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

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

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

CAPSL Technical Memo 65
April 24th, 2006

Copyright © 2005 CAPSL at the University of Delaware

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

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

One of the major 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 two technical problems related to optimized assignment of locks to critical sections. 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.

Contents

1 Introduction **1**
1.1 OpenMP and Motivation 1
1.2 Problem Statement 4
1.3 Main Contributions 5

2 Concurrency Graph and Critical Sections **5**
2.1 Concurrency Graph 5
2.2 Non-interfering Critical Sections 6
2.3 Interfering Critical Sections 7
2.4 Concurrency Graph Partition 7
2.5 Serializing Non-Interference Graph 8

3 Minimum Lock Assignment Solution **9**
3.1 MLA Heuristic 9
3.2 ILP Formulation 13

4 K-Lock Allocation Solution **14**
4.1 Serialization Cost Estimation 14
4.2 Lock Allocation Heuristic 15
4.3 ILP Formulation 15

5 Experimental Results **17**
5.1 Optimality Evaluation 17
5.2 Omni Compiler and Benchmark 18
5.3 Performance Study on Cyclops-64 20

6 Related Work **21**

7 Conclusions **22**

A The MLA Problem is NP-Complete **27**

B Proof of Theorem 3.1 **27**

C Proof of Theorem 3.2 **27**

D Proof of Theorem 3.3 **27**

E Omni OpenMP Compiler (OOC) **28**

List of Figures

1 (a) Example program (b) Concurrency graph and the Minimum Lock Assignment solution (within brackets) (c) 2-Lock Allocation solution (within brackets) 3
2 An example of (a) Non-interfering critical sections and (b) Interfering critical sections. In the figure, [ ] denotes the lock assignment. 6
3 Example SNIG for Observation 2.1 9
4 (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 10

5 Lock Assignment Heuristic 11
6 K-LA Algorithm 15
7 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 16
8 Optimality of MLA and K-LA heuristics 19
9 Structure of the parallel region in function *transfb_c_2* 20
10 UA with/without Lock Assignment on C64 platform 21
11 Omni OpenMP Compiler and Runtime System Infrastructure 29

1 Introduction

One of the major 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. Given that processors in current and future computer systems are becoming multi-core by default, this productivity issue is gaining in importance for mainstream programming in addition to high performance computing.

In this paper, we propose to achieve this ideal by (1) letting programmers focus on the correctness of the application by using 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. We introduce and solve two technical problems related to this optimized assignment of locks to critical sections. 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. We adopt isolation semantics from transaction processing as the intended semantics for coarse-grained critical sections, and we guarantee the correctness of our lock assignments with respect to the isolation semantics [11].

We present our approach in the context of the OpenMP programming model, though this approach is applicable to any other parallel programming model with mutual exclusion such as pthreads [16], Java [31] and X10 [4]. The productivity benefits of using critical sections instead of explicit locks in OpenMP was confirmed in a recent productivity study reported in [18]; the results presented in our paper can be used to improve the scalability by exploiting concurrency among multiple critical sections, compared to the default OpenMP implementation of using a single lock to control all critical sections.

1.1 OpenMP and Motivation

OpenMP [23] is a standardized set of (pseudocomment) language extensions and API's for writing shared memory parallel applications in C/C++ and FORTRAN. OpenMP contains three kinds of constructs to support mutual exclusion: (1) *lock* routines for explicit lock management, (2) *atomic* constructs for a single read-modify-write operation on a single location, and (3) named and unnamed *critical* constructs. These constructs are used in both regular and

irregular parallel applications. For example, four of the eleven SPEC OMP benchmarks [30] use one of these constructs (310.wupwise, 326.gafort, 328.fma3d, 332.ammp), and seven of the eleven OpenMP implementations of the NAS Parallel Benchmarks [21] use one of these constructs (BT, DC, EP, LU, LU-HP, SP, UA). Some of the SPLASH-2 benchmarks [34] (e.g., radiosity) also use mutual exclusion constructs.

Despite these examples, the adoption of mutual exclusion constructs is still quite limited among parallel application developers. We believe that this is primarily because current implementations of OpenMP critical sections incur significant overhead. As an example, consider the comments in the README file of the UA benchmark [21] which recommends the use of atomic constructs for updating sparsely overlapped mortar points, but observes that a data parallel version will be a more efficient approach if the atomic constructs are implemented using critical sections. Unfortunately, the data parallel version results in a loss of productivity because it forces the programmer to explicitly keep track of which array elements are assigned to which threads. We expect that the demand for mutual exclusion constructs will continue to increase in the future as more and more irregular applications are enabled for parallelism.

