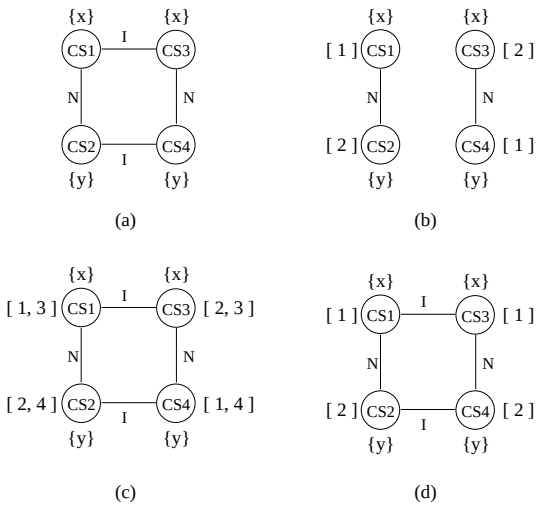


Figure 6: Example for in-optimality of MLA heuristic (a) Example concurrency graph (b) Non-interfering subgraph and its graph coloring result. (c) The tentative MLA solution after handling serializing edges (d) The optimal solution

The total number of locks used is:

$$N = l_1 + \cdots + l_{|M|} \tag{10}$$

Therefore, the ILP problem is to minimize N subject to inequalities (1) to (9).

3.3 Discussion

As Theorem 3.3 states, the lock assignment on a concurrency graph G is optimal if and only if the lock assignment on its SNIG G_n^s is optimal. In other words, the in-optimality of a lock assignment heuristic comes from assigning locks to G_n^s. In our case, such in-optimality comes from either the graph coloring heuristic, or handling the serializing edges. Since the former is obvious, we discuss the latter in this section.

Consider the example concurrency graph shown in Figure 6(a). Its non-interfering subgraph is composed of two strong connected components, as shown in Figure 6(b). Since each strong connected component is colored separately, we might get the graph coloring result as shown in this graph, and it is an optimal coloring solution. Figure 6(c) shows that we need another two locks to handle serializing edges, thus we assign two more locks than the optimal solution which is shown in Figure 6(d).

Interestingly, this example also shows the cases in which the number of locks used to assign a concurrency graph (i.e., HLB) according to our heuristic might exceed the total number of accessed memory locations (i.e., UB). By choosing the smaller one from HLB and UB, and performing some necessary adjustment (Line 10-13 in Figure 5), our heuristic can still give the optimal solution.

	Avg	Min	Max
# Vertices (V)	8.63	2	16
# Edges(E)	16.73	1	53
Edge Density E/V^2	0.19	0.09	0.28
# Non-interfering edges (E_n)	3.37	0	20
E_n/E	0.22	0	1
# Serializing interfering edges	2.85	0	27

Table 1: Features of random concurrency graphs

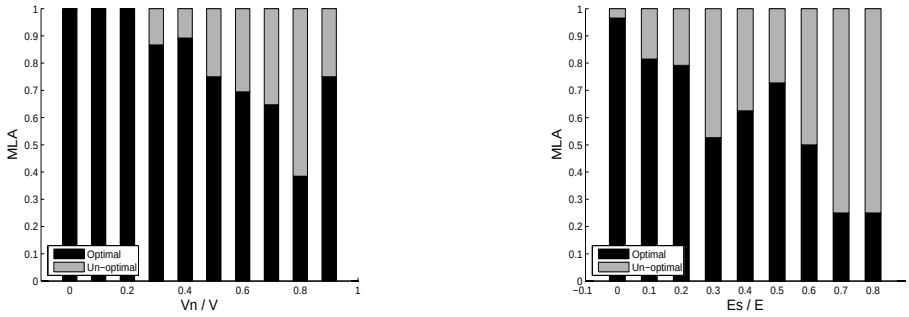
4 Experimental Results

In this section, we present three sets of experiments to evaluate our lock assignment algorithm. First we compare the precision of our MLA heuristic with the optimal solution based on the ILP formulation on a set of 300 random concurrency graphs. We also try to illustrate that the in-optimality of the MLA heuristic is mainly caused by coloring the SNIG. Second we evaluate the effectiveness of the MLA heuristic using Splash2 benchmarks. Finally, we evaluate the MLA heuristic using NPB benchmark UA on the Cyclops-64 simulator.

4.1 Optimality Evaluation

To study the precision of our MLA heuristic we implemented our ILP formulation in the commercial ILP solver CPLEX, and tested the heuristic and the ILP formulation on a set of 300 randomly generated concurrency graphs with characteristics shown in Table 1. We limited our random concurrency graphs to contain at most 16 nodes due to time constraints in the ILP solver. It shows that our heuristic solution is optimal for 83.3% of tested graphs. For the remaining 16.7% of graphs our heuristic assigns more locks than the optimal solutions, and in the worst case at most two extra locks than optimal solutions.

