vLock: Lock Virtualization Mechanism for Exploiting Fine-grained Parallelism in Graph Traversal Algorithms

Jie Yan[1,2] Guangming Tan[1] Xiuxia Zhang[1,2] Erlin Yao[1] Ninghui Sun[1]

[1]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
[2]University of Chinese Academy of Sciences
{yanjie, tgm, zhangxiuxia, yaoerlin, snh}@ict.ac.cn

For graph traversal applications, fine synchronization is required to exploit massive fine parallelism. However, in the conventional solution using fine-grained locks, locks themselves suffer huge memory cost as well as poor locality for inherent irregular access to vertices. In this paper, we propose a novel fine lock solution—-`vLock`. The key idea is lock virtualization that maps the huge logical lock space to a much smaller physical lock space that can reside in cache during the program life cycle. Lock virtualization effectively reduces lock incurred overheads of both memory cost and cache misses. It also achieves high usability in legacy graph programs, as from users's view `vLock` is the same as lock methods in Pthreads. We implement `vLock` as a Pthreads-like library and evaluate its performance in four classical graph algorithms (BFS,SSSP,CC,PageRank). Experiments on a SMP system with two Intel Westemere six-core processors show that, compared to conventional fine locks, `vLock` significantly reduces locks' cache misses and has competitive performance. Particularly, PageRank with `vLock` has about 20% performance improvement.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming–Parallel Programming

General Terms Algorithms, Performance

Keywords Graph Algorithms, Fine Synchronization, vLock

1. Introduction

Large-scale graph analysis has become an important procedure to mine valuable information in data-intensive applications such as web mining, social network analysis, bioinformatics, information retrieval and so on. Graph traversal problems are notoriously difficult to optimize due to their inherent irregular computation behaviors. First, graphs in real world are sparse and scale-free. Generally the adjacency of graph vertices is described by sparse matrix or list data structure. Access pattern of the sparse data structure is random so that locality is hard to be exploit. In real-world applications the operations of irregular memory access are intensive because the graph is composed of billions of vertices and edges. Second, the graph is difficult to partition for coarse-grained parallelism due to the irregular access pattern and data dependency. A consensus is that there is massive fine-grained parallelism during the course of graph traversal. However, we observe two problems in the fine-grained parallel implementations of graph algorithms.

- *Low efficiency:* In order to resolve conflicts of concurrent updates to the same vertex among parallel threads, most of parallel graph libraries (e.g., SNAP [6], etc.) use fine-grained locks for synchronization. However, in graph traversal applications, the *useful* work on vertices or edges concerns only some trivial operations, e.g. changing their states or accumulation. In [22] Tu et.al. profiled the execution of $SSCA\#2$ benchmark [5] where a scale-free graph is traversed to calculate each vertex's betweenness centrality. They showed the *useful* work of critical section is too small to amortize lock overhead. Thus, efficient thread synchronization mechanism is critical to performance of fine-grained parallel graph algorithms. One solution is architectural support such as word level Full-Empty Bits on Cray XMT [2] and SSB on IBM Cyclops64 [24]. However, such special architectural features are not available on commercial multi-core architectures for general-purposes. Therefore, other solutions to support fine-grained parallelism on multi-core resort to software optimizations including programming model and runtime system.
- *Low usability:* In order to overcome shortcomings of lock mechanism, most recent popular work focuses on lock-free algorithms [12, 18], transactional mem-

CGO '13 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

ory [14, 15] and optimistic parallelism [16]. These approaches dedicate to speculatively eliminate redundant synchronization, expecting higher performance. However, we note that it is not easy to adopt these approaches in graph traversal. For example, generally a lock-free algorithm is difficult to be developed and reasoned for correctness. Besides, programming for a lock-free algorithm is much more complicated than a conventional lock-based algorithm. Conceptually, to some extent both transactional memory and optimistic parallelism simplify parallel programming by avoiding explicit lock synchronization. In order to effectively support thread-level speculative execution, these approaches usually require specific hardware modification to existed architecture [13], and thus require manual work or special compiling efforts to match the hardware. In fact, if we port a parallel software to these new programming models, there are still a lot of legacy codes to be modified significantly. Obviously, the low usability limits their popularity.