The region associated with an OpenMP critical construct can include multiple statements, unlike an atomic construct which only applies to a single statement. The critical construct enforces mutual exclusion with respect to all critical constructs with the same name. Note that all unnamed critical constructs are considered to have the same (unspecified) name. Though atomic and critical constructs were provided in OpenMP as a convenience compared to explicit locks, there is still a lot of complexity associated with their use. For example, there is no guarantee of mutual exclusion between an atomic construct and a critical construct. Also, it is possible for a programmer to introduce deadlock when working with named critical sections. To avoid these complexities, we will focus our attention on unnamed critical sections in OpenMP as the sole construct used by the programmer to enforce mutual exclusion.

Consider the simple OpenMP program shown in Figure 1(a). [1] 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 parallel 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 approach 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. [2] Each critical section can belong to zero or more lock classes. A lock class can be implemented in a portable way by having the compiler generate OpenMP code that uses named critical constructs or explicit lock management, or in

[1] Two sections are listed horizontally to simplify the presentation.

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

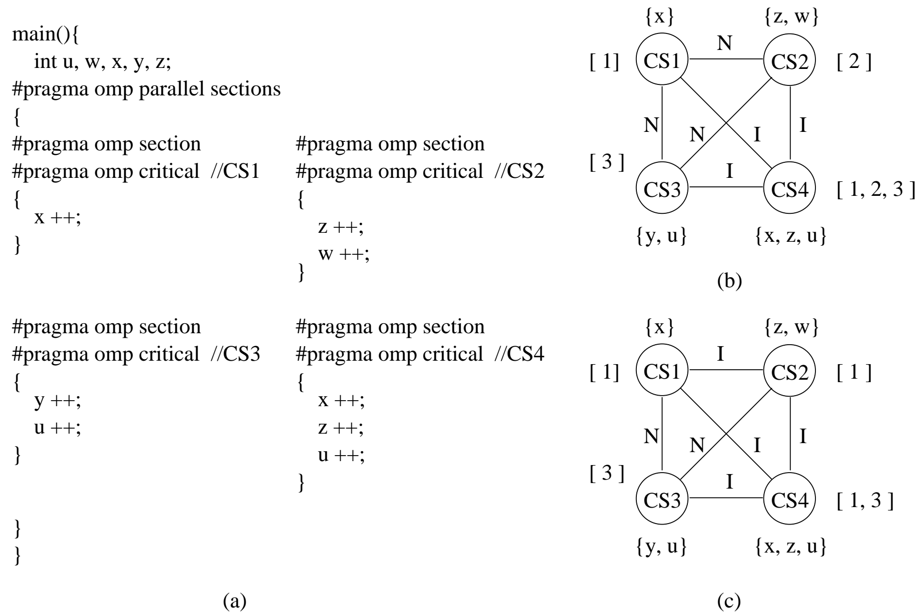


Figure 1: (a) Example program (b) Concurrency graph and the Minimum Lock Assignment solution (within brackets) (c) 2-Lock Allocation solution (within brackets)

a platform-specific way by generating special hardware instructions for lock management.

In this paper we present a framework for assigning multiple locks to a set of unnamed critical sections in a way that it preserves the semantics of unnamed critical sections. For example, using the approach described in this paper we can use three locks to control the critical sections in Figure 1(a). The results are shown as a *labeled concurrency graph* in Figure 1(b). Each vertex in the 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. However, if the number of locks required by the minimum solution is unavailable, a legal lock assignment can still be achieved by sacrificing some parallelism. For example, a solution with two locks is shown in Figure 1(c), where critical sections CS_1 and CS_2 are serialized even though they could potentially execute in parallel.

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 [11]. In order to avoid deadlocks, the set of locks should be acquired in a predetermined order [11].

The correctness of our framework for lock assignment and lock allocation is based on the semantics of isolation and locking defined in the area of transaction processing and concurrency control [11]:

- **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.*

1.2 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 an OpenMP 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 different locks if they are concurrent and do not access any common data, and

(b) Two critical sections are assigned at least one common lock if they are concurrent and access some common data.

Assume all critical sections are analyzable [26], that is, the 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.

The next problem we will address in this paper is the **K-Lock Allocation** (K-LA) problem, in which we are allowed to use at most K locks to control critical sections.

Problem 1.2 (K-Lock Allocation)**.** Given an OpenMP 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 they are concurrent and access some common data, and

(b) The serialization cost is minimized.