We also evaluated the influence of non-interfering subgraph G_n and serializing interfering edges E_s for lock assignment. For this purpose, we listed the optimality of the MLA heuristic with the increase of the relative size of non-interfering subgraph, given by V_n/V, and with the increase of the relative number of serializing interfering edges, given by E_s/E, in Figure 7(a) and (b), respectively. As an example, Figure 7(a) shows that our MLA heuristic gives optimal solutions to about 70% of test cases that have $V_n/V = 0.6$, and un-optimal solutions (i.e., assign extra locks) for the rest 30% of test cases that have $V_n/V = 0.6$. Figure 7(a) and (b) illustrate that the optimality of our heuristic depends on the non-interfering subgraph size and the relative number of serializing interfering edges. This result supports our discussion in Section 3.3 that the precision of the MLA heuristic is dependent on the precision of the graph coloring on the non-interfering subgraph and the handling of serializing edges.



(a) MLA Vn / V

(b) MLA Es / E

Figure 7: Optimality of the MLA heuristic

Application	Short description	Problem size	CSs	CS time (1 procs)	Analyzable CS time (%)	Non-analyzable CS time (%)	Locks assigned
Barnes	N-body	262144 bodies	6	6.29%	3.05e-3%	100%	3
Cholesky	Matrix factoring	tk29.O B8 C256	7	32.37%	6.07%	93.93%	4
Ocean-cont	Hydrodynamics	514×514	4	0.11%	100%	0	4
Radiosity	3-D rendering	largeroom batch	37	9.93%	59.79%	40.21%	18
Water-nsq	Water Molecules	512 molecules	9	11.54%	0.42%	99.58%	7

Table 2: Benchmarks and their features

4.2 Performance Study on Sun-Fire

Next we study the performance of the MLA heuristic using a set of Splash2 benchmarks listed in Table 2.[4] Splash2 benchmarks call the Pthread library, and mutual exclusions are enforced by `pthread_mutex_lock(<lock_var>)` and `pthread_mutex_unlock(<lock_var>)` functions

[4]http://www-flash.stanford.edu/apps/SPLASH

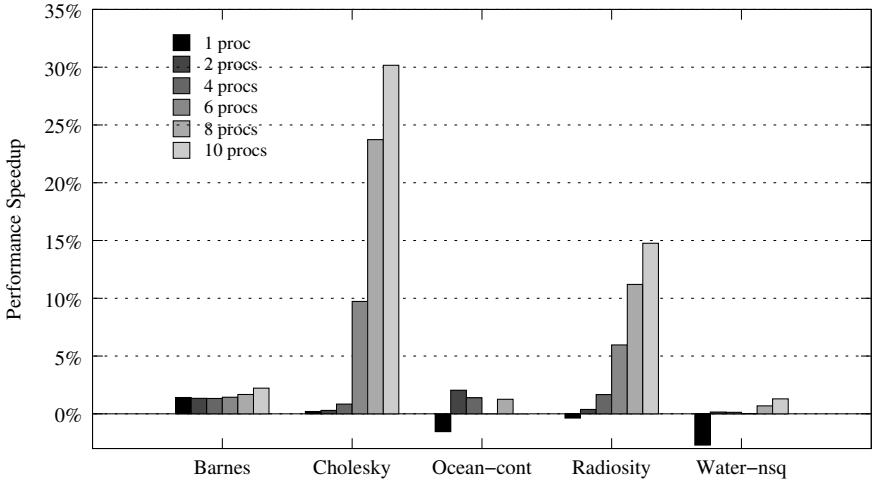


Figure 8: Performance speedup from lock assignment

with explicit lock variables. For the purpose of our performance study, we manually transformed each critical section to be controlled by a single lock. We constructed the concurrency graph for each benchmark manually, and applied the MLA algorithm to calculate the lock assignment. The number of lock assigned for each benchmark is shown in Table 2.