We attempt to seek for an approach to balancing between performance and programmability for developing efficient fine-grained parallel graph programs on multi-core architectures. By investigating various lock-based parallel graph algorithms, we identify that source of inefficiency is the irregular memory access pattern. Most fine-grained parallel graph programs associate each vertex with a lock to resolve conflicts so that lock operations result in amounts of extra irregular memory accesses. Based on an important observation that the proportion of true conflict vertices is small at each single level of graph traversal, we propose a lock virtualization mechanism to reduce the number of irregular memory accesses while providing the same API with Pthreads. As in the original fine-grained parallel program, each vertex is associated with a lock. But, the difference is that the lock is just a *logical* one. The lock virtualization mechanism maps the huge *logical* lock space to a much smaller *physical* lock space which is small enough to be resident in cache during the program life cycle. Because the virtualization is a software mechanism, the cost of mapping strategy should be as little as possible. With respect to speed advantage of hash, we leverage a hash method to implement the mapping from logical lock space to physical one. Specifically, the main contributions of this paper include:

- We design and implement a lock virtualization mechanism `vLock` to develop fine-grained parallel graph programs. It consists of a lock virtualization layer and an underlying spin lock array. By virtualization, `vLock` can provide millions or even billions of fine-grained locks on current shared memory multi-core platforms, with much smaller cost of memory and higher cache performance than conventional fine-grained locks.
- We provide a set of Pthreads-style APIs for the use of `vLock`. With the interface, users can easily port a fine-grained Pthreads-style parallel graph algorithms without any change of the original program structure.
- We prove that a small and constant number of physical lock entries could ensure low probability of lock races, regardless of the graph scales.
- We evaluate `vLock` with four typical graph traversal algorithms(BFS, CC, SSSP, PageRank) on multi-core processors. `vLock` improves performance than the corresponding Pthreads-based program by 5% on average. Particularly, for PageRank, `vLock` outperforms by 20%.

In the rest of this paper, we first introduce background of the fine-grained parallel graph algorithm and outline our motivations in section 2. In section 3 we propose our lock virtualization methodology and its implementation. Section 4 presents a formal analysis of `vLock`. Section 5 reports our experimental results. The related work is presented in section 6. Finally, we conclude in section 7.

2. Background and Motivation

Traversal pattern is able to efficiently describe a broad set of graph algorithms, including path exploration(e.g., search, shortest path, betweenness centrality) and iterative algorithms(e.g., pagerank, connected component). In graph traversal applications, vertices and edges are explored in some order. Usually a traversal starts from a set of source vertices, and then computes and activates the neighboring vertices recursively. For a vertex v, computation on it could be triggered by any incoming edge, and changes to v's state would trigger further computations on vertices connected by v's outgoing edges. In parallel graph traversal, generalized breadth-first-search (BFS) is often adopted as a fundamental framework, as depicted in Algorithm 1.

Algorithm 1: Breadth-first Graph Traversal Framework

```
   Input  : S: set of source vertices
   Input  : G: graph=(V, E)
1  Q: Queue of active vertices in current phase;
2  Q': Queue of active vertices in next phase;
3  f(V): Computation on vertex;
4  initialize the states of S;
5  Q.enqueue(S);
6  while Q ≠ ∅ do
7      while Q ≠ ∅ in parallel do
8          u ← Q.dequeue();
9          foreach u's outgoing edge (u, v) do
10             lock();
11             do some computation f(v);
12             unlock();
13             Q'.enqueue(v);
14     Q ← Q';
15     Q'.clear();
```

2.1 Fine-grained Parallelism

In graph traversal, once a vertex v's state is changed over a threshold, all of its outgoing edges are activated. Generally, during the course, there are massive edges ready to handle

and multiple ready edges may compute on the same vertex. For example, during BFS on a typical real world graph, there are millions of concurrent edges or vertices to be processed in middle phases of traversal. Note that the computational workload is lightweight and usually involves only a few simple arithmetic operations (i.e., accumulation, comparison,etc.). It means that the potential parallelism is massively fine-grained.