Informally, the serialization cost is defined as the "extra" cost incurred when we force to serialize two critical sections that can be run concurrently without violating the isolation semantics (see Section 4). If K locks is sufficient to exploit all the parallelism in the program, then the K-LA problem is same as the MLA problem, and the serialization cost is trivially zero. Otherwise, we need to find a solution with the minimum serialization cost.

1.3 Main Contributions

The main contributions of this paper are as follows:

- We propose and formulate the MLA problem and the K-LA problem to exploit the concurrency among critical sections in an OpenMP program. We show that the MLA problem is related with the classical graph coloring problem, and the K-LA problem is related with the register allocation problem.
- We show that both MLA problem and K-LA problem are NP-hard. We then present heuristics for solving the two problems.
- We formulate both problems into the Integer Linear Programming (ILP) problems, and evaluate our heuristics by comparing their solutions with the optimal solutions 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, and the K-LA heuristic produces optimal solutions for nearly half of them.
- We have implemented our heuristic approaches in the Omni [22] OpenMP compiler, and evaluated its performance on the Cyclops-64 [7] architecture - a state-of-the-art multi-core, using the UA benchmark from NAS Parallel Benchmarks. We have observed a 2× speedup compared to the default single lock implementation for critical sections in function `transfb_c_2`.

2 Concurrency Graph and Critical Sections

The concurrency and interference relation among critical sections in an OpenMP 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 static critical section, and there is an edge $(u, v) \in E$ if instances of critical sections u and v may be concurrent.

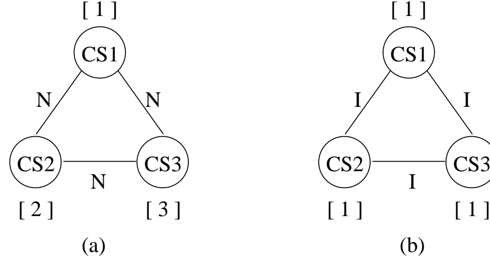


Figure 2: An example of (a) Non-interfering critical sections and (b) Interfering critical sections. In the figure, [ ] denotes the lock assignment.

In the above definition, if an instance of critical section u is concurrent to another instance of u, then we introduce a self-loop at u. This can happen when u is inside a parallel for loop. 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 [29]. For convenience, we assume that critical sections are analyzable and never nested. 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 above by labeling an edge (u, v) with label I whenever critical sections u and v are interfering, and with label N whenever u and v are non-interfering. Note that given a labeled concurrency graph, we need to first perform a pre-pass to remove all vertices that have only non-interfering edges (and all corresponding incident non-interfering edges), since locks are not needed for such critical sections. The resulting graph may induce a forest of concurrency graphs, and we analyze each connected component independently. In the following discussion, we simply assume that a concurrency graph G after the pre-pass is a connected graph. To simplify the presentation we also ignore self-loops in the rest of the paper.[3]

2.2 Non-interfering Critical Sections

Now consider a class of OpenMP programs P_n whose corresponding concurrency graph contains only non-interfering edges. Figure 2(a) illustrates an example of such *non-interfering*

[3]We ignore them because self-loops unnecessarily complicated the solution of lock assignment.

concurrency graph.[4] We observe that each vertex (critical section) in a non-interfering concurrency graph can be assigned a unique lock. Furthermore, we can minimize the total number of locks that are needed by observing that two critical sections require two different locks only if they are concurrent (connected). 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[5]. 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.

2.4 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 4(b) illustrates the G_n subgraph of (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 4(c) illustrates the interfering subgraph for our example. 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 occur in the non-interfering subgraph, for example, edges (CS_1, CS_3) and (CS_3, CS_4). 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

[4]This graph can be obtained when CS_1, CS_3 and CS_3 are in a parallel for loop.

[5]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.

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 4(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 4(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 constraint the inherent parallelism in a non-interfering concurrency graph. They also constraint the minimum number of locks for a SNIG. We can restate the MLA problem (Problem 1.1) for SNIG 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 3. 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 3 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

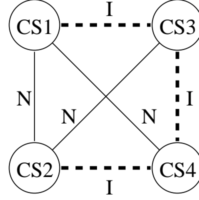


Figure 3: Example SNIG for Observation 2.1

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 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, as shown in Appendix A. 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).