We then ran the set of benchmarks with and without lock assignment on Sun-Fire 10-processor 750MHz machine. Denote T_s as the execution time of the benchmark with a single lock for all critical sections, and T_{MLA} as the execution time of the benchmark with lock assignment using our MLA heuristic. Figure 8 shows the performance improvement of our lock assignment with respect to the single global lock, i.e., $(T_s - T_{MLA})/T_s$, running on different number of processors. Cholesky and Radiosity have shown a performance improvement of 30.17% and 14.76%, respectively, due to the decrease of lock contentions. On the other hand, Barnes, Ocean-cont and Water-nsq show a much lower performance improvement for two main reasons. First, the amount of time spent on critical sections is a small portion of the totally execution time. For instance, as shown in Table 2, for Ocean-cont, the time spent on critical sections takes only 0.11% of the total execution time. Second, these benchmarks contain critical sections that require more sophisticated static analysis, such as array analysis and shape analysis, to compute concurrency graphs. For the lack of a better term, we call such critical sections "non-analyzable" critical sections in this paper. Table 2 shows the relative time spent in analyzable and non-analyzable critical sections. Both Barnes and Water-nsq spend more time in non-analyzable critical sections than analyzable critical sections, therefore we get a much lower performance improvement. Note that even for non-analyzable critical sections we do coarse-grained analysis such as treating an array as a scalar unit. Such coarse-grained analysis is very efficient for Cholesky.

4.3 Performance Study Using Cyclops-64 Simulator

In this section we present our empirical results using Cyclops-64 (C-64) simulator. For the experiment we chose Unstructured Adaptive (UA) benchmark from the NAS Parallel Benchmark suite, which solves a "stylized heat transfer problem on a cubical domain with Dirichlet boundary conditions" [13]. We tested the performance of UA on a simulator of a single C-64 (C64) processor [10, 9], with and without lock assignment. The full C64 system is a petaflop supercomputer currently being developed at IBM and the University of Delaware. It exploits the Many-Core System-on-a-Chip (SoC) technique to achieve high computation rates, low power consumption and low cost. The C64 system consists of 13,824 C64 chips connected by a 3D mesh network. It delivers 1 petaflop peak performance. To conduct our experiments, we ported Omni [20] C compiler and its runtime system to C-64, and the runtime system was optimized to exploit C64 hardware features [7].[5]

UA is written in Fortran/OpenMP. In order to test it on C64 architecture, we first manually translated the parallel region in function `transfb_c_2` into an independent C/OpenMP program, and preserved the execution environment, including all global variables and its pa-

[5]C-64 chip is still under development and so we are using a simulator to conduct our experiments.

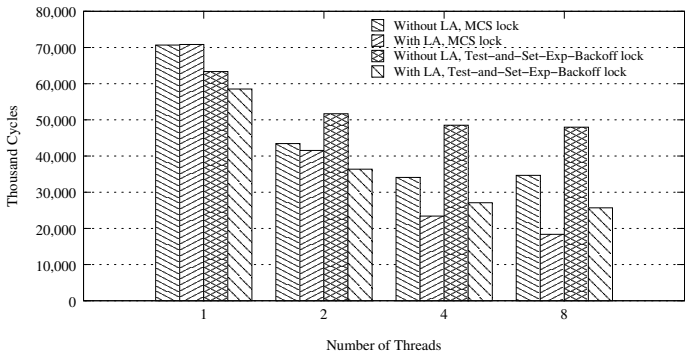


Figure 9: UA with/without Lock Assignment on C64 platform

rameters, during the translation.[6] We ran the translated code with problem size S on C-64 FAST simulator [8], and report the execution results in Figure 9.

The function `transfb_c_2` within the application contains 18 concurrent critical sections in a parallel region. Nine of these critical sections access array *tmort* and array *tx*, and the remaining nine critical sections access array *mormult*. The index of each array access is dynamically calculated. Each critical section performs at most three memory accesses and at most two mathematical operations. The resulting concurrency graph is a clique of size 18. In this graph, one half of the edges are non-interfering edges, and the other half of edges are interfering edges. Our lock assignment heuristic gives an optimal solution – assigns one lock to critical sections accessing *tmort* and *tx*, and the other lock to critical sections accessing *mormult*.

To verify the effectiveness of the lock assignment, two widely accepted spin lock algorithms were used in the experiments. One is the Test-and-Set (TS) lock with exponential backoff, and the other is the linked-list based MCS lock [19], both customized for the C64 [7]. Figure 9 demonstrates that the lock assignment improves the performance in all multi-thread cases. When running with 8 threads, for both TS and MCS locks, the programs with lock assignment are about two times faster than those without lock assignment. This speedup comes from the fact that at any time there are at most two critical sections that can be executed in parallel without violating the isolation semantics. By assigning two different locks to them, we allow them to execute concurrently and therefore explore the parallelism to achieve nearly a 2× speedup compared to the default single-lock implementation for critical sections. The program with lock assignment also shows 2× speedup when running on up to 128 threads.

5 Extensions

There are two categories of problems in optimizing synchronizations in multithreaded programs: (1) what is the minimum resource requirement to exploit the maximum parallelism in a multithreaded program? (2) What is the minimum cost we have to pay if the resource is limited?