In multi-threading parallel mode, state of a vertex may be concurrently updated through several incoming edges by multiple threads, which requires synchronization to ensure its consistency. In order to exploit the massive fine parallelism, it is needed to adopt fine synchronization. To demonstrate this point, we consider two synchronization schemes implementing `lock` and `unlock` in Algorithm 1: `CoarseLock` that associates all vertices with one lock and `FineLock` that associates each vertex with one lock. We adopted multi-threaded BFS of Graph500 [1] and evaluated on a SMP machine with two Intel Westmere 6-core processors. As shown in Figure 1 (left), `CoarseLock` is not scalable, whose speedup is less than 3 on 12 cores, while `FineLock` achieves higher scalability, whose performance is 3 times better than `CoarseLock` on 12 threads. It is quite intuitive that fine-grained lock enables higher degree of runtime parallelism has better scalability. T flict rate shown in Fig of `CoarseLock` rise threads and reaches mo curve for `FineLock` l

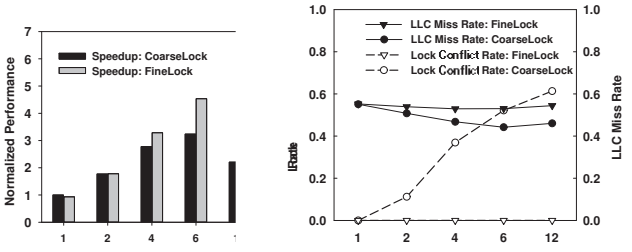


Figure 1. Results of BFS on the graph of 2^{24} vertices with 2^{28} edges, where x-axis represents number of threads. (Left): Performance, normalized to one thread with CoarseLock; (Right): L3 cache(LLC) miss rate and lock conflict rate.

2.2 Motivation

In fact, the previous work of parallel graph algorithms on shared memory architectures have already adopted the fine-grained lock synchronization. However, conventional fined-grained lock scheme still suffers two crucial problems.

First, in fine-grained lock scheme, each vertex is associated with a lock, and thus the memory space (storage) for locks is proportional to the number of vertices. For a typical real world graph with more than hundreds of millions vertices, memory cost for locks is too large, especially for out-of-core algorithms. As we know, the size of graph data (vertices and edges) itself is too large to be resident in memory space on most commodity computer system, the extra space cost of locks makes this problem worse. Although there are alternative hardware solutions like word-level full/empty bits attached to each memory words [2, 24], they are too expensive to be widely adopted.

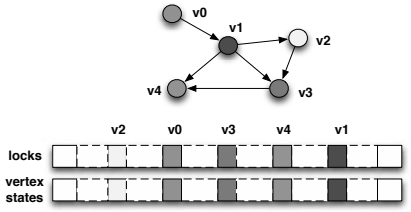


Figure 2. An illustration of memory access pattern.

Second, there is little cache locality for accessing the lock memory space. Figure 2 illustrates BFS on part of a graph, where the program starts from vertex v_1 to explore its neighbors: $\{v_2, v_3, v_4\}$. Data of these vertices are scattered in the memory region. As access to vertex states, access to locks is also random and noncontinuous. The irregular access pattern in lock memory region results in extra cache misses. Consider the BFS example again. As shown in Figure 1(right figure), compared to coarse-grained lock, fine-grained lock has significantly higher L3 cache miss rate and the gap has a widening trend over increasing threads.

However, further investigation reveals an important observation: *in graph traversal, the rate of runtime lock conflicts is very low.* For example, in experiments of Figure 1, even for 12 threads the lock conflict rate is as low as in an order of 10^{-6}. That means, most of lock operations are actually redundant and thus it is not necessary to associate each vertex with a single lock. Obviously, this observation also supports feasibility of speculative or optimistic parallelism. As noted in the introduction, however, these approaches require major changes to the structure of the original parallel programs. To be on the high usability side, instead we propose a lock virtualization mechanism, `vLock`, that attempts to build a seamless dock to the legacy codes using fine-grained locks. In `vLock`, only a small number of physical lock entries are actually allocated, and virtualized to associate each vertex with a logical lock. Programmers think the logical locks as conventional fine-grained locks, and thus no need to change the structure of original codes.

3. Methodology

`vLock` is designed for parallel graph algorithms. Here we first define the following notations used in next sections.