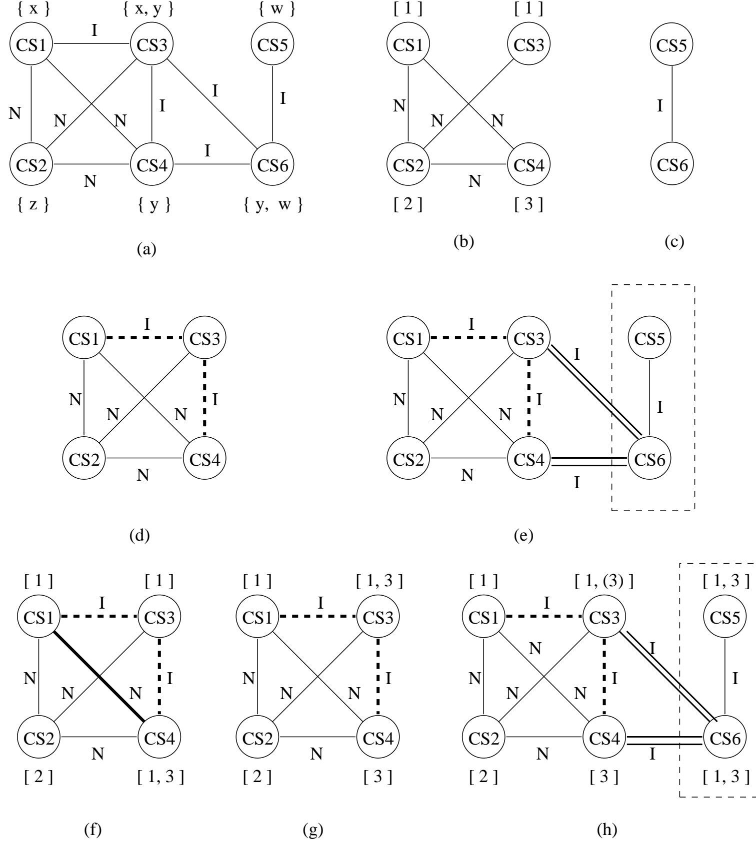


Figure 4: (a) A general concurrency graph (b) The non-interfering subgraph G_n (c) The interfering subgraph G_i (d) The SNIG G_n^s (e) The crossing edges (double lines), serializing interfering edges (dotted lines), and the interfering subgraph (in dotted box) (f) A un-safe borrowing from CS_3 to CS_4 (g) A safe borrowing from CS_4 to CS_3 (h) Final lock assignment result

The first step is straightforward. We use a heuristic graph coloring algorithm [33] to color G_n, and the result for our example is shown in Figure 4(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 in Figure 5. In Figure 4(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

```
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

$Lock(v) \cap NIN(u) = \emptyset$, 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 4(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 lock 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 4(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 4(h). We refer to the number of locks required to color G as the Heuristic Lock Bound (HLB). We have mentioned in Section 1.2 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, 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, proven in Appendix B and Appendix C, respectively, show that the lock assignment algorithm 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.*

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

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 significance of this lemma is that in order to assign locks to an arbitrary concurrency graph G, we only need to consider the SNIG of G. The following theorem, proven in Appendix D, 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.*

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, $1 \leq u \leq |V|$, and $1 \leq i \leq |M|$, 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:

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

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:

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

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

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:

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

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

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:

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

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

Then

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

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

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).

4 K-Lock Allocation Solution

In the MLA problem discussed in the previous section, we assume there are unlimited number of resources (locks), and the goal is to find the minimum number of locks to achieve the maximum parallelism. In this section, we consider the problem of allocating K locks to a concurrency graph that minimize the "serialization cost". We formulate this problem as the K-Lock Allocation (K-LA) problem. The K-LA problem is also NP-hard. In this section we first present our serialization cost model (see Section 4.1). Then we present a simple heuristic to solve the K-LA problem (see Section 4.2). Once again we formulate the problem as an ILP so that we can solve it optimally. We will use the ILP solution to quantitatively understand how good our heuristic is.

4.1 Serialization Cost Estimation

Our serialization cost model is 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 [6]. Now let cs_1 and cs_2 be two concurrent non-interfering critical sections. Assume there are 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:

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

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:

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

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

```
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 6: K-LA Algorithm

4.2 Lock Allocation Heuristic

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 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 6.

For example, we have known from Section 3 that the concurrency graph shown in Figure 7(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).