[6]Currently C-64 simulator can only run C programs.

These two categories of problems are tightly-coupled and complement to each other. Our MLA problem falls into the first category where locks are our concerned resource. We claim that the key to solve problems in both categories lies in how to model the potential parallelism and the orders (dependences) or constraints that must be preserved to guarantee a correct execution. In our MLA case, the constraint we must preserve is the isolation semantics of critical sections, and this constraint and the potential parallelism are modeled as interfering edges and non-interfering edges in the concurrency graph, respectively. We expect that the way we formulate and solve the MLA problem has revealed some clues on how to solve problems in category one generally.

In this paper, we also propose and solve MLA's counterpart in category two. We formulate it as the K-Lock Allocation (K-LA) problem, which allocates K locks to a multithreaded program in order to minimize the "serialization cost". Although it is unclear that the K-LA technique is instantly applicable to the current mainstream architectures, it is a good time to explore the general solution for category two problems for future use. In the remaining part of this section, we first present our serialization cost model (see Section 5.1), then formulate the K-LA problem. The K-LA problem is also NP-hard, and we present a tentative heuristic (see Section 5.2). Once again we formulate the K-LA problem as an ILP problem, and tested the optimality of the K-LA heuristic by comparing it with the optimal solutions produced by the ILP solver. Some preliminary results are shown in Section 5.4.

5.1 Serialization Cost Estimation and K-LA Formulation

Before we formulate a problem in category two, we need to define the cost model. In our K-LA case, locks are still the concerned resource. If the number of locks required to explore the parallelism in the program (as obtained by the MLA solution) is unavailable, we need to sacrifice some parallelism by forcing some critical sections that can be run concurrently to be serialized. The extra cost caused by the forced serialization is called "serialization cost".

Our serialization cost model is constructed based on a simple observation. Given a critical section cs, let $Cost(cs)$ denote the cost of executing cs. For our purpose it is the maximum amount of time required to execute the body of cs by one thread [7]. Now let cs_1 and cs_2 be two concurrent non-interfering critical sections. Assume there is an infinite number of threads. If cs_1 and cs_2 are executed in parallel by two different threads then the cost of executing them is the maximum of the two costs, that is, $Max(Cost(cs_1), Cost(cs_2))$. On the other hand if cs_1 and cs_2 are serialized (say, by allocating the same lock to them), the cost of executing them would be the sum of their costs, that is, $Cost(cs_1) + Cost(cs_2)$. Therefore the serialization cost $SC(cs_1, cs_2)$ for a non-interfering edge $e = (cs_1, cs_2)$ is given by the difference between the cost of executing cs_1 and cs_2 in sequence and the cost of executing them in parallel:

[7] A critical section can have conditional statements so we consider the maximum time.

$$
\begin{aligned}
SC(e) &= SC(cs_1, cs_2) \\
&= Cost(cs_1) + Cost(cs_2) - \\
&\qquad Max(Cost(cs_1), Cost(cs_2)) \\
&= Min(Cost(cs_1), Cost(cs_2)) \qquad (11)
\end{aligned}
$$

The serialization cost for a non-interfering edge is essentially the "extra cost" that is incurred when the two critical sections are executed in sequence, and this extra cost is same as the minimum of the two costs.

Now recall that given a concurrency graph G, the maximum parallelism in G is the number of non-interfering concurrency edges in G. When we serialize such edges we loose parallelism and the serializing cost of G is given by the sum of all non-interfering edges that have been serialized:

$$
SC(G) = \sum_{e \in E_n} SC(e) \qquad (12)
$$

Based on the serialization cost model, we formulate the K-Lock Allocation problem as follows.

Problem 5.1 (K-Lock Allocation). *Given a parallel program with a set of critical sections, allocate K locks to critical sections such that*

(a) Two critical sections are assigned at least one common lock if (1) they are concurrent and (2) they access some common data and at least one of them write to the common location, and

(b) The serialization cost is minimized.