- $\mathbb{O}$: object space,
- $\mathbb{V}$: vertex space or virtual lock space,
- $\mathbb{L}$: physical lock space,
- h: $\mathbb{V} \rightarrow \mathbb{L}$, return the lock of a specified vertex or address,
- $addr$: $\mathbb{O} \rightarrow \mathbb{A}$, return the address of a specified object,

- f: $\mathbb{V} \rightarrow *$, do computation with updates on an object.

For clearness of further description, we also define the scenarios where vLock is used as follows. Assume $\mathbb{O}$ is the set of objects. Each object $o \in \mathbb{O}$ is strictly associated with a vertex v and can be indexed by v' unique id, i.e., o serves as a property of vertex v[1]. Physically, $\mathbb{O}$ is stored in an array. In order to support concurrent operations on objects, each $o \in \mathbb{O}$ is associated to a lock in $\mathbb{V}$.

3.1 Virtualization Mechanism

Th essential idea of vLock is lock virtualization, that is logically each vertex is associated with a virtually exclusive lock while physically multiple vertices may share the same lock. Formally, we build a physical lock space, $\mathbb{L}$, where $|\mathbb{L}|$ is much smaller than $|\mathbb{V}|$ (or $|\mathbb{O}|$). Virtual lock space is mapped to $\mathbb{L}$ by some mapping function h. Figure 3 illustrates the mechanism of vLock, where each set of virtual (logical) locks sharing the same color are mapped to one physical lock with that color. By virtualization, the huge virtual lock space is projected to such a small physical lock space that can be kept in cache.

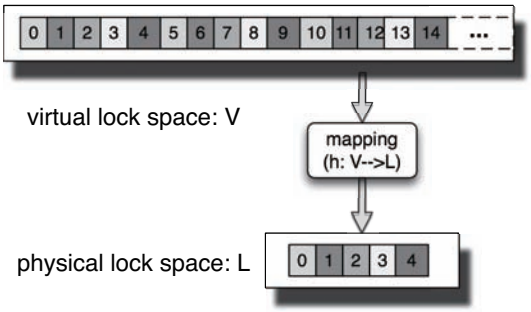


Figure 3. An illustration of lock virtualization mechanism.

The above virtualization mechanism is transparent to users, i.e., in the view of programmers, every vertex has an exclusive lock. Thus, vLock promises the same programmability as traditional fine-grained locks. Generally, vLock provides a minimal set of primitives shown in Figure 4. The interface of vLock and semantics of its primitives are the same as their counterparts in traditional fine locks. [2]

Compared to traditional fine-grained locks, vLock significantly reduces the memory space of locks. For example, given a graph with 2^{25} vertices, vLock needs only 8K entries of physical locks that cost 32KB memory, while the traditional fine-grained locks need 128MB memory.

Obviously, if the size of vLock ($|\mathbb{L}|$) can be reduced to be small enough, vLock has better cache performance. However, reducing $|\mathbb{L}|$ risks increasing probability of lock races. Fortunately, in graph traversal algorithms, the scale free property of real world graphs makes the pattern of access to vertices random, which further ensures that a small $|\mathbb{L}|$ suffices to keep probability of lock conflicts low enough. This fact is the most important basis of vLock. It shall be theoretically proved in section 4 and experimentally verified in section 5.3. Another disadvantage of vLock is the overhead of mapping approach in virtualization. In practice, we adopt two simple but fairly efficient hash methods which will be described later.

[1] Similarly, $\mathbb{O}$ can be associated with edges and uniquely indexed by edge id. In this paper, we don't concern this case.

[2] Note that the side effect of primitive $trylock$ is different from that in traditional fine-grained locks. In traditional fine locks, returning a failure value implies some other thread is operating on v while in vLock it doesn't.

3.2 Implementation

3.2.1 API

In order to facilitate seamlessly portting the previous programs in Pthreads, vLock adopts the similar application programming interface (API) with Pthreads library. Figure 4 lists the minimal basic lock functions in vLock library.

```
pthread_vlockattr_init(pthread_vlockattr_t* att, int size);
pthread_vlockattr_destroy(pthread_vlockattr_t* att);
pthread_vlock_init(pthread_vlock_t* lockid,
                   pthread_vlockattr_t *att);
pthread_vlock_destroy(pthread_vlock_t* lockid);
pthread_vlock_lock(ulong v, pthread_vlock_t lockid);
pthread_vlock_unlock(ulong v, pthread_vlock_t lockid);
pthread_vlock_trylock(ulong v, pthread_vlock_t lockid);
```

Figure 4. A minimal set of vLock API.