4.3 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

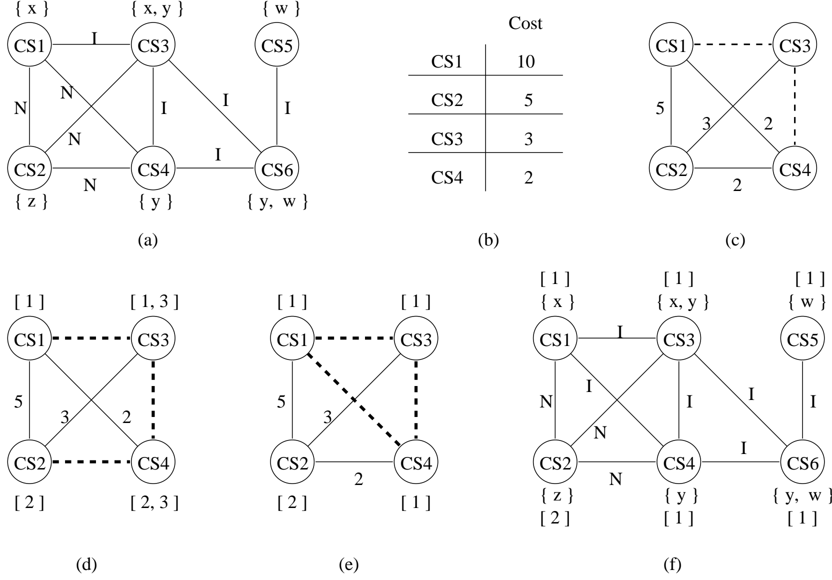


Figure 7: Example of lock allocation (a) Concurrency graph (b) Costs of critical sections (c) Serialization cost estimation (d) Lock assignment if serializing (CS_2, CS_4) (e) Lock assignment if serializing (CS_1, CS_4) (f) Final lock assignment using 2 locks

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:

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

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

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:

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

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

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

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

	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

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 Experimental Results

In this section, we present two sets of experiments to evaluate our lock assignment and allocation algorithms. First we compared the precision of our MLA and K-LA heuristics with the optimal solutions based on the ILP formulations on a set of 300 random concurrency graphs. Second we integrated our heuristics into the Omni [22] OpenMP compiler and we present our performance results for the UA benchmark in NPB benchmark suite on a Cyclops-64 processor [8, 7].

5.1 Optimality Evaluation

To study the precision of our MLA and K-LA heuristics we implemented our ILP formulations in the commercial ILP solver CPLEX, and tested both heuristics and ILP formulations on a set of 300 randomly generated concurrency graphs with characteristics shown in Table 1. Due to time constraints in the ILP solver, we limited our random concurrency graphs to contain at most 16 nodes.

For the MLA problem, our heuristic solution is optimal for 83.3% of graphs. For the remaining 16.7% of graphs our heuristic assigns more locks than the optimal solution, and in the worst case two extra locks than the optimal solutions. In Table 2, we present the number of test graphs for which our heuristic assigns d more locks than the optimal solutions for each graph size.

There are two reasons for our lock assignment heuristic being not optimal – the graph coloring of the non-interfering subgraph G_n and the existence of the serializing interfering edges E_s. To verify their influence, we calculated the optimality of the heuristic with the increase of the size of the 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, respectively. The results shown in Figure 8(a) and (b) illustrate that the optimality of our heuristic heavily depends on the non-interfering subgraph size and the relative number of serializing interfering edges.

# Vertex	d=0	d=1	d=2	Total
2	31	0	0	31
3	20	0	0	20
4	25	2	0	27
5	22	0	0	22
6	12	0	0	12
7	14	2	0	16
8	16	2	0	18
9	16	2	2	20
10	20	1	0	21
11	16	5	0	21
12	11	5	0	16
13	14	2	0	16
14	13	8	0	21
15	10	5	2	17
16	10	11	1	22

Table 2. Results of MLA heursitic on random concurrency graphs for Cases when MLA Heuristic is optimal (d=0), suboptimal by 1 lock (d=1), and suboptimal by 2 locks (d=2)

# Vertex	Optimal	Inoptimal	Total
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	1	1
8	3	0	3
9	6	0	6
10	4	6	10
11	2	4	6
12	4	3	7
13	5	6	11
14	3	5	8
15	1	6	7
16	3	6	9