5.2 K-Lock Allocation Heuristic

The K-LA problem is related to the register allocation problem and it is NP-hard. In this section we present a simple heuristic for lock allocation. First we associate a weight function $W(u, v)$ for each edge $(u, v) \in E$:

$$
W(u,v) = \begin{cases} SC(u,v) & : \text{ for non-interfering edge } (u,v) \\ 0 & : \text{ for interfering edge } (u,v) \end{cases}
$$

When the lock resource is limited, we sacrifice the parallelism between critical sections cs_i and cs_j with the lowest serialization cost. If there are more than one edge with the lowest serialization cost, we arbitrarily choose one from them. We then transform G into G' by changing (cs_i, cs_j) from the non-interfering edge to the interfering edge, and perform the lock assignment (using our MLA heuristic described in Section 3.1) to see whether the given K

```
LockAllocation(G, K)

1.      calculate the cost of each cs ∈ V
2.      if K = 1
3.         assign a global lock to each critical section and return
4.      end if
5.      HLB = LOCK-ASSIGNMENT(G)
6.      while HLB > K do
7.         sort non-interfering edges according to their weights
8.         do
9.            get the next edge (u, v) from the sorted list
10.           transform G to G' by changing (u, v) into the interfering edge
11.           HLB' = LOCK-ASSIGNMENT(G')
12.        while HLB' < HLB
13.     end while
```

Figure 10: K-LA Algorithm

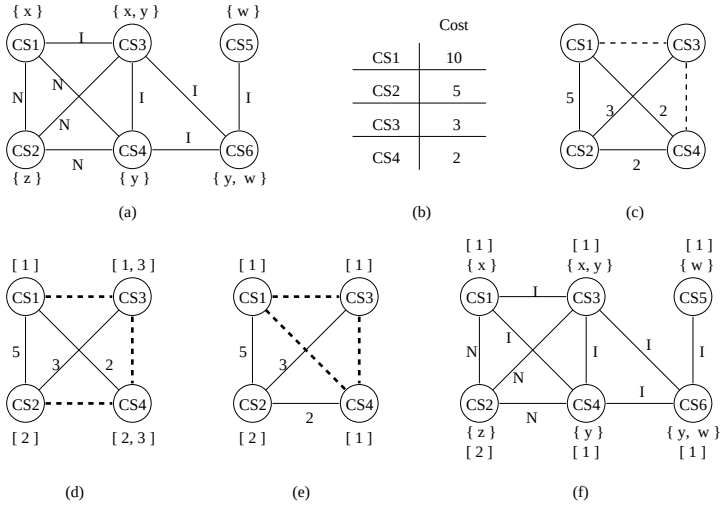


Figure 11: Example of lock allocation (a) Concurrency graph (b) Costs of critical sections (c) Serialization cost estimation (d) Lock assignment if serializing (CS_2, CS_4) (e) Lock assignment if serializing (CS_1, CS_4) (f) Final lock assignment using 2 locks

locks is sufficient for G'. Note that (cs_i, cs_j) becomes a serializing interfering edge in G', and G' may need more locks than G. In this case we will choose the second lowest serialization cost edge. This process is repeated until K locks is sufficient for lock assignment. This approach is summarized in Figure 10.

For example, we have known from Section 3 that the concurrency graph shown in Figure 11(a) requires 3 locks. Now we consider to allocate 2 locks to this graph. Assume we have known the cost of each critical section in this graph, as shown in (b). (c) shows the serialization cost of each non-interfering edge. We have two edges with the lowest serialization cost, (CS_2, CS_4) and (CS_1, CS_4). If choosing (CS_2, CS_4), we still have to use 3 locks, due to the interference between CS_3 and CS_4. If choosing (CS_1, CS_4), we successfully use 2 locks to color

the graph. (d) and (e) illustrate these two cases, respectively. The final lock allocation result is shown in (f).

5.3 ILP Formulation

In this section, we formulate the K-LA problem into an integer linear programming problem. Given a concurrency graph G, k locks and the serialization cost of each edge in G, we introduce 0-1 variables $f_{u,i}$ to indicate whether lock i is assigned to vertex u in G, where $1 \leq u \leq |V|$, and $1 \leq i \leq k$. Since each critical section must be assigned at least one lock, we have the following constraint:

$$f_{u,1} + f_{u,2} + \cdots + f_{u,k} \geq 1 \quad \text{for all } u \in G \tag{13}$$

Similar as in Section 3.2, we introduce 0-1 variable $s_{u,v,i}$ to indicate whether vertices u and v share lock i, $1 \leq i \leq k$, then $s_{u,v,i} = f_{u,i} \wedge f_{v,i}$. This condition is imposed by the following constraints:

$$f_{u,i} + f_{v,i} \geq 2 \times s_{u,v,i} \tag{14}$$

$$f_{u,i} + f_{v,i} \leq 2 \times s_{u,v,i} + 1 \tag{15}$$

We use 0-1 variable $s_{u,v}$ to indicate whether u and v share any lock. $s_{u,v} = s_{u,v,1} \vee \cdots \vee s_{u,v,k}$, and we have the following two constraints:

$$s_{u,v,1} + \cdots + s_{u,v,k} \geq s_{u,v} \tag{16}$$

$$s_{u,v,1} + \cdots + s_{u,v,k} \leq k \times s_{u,v} \tag{17}$$

Any pair of interfering critical sections should share some lock. This condition is simply represented as:

$$s_{u,v} \geq 1 \quad \text{for interfering edge } (u, v) \tag{18}$$

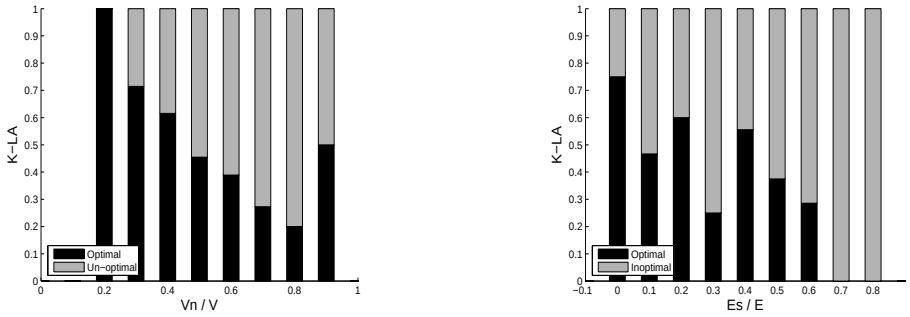
The total serialization cost introduced by the lock allocation is:

$$T = \sum_{(u,v) \in E} W(u, v) s_{u,v} \tag{19}$$

The K-LA problem is therefore to minimize T subject to inequalities (11) to (16).

5.4 Optimality Evaluation

Similar as the optimality evaluation for the MLA heuristic, we implemented the ILP formulation of the K-LA problem in the commercial ILP solver CPLEX, and tested the K-LA heuristic and the ILP formulation on a set of random graphs. From those 300 randomly generated graphs shown in Section 4.1, we selected those for which our MLA heuristic procedures optimal



(a) kLA Vn / V

(b) kLA Es / E

Figure 12: Optimality of the K-LA heuristic

solutions, and the optimal solutions use more than 2 locks. We set $K = 2$. We also randomly assign the cost of ≤ 30 units for each vertex. In 45.6% of cases our K-LA heuristic can obtain optimal solutions with minimum serialization cost. Similar to the MLA heuristic, the K-LA heuristic also depends on the non-interfering subgraph size and the relative number of serializing interfering edges, as illustrated in Figure 12(a) and (b), respectively.

6 Related Work

We discuss two areas of related work: synchronization optimizations, and transactional memory.

6.1 Synchronization Optimizations

Synchronization optimizations for multithreaded programs is of increasing recent interest, while the locking technique has been thoroughly studied in the area of concurrency control [14]. One of the key issues of locking technique is to identify the lock granularity. This is similar to our goal of refining the global lock to explore parallelism. The main difference between the locking in concurrency control and our lock assignment is that the locking algorithm considers currently executing transactions. It assigns locks dynamically, and cannot change the program generating the locking requests. The lock assignment algorithm, on the other hand, statically analyzes the parallel program and optimizes it during the compilation time.

Some optimization techniques on locks have been reported recently. Diniz and Rinard [11] presented two lock coarsening techniques to reduce the overhead of fine grained locks in Java programs. One is data lock coarsening, which tries to associate one lock with multiple objects that tend to be accessed together. The other is computation lock coarsening, which reduces the computation that repeatedly acquires and releases the same lock. Both techniques decrease the lock granularity in object-oriented programs with atomic operations. Choi et.al. [6] remove unnecessary thread-local synchronizations that can only be executed by a single thread using escape analysis. Aldrich et.al. [1] reduce three forms of unnecessary synchronization from Java

programs: thread-local synchronization, reentrant synchronization and enclosed lock synchronization. The thread-local synchronization elimination is similar as Choi's work. The reentrant synchronization elimination and the enclosed lock synchronization are similar as computation lock coarsening and data lock coarsening, respectively. Our lock assignment and allocation are working on the opposite direction. They start from the global lock – the lock with the least granularity, and refine it to explore the parallelism in the program.

Recently Emmi et.al. [12] proposed a lock allocation technique which takes a multithreaded program annotated with *atomic* sections and infers a lock assignment to *atomic* sections to preserve its atomicity and deadlock freedom. At the presence of pointers, they formulates the lock allocation problem as an ILP problem which minimizes the conflict cost between atomic sections and minimizes the number of locks. No heuristic solution is discussed in their work. Our lock assignment and allocation problems were proposed in July 2005 [26] and they are more general than Emmi et.al.'s work. First of all, our lock assignment is similar as the lock allocation technique in Emmi et.al.'s work, and our heuristic produces near-optimal solutions compared with our ILP formulation. Second, we propose two categories of synchronization optimization problems, and show how to solve both categories of problems using the concurrency graph. Third, our heuristics based on the concurrency graph can avoid deadlocks, which is an inherent drawback of Emmi et.al.'s formulation.