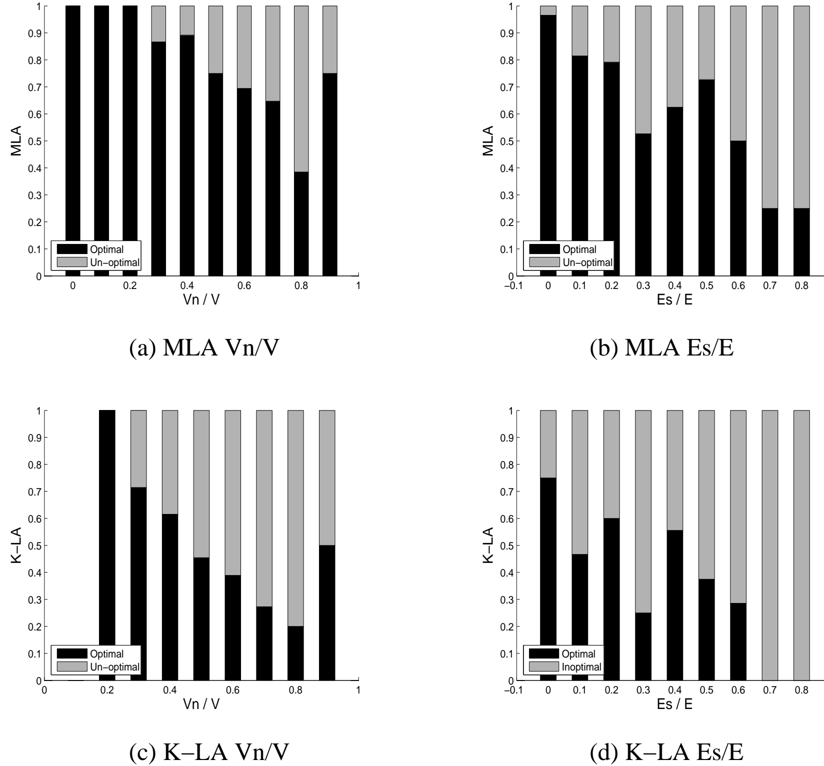
Table 3. Results of K-LA Heuristic on Random Concurrency Graphs ($K = 2$)

To test the K-LA heuristic, we selected those graphs for which our lock assignment heuristic produces optimal solutions, and the optimal solutions use more than 2 locks. $K = 2$. We also randomly assign the cost of ≤ 30 units for each vertex. In 45.6% of the cases our lock allocation heuristic can obtain optimal solutions with minimum serialization cost. The results are shown in Table 3. Similar to the MLA heuristic, the lock allocation heuristic also depends on non-interfering subgraph coloring and the relative number of serializing interfering edges, as illustrated in Figure 8(c) and (d).

5.2 Omni Compiler and Benchmark

To study the runtime performance of lock asignment we integrated our heuristics into the Omni OpenMP Compiler (OOC) [22] (Appendix E gives more details of OOC).

For experiments, we chose the OpenMP implementation of the Unstructured Adaptive (UA)



(a) MLA Vn/V

(b) MLA Es/E

(c) K–LA Vn/V

(d) K–LA Es/E

Figure 8: Optimality of MLA and K-LA heuristics

benchmark from the NAS Parallel Benchmark suite. The application solves a "stylized heat transfer problem on a cubical domain with Dirichlet boundary conditions" [10]. Unlike most of the other benchmarks in NPB that access memory in (mostly) fixed-stride patterns, UA's memory access patterns are irregular and continually changing. It is significant because it provides a standardized method for gauging the performance of computer systems for a class of scientific applications with irregular and dynamically changing memory accesses.

The program contains three major loops that update sparsely overlapped mortar points [17]. Parallelization of these loops requires atomic update of memory references by mortar points. These atomic updates were originally implemented by *atomic* directives. However, some *atomic* directives are not in accordance with the OpenMP standard, and cannot be mapped to a single instruction in our testing platform. Therefore we have translated all *atomic* directives into *critical* constructs that can preserve the mutual exclusion semantics of *atomic*.

The function `transfb_c_2` within the application contains 18 concurrent critical sections in a parallel region, as shown in Figure 9. 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

```
#pragma omp parallel for
{
  do(...)
    do(...) {
      ......
      #pragma omp critical { }      4 critical sections accessing
      ......                        tmort and tx
      #pragma omp critical { }      4 critical sections accessing
      ......                        mormult
      do(...) {                     5 inner loops, each contains 2
        #pragma omp critical { }    critical sections, one accesing
        #pragma omp critical { }    tmort and tx, the other
        ......                      accessing mormult
      }
      ......
    }
}
```

Figure 9: Structure of the parallel region in function *transfb_c_2*

lock to critical sections accessing *tmort* and *tx*, and the other lock to critical sections accessing *mormult*.

5.3 Performance Study on Cyclops-64

We tested the performance of UA on a simulator for a single Cyclops-64 (C64) processor [8, 7], with and without lock assignment. The full C64 system is a petaflop supercomputer currently being developed at IBM. 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. C64 is intended to serve as a dedicated computer engine for high end scientific applications such as molecular dynamics. A single C64 chip is composed of 160 hardware thread units connected with 160 embedded SRAM memory banks through a on-chip crossbar network. The C64 chip architecture exposes an explicitly visible and programmable memory hierarchy to programmers. It supports massive intra-chip parallelism – up to 160 threads can run non-preemptively and concurrently at any time. To conduct our experiments, the Omni [22] compiler and its runtime system were ported to C64, and the runtime system was optimized to exploit C64 hardware features [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 parameters, during the translation. We ran the translated code with problem size S on C64 FAST simulator [6], and report the execution results in Figure 10.

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 [20], both customized for the C64 [5]. Figure 10 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.

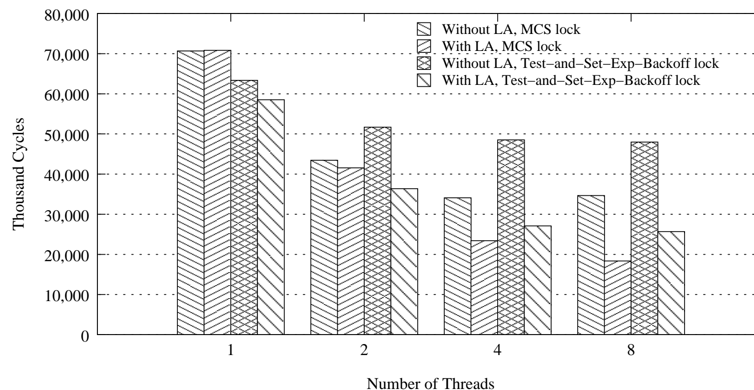


Figure 10: UA with/without Lock Assignment on C64 platform

6 Related Work

One of the key issues of locking technique in the area of concurrency control [11] 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 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 optimize it during the compilation time.

Transactional memory [15, 28, 24, 25, 14, 12, 3, 27] proposes a non-blocking synchronization mechanism. A transaction is a sequence of memory reads and writes executed by a single thread. Transactions are atomic and serializable. Programmers need to explicitly mark the transaction regions, and the system (either hardware or software) dynamically resolves the conflict. Distinguished from the transaction memory, critical sections are blocking synchronization mechanism, and our lock assignment is accomplised statically by the compiler. Our approach can also easily be extended to optimistic lock architecture with abort based on transactional memory [19].

There has 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) [4, 13, 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.

Vaziri et al. [32] introduced a data-centric approach for writing concurrent programs using atomic sets. Atomic sets are 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.

Diniz and Rinard [9] presents 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. Aldrich et.al. [1] reduce three forms of unnecessary synchronization from Java programs: thread-local synchronization, reentrant synchronization and enclosed lock synchronization. Our lock assignment and allocation are working with 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.

7 Conclusions

In this paper we proposed and solved two problems related to optimized assignment of locks to coarse-grained mutual exclusion, in the context of OpenMP programming model. 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 show that our methods can be used to improve the scalability of OpenMP programs by exploiting concurrency among multiple critical sections, compared to the default OpenMP implementation of using a single global lock to control all critical sections. Our proposed approaches are applicable to any other parallel programing model with mutual exclusion.

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. *Sci. Comput. Program.*, 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] 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, New York, NY, USA, 2005. ACM Press.

[5] 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.

[6] 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), in conjuction with the 32nd Annual International Symposium on Computer Architecture (ISCA2005)*, Madison, WI, June 2005.

[7] 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.

[8] Monty Denneau. Computing at the speed of life: The BlueGene/Cyclops supercomputer, Sep 25th, 2002.

[9] 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.

[10] Huiyu Feng, Rob F. Van der Wijingaart, 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.

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

[12] 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.

[13] 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, New York, NY, USA, 2003. ACM Press.

[14] M. Herlihy, V. Luchangco, M. Moir, and W.N. Scherer III. Software transactional memory for dynamic-sized data structures. In *the 22nd Annual ACM Symbosium on Principles of Distributed Computing (PODC'03)*, pages 92–101, Boston, MA, July 2003.

[15] 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, New York, NY, USA, 1993. ACM Press.

[16] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6*. IEEE, 2001. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992) Open Group Technical Standard Base Specifications, Issue 6.

[17] Henry Jin. Personal communication, April 2006.