As critical sections and locks are code-centric approaches to ensure that a sequence of statements in a multithreaded program are executed atomically, Vaziri et al. [29] introduced a data-centric approach for writing concurrent programs using atomic sets. An atomic set is a set of locations that have "similar" data consistency properties. Accesses to fields in an atomic set are assumed to take place atomically in a "unit of work". In the data-centric approach, programmers explicitly annotate a set of locations as belonging to the same atomic set, and units of work are inferred and translated into synchronized block by compiler. Our work complements Vaziri et al.'s work in that we can analyze and determine the atomic sets and units of work using our concurrency analysis and lock assignment algorithm. Data accessed within a critical section and controlled by the same set of locks all belong to the same atomic sets. Our starting point is a single global lock and we show how to assign locks to improve concurrency. Vaziri et al. use explicit data annotation to define atomic sets and then propose a code generation framework for assigning locks to units of work. Currently the units of work are at method level and so it is difficult to assign locks that finer than method-level locks. Xu et al. [30] introduced a method to detect serializability violation for shared memory server programs. Their model does not require any *priori* program annotation, like critical sections in our framework, and atomic sets in Vaziri et al's work. Instead, the dynamic detector automatically infers approximations of atomic regions, called *computational units*.

There have been optimization techniques reported for other synchronization constructs besides critical sections and locks. Zhang and Evelyn [31] tried to match the barrier statements in the program and figure out any synchronization errors due to the misplacement of barriers. Tseng [28] reduced synchronization overhead of barrier constructs by either eliminating it or replacing it with less expensive forms of synchronization.

IBM 801 database Storage [21] supports lock attribute bits on virtual memory at 128 byte granularity. Such lock bits support can be used as lock resource for our lock assignment and allocation algorithms.

Synchronization State Buffer (SSB) [32] is a hardware mechanism to support fine-grained synchronization on many-core architectures. It records and manages states of active synchronized data units to efficiently support word-level fine-grain synchronization, which can enforce mutual exclusion or read-after-write data dependencies between concurrent threads. SSB-based synchronization solution is able to smoothly and efficiently handle the cases where the precise synchronization point cannot be resolved statically at compile time, therefore it is complement to our static lock assignment method.

6.2 Transactional Memory

Transactional memory is a dynamic conflict resolving technique that is based on a non-blocking synchronization mechanism [17, 25, 22, 23, 15, 3, 24]. A transaction is a sequence of memory reads and writes executed by a single thread. Transactions have two main properties: serializability and atomicity. The serializability requires that transactions appear to be executed serially and never appear to be interleaved with each other. This property is a "narrow-down" of the isolation semantics [14] of critical sections and transactions in the area of concurrency control. Programmers need to explicitly mark the transaction regions, and the system, either hardware or software, dynamically resolves the conflicts, by either successfully committing the transactions or aborting and then rolling back.

Distinguished from the transaction memory, critical sections and locks are blocking synchronization mechanisms, and our lock assignment complements the transactional memory in two aspects: (1) when to resolve the conflicts, and (2) where and how to resolve the conflicts. For (1), lock assignment resolves the conflicts among critical sections statically during the compilation time while the transaction memory resolves the conflicts dynamically during the execution time. Lock assignment can successfully resolve most of the coarse-grained conflicts, thus it can simply the dynamic job of the transactional memory. However, due to nature of the static analysis and optimization, lock assignment is less accurate when the memory is accessed in a dynamic pattern, in which cases transactional memory turns out more effective. For (2), critical sections require locks before their execution of body statements. Locks play the role of "permission" of the execution. One can consider critical sections as "committing at the beginning", and the commitment is successful when all locks are successfully required. No roll-back is necessary. On the contrary, Transactional memory allows the body statement to begin as early as possible and it "commits at the end". The commitment is successful if no interleaving is generated. Roll-back is necessary if the commitment fails.

Note that our approach can also be easily extended to optimistic lock architecture with abort based on transactional memory [18].

There have been several language proposals for transactional memory support with the use of `atomic` construct that essentially replaces the `synchronized` construct of Java (and

other languages) [5, 16, 2]. The underlying semantics of `atomic` construct varies between weak atomicity and strong atomicity, and also how and when to abort or roll-back is effected. Our isolation semantics is closely related to the semantics of atomicity, except that we do not model roll-back semantics of the transactional memory.