[18] Joseph Manzano, Yuan Zhang, and Guang R. Gao. P3i: The delaware programmability, productivity and proficiency inquiry. In *SE-HPCS'05: Second International Workshop on Software Engineering for High Performance Computing System Applications*, 2005.

[19] 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.

[20] 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.

[21] NASA Advanced Supercomputing Divsion. The NAS parallel benchmarks. In http://www.nas.nasa.gov/Software/NPB/.

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

[23] OpenMP Architecture Review Board. OpenMP application program interface. Technical Report 2.5, OpenMP Architecture Review Board, May 2005. In http://www.openmp.org/specs.

[24] 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.

[25] 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.

[26] Vivek Sarkar and Guang R. Gao. Analyzable atomic sections: Integrating fine-grained synchronization and weak consistency models for scalable parallelism. Technical Report CAPSL-TM-052, University of Delaware, Newark, DE, February 2004.

[27] 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.

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

[29] 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.

[30] Standard Performance Evaluation Corporation. SPEC OMP (OpenMP benchmark suite). In http://www.spec.org/omp/.

[31] Sun Microsystems. *The Java Language Specification, Third Edition*, 2005. In http://java.sun.com/docs/books/jls/.

[32] 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, New York, NY, USA, 2006. ACM Press.

[33] Chung C. Wang. An algorithm for the chromatic number of a graph. *J. ACM*, 21(3):385–391, 1974.

[34] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characteriation and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, New York, June 22–24 1995. ACM Press.

A The MLA Problem is NP-Complete

To prove the MLA problem is NP-complete, we first cast it into a decision problem that determines whether there exists a lock assignment solution for the given concurrency graph using at most k locks. In the following theorem we prove this decision problem is NP-complete.

Theorem A.1. *The MLA problem is NP-complete*

Proof. We first show that the MLA problem is NP. Given an interfering concurrency graph $G = (V, E)$ and k colors (locks), our certificate is a color set $Color(v)$ for each $v \in V$, so that $|\bigcup_{v \in V} Color(v)| = k$. The certificate algorithm checks each edge $e = (u, v) \in E$. If e is an interfering edge, then u and v should share some locks, i.e., $Color(u) \cap Color(v) \neq \emptyset$. Otherwise, they have two disjoint sets of locks $Color(u) \cap Color(v) = \emptyset$. This verification can be performed in polynomial time.

To prove the MLA problem is NP-hard, we simply construct its special case by restricting it to non-interfering critical sections. From the discussion in Section 2.2 we know that the non-interfering special case is equivalent to the graph coloring problem. Since the graph coloring problem is NP-hard, the general MLA problem is also NP-hard. □

B Proof of Theorem 3.1

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. □

C Proof of Theorem 3.2

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. □

D Proof of Theorem 3.3

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.

E Omni OpenMP Compiler (OOC)

Omni OpenMP Compiler (OOC) [22] is a source-to-source compiler that translates OpenMP Fortran or C/C++ code with directives to C programs with Omni runtime support. Figure 11 shows the infrastructure of the OOC. We implemented concurrency analysis, pointer analysis, lock assignment and allocation as parts of Omni's backend. The runtime library API provides implementations for 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 platform, with the help of scheduling and resource management part.

□

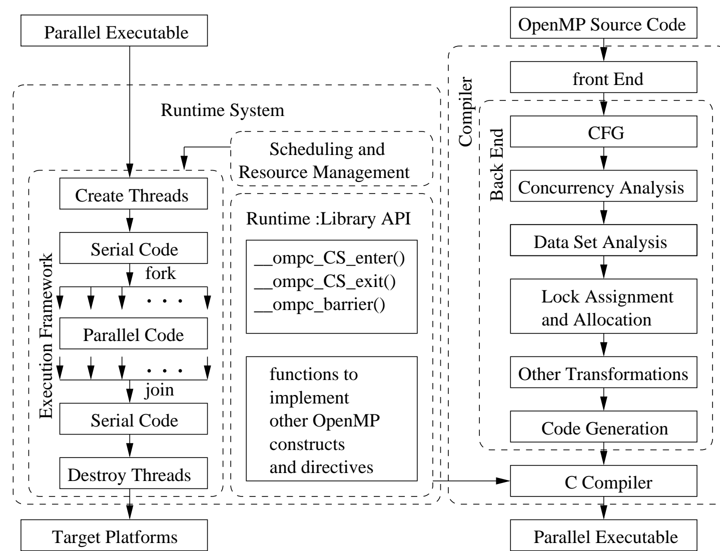


Figure 11: Omni OpenMP Compiler and Runtime System Infrastructure