7 Conclusions

In this paper we proposed and solved two problems related to optimized assignment of locks to coarse-grained critical sections in the context of OpenMP and Pthread models. The first problem, Minimum Lock Assignment (MLA), addresses the problem of finding the minimum number of locks needed to enforce mutual exclusion among interfering critical sections without any loss of concurrency. The second problem, K-Lock Allocation (K-LA) addresses the problem of allocating a fixed number (K) of locks to critical sections so as to minimize the serialization overhead. Experimental results are very encouraging and show that our methods can be used to improve the performance of parallel programs with mutual exclusion by exploiting concurrency among multiple critical sections. An extension of this work to support reader/write locks is a subject for future work.

Acknowledgement

This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004 as part of the IBM PERCS project.

References

[1] Jonathan Aldrich, Emin Gun Sirer, Craig Chambers, and Susan J. Eggers. Comprehensive synchronization elimination for java. *Science of Computer Programming*, 47(2-3):91–120, 2003.

[2] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification.* Sun Microsystems, 2005.

[3] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, pages 316–327. Feb 2005.

[4] Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, Houston, Texas, April 1992. Published as Rice COMP TR92-183.

[5] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented

approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, 2005.

[6] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, 2003.

[7] Juan del Cuvillo, Weirong Zhu, and Guang R. Gao. Landing OpenMP on Cyclops-64: An efficient mapping of OpenMP to a many-core System-on-a-chip. In *the 3rd ACM International Conference on Computing Frontiers (CF2006)*, May 2006.

[8] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. FAST: A functionally accurate simulation toolset for the Cyclops64 cellular architecture. In *Workshop on Modeling, Benchmarking, and Simulation (MoBS2005)*, June 2005.

[9] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Toward a software infrastructure for the Cyclops-64 cellular architecture. In *the 20th International Symposium on High Performance Computing Systems and Applications (HPCS2006)*, St. John Rs, Newfoundland and Labrador, Canada, May 2006.

[10] Monty Denneau. Computing at the speed of life: The BlueGene/Cyclops supercomputer. CITI Distinguished Lecture Series, Rice University, Sep 25th, 2002.

[11] Pedro C. Diniz and Martin Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. In *LCPC'96, Languages and Compilers for Parallel Computing*, pages 284–299, 1996.

[12] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 291–296, 2007.

[13] Huiyu Feng, Rob F. Van der Wijngaart, Rupak Biswas, and Chatherine Mavriplis. Unstructured adaptive (ua) nas parallel benchmark, version 1.0. Technical Report NAS-04-006, NASA Ames Research Center, Moffett Field, CA, July 2004.

[14] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[15] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, Jun 2004.

[16] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, 2003.

[17] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, 1993.

[18] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. In *The 33rd Annual International Symposium on Computer Architecture (ISCA2006)*, June 2006.

[19] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization onshared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[20] Omni OpenMP Compiler Project. Omni OpenMP Compiler. In http://phase.hpcc.jp/Omni/home.html [September, 2004].

[21] George Radin. The 801 minicomputer. *IBM Journal of Research and Development*, 27(3):237–246, 1983.

[22] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *the 34th International Symposium on Microarchitecture*, pages 294–305, Dec 2001.

[23] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the Tenth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 5–17. Oct 2002.

[24] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *the 24th ACM Symposium onf Principles of Distributed Computing (PODC'05)*, Las Vegas, Nevada, July 2005.

[25] Nir Shavit and Dan Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[26] Vugranam C. Sreedhar, Yuan Zhang, and Guang R. Gao. A new framework for analysis and optimization of shared memory parallel programs. Technical Report CAPSL-TM-063, University of Delaware, Newark, DE, 2005.

[27] The Stanford FLASH Prjoect. Stanford parallel applications for shared memory (SPLASH) benchmark. In http://www-flash.stanford.edu/apps/SPLASH/ [November 2005].

[28] Chau-Wen Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 144–155, Santa Barbara, California, July 1995.

[29] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL'06: Conference record of the 33rd ACM*

SIGPLAN-SIGACT symposium on Principles of programming languages, pages 334–345, 2006.

[30] Min Xu, Rastislav Bodik, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–14, 2005.

[31] Yuan Zhang and Evelyn Duesterwald. Barrier matching for programs with textually unaligned barriers. In *PPoPP '07: Proceedings of the twelfth ACM SIGPLAN symposium on Principles and practice of parallel programming (to appear)*, 2007.

[32] Weirong Zhu, Vugranam C. Sreedhar, Ziang Hu, and Guang R. Gao. Synchronization state buffer: Supporting efficient fine-grain synchronization on many-core architectures. In *ISCA '07: Proceedings of the 34th International Symposium on Computer Architecture (to appear)*, 2007.