The usage of vLock is similar to the corresponding lock methods in Pthreads, such as pthread_mutex_t and pthread_spinlock_t . For example, the code shown in Figure 5 supports concurrent computation f on object o that is associated to vertex v. Pthreads lock methods directly operate on the allocated physical lock (i.e., mutex). In our virtualization approach, the APIs "operate" a virtual lock whose id is a long integer. The virtual lock id would use either a vertex id or the address of an object, which can be set by pthread_vlockattr_init function. In the simple example, we used the default option, i.e., a vertex id based lock virtualization.

```
pthread_mutex_init(&lock, NULL);
pthread_mutex_lock(&lock);
f(o);
pthread_mutex_unlock(&lock);
```

(a) Mutex lock

```
pthread_vlock_init(&lockid, NULL);
pthread_vlock_lock(v, lockid);
f(o);
pthread_vlock_unlock(v, lockid);
```

(b) vLock

Figure 5. An example of using vLock.

Note that vLock itself doesn't support general usage of nested locks. For the existence of physical lock sharing, nested usage may lead to deadlock. For example, consider such a situation where objects $a, b \in \mathbb{O}$ and $h(a) = h(b)$. Such a nested usage is an obvious deadlock:

```
pthread_vlock_lock(a, 0); pthread_vlock_lock(b, 0);
pthread_vlock_unlock(b, 0); pthread_vlock_unlock(a, 0);
```

However, vLock supports a special case of nested usage that is enough for graph traversal algorithms. Suppose $\mathbb{O}_1$ and $\mathbb{O}_2$ are two disjoint sets of objects, which are mapped to two different sets of vLocks with $h_1 : \mathbb{O}_1 \rightarrow \mathbb{L}_1$ and $h_2 : \mathbb{O}_2 \rightarrow \mathbb{L}_2$ respectively, then such nested usage is safe:

```
pthread_vlock_lock(a, 1); pthread_vlock_lock(b, 2);
pthread_vlock_unlock(b, 2); pthread_vlock_unlock(a, 1);
```

3.2.2 Hash Mapping

In an implementation of lock virtualization in Figure 3, the mapping function h plays a key role. In order to keep overhead of lock virtualization low enough, `vLock` uses a simple $module$ operation to perform hash mapping. With a tradeoff between performance and generality, we provide two hash strategies: hash by vertex id and hash by object address.

Hash by vertex id is the default mapping method in `vLock`. Generally, each vertex in graph is indexed by an unique id of long integer. Any object is associated to a vertex, so the virtual lock of an object can be uniquely indexed by the vertex id. Thus, it is convenient to map virtual locks to physical locks by vertex id. Because the vertex indices are continuous integers, i.e., {0,1,2,...}, a hash function using $module$ computation always maps vertices to physical locks uniformly. Let the size of physical lock space ($|\mathbb{L}|$) be power of 2, then the $module$ operation can be implemented in the *bit and* operation, i.e., $h(v) = v \ \& \ (|\mathbb{L}| - 1)$. This is fast enough so that in practice its overhead can be neglected.

Hash by object address is a more general method, and it is suitable for objects that can not be indexed by IDs. In a shared memory multithreading application, the address space is linear and any object can be uniquely identified by its beginning address. In current mainstream systems, address value is an unsigned integer of 64 bits. We denote the address space as $\mathbb{A}$ and operation $addr : \mathbb{O} \rightarrow \mathbb{A}$ returns the address of an object. Further, we define a new hash function $h : \mathbb{A} \rightarrow \mathbb{L}$. `vLock` adopts the hash function $h(addr(o)) = addr(o) \ mod \ |\mathbb{L}|$, where $|\mathbb{L}|$ must be prime. In section 4.2, we shall prove that for a set of objects in array, the above hash function maps their virtual locks to physical locks uniformly.

The computation in hash by vertex id is an `AND` instruction of 64-bit integer, while in hash by object address it is an `DIV` instruction of 64-bit integer. Take the processor(Intel Xeon X5650) of our system as an example, the `AND` on 64-bit integer costs 1 cycle, while the `DIV` on 64-bit integer costs 70∼80 cycles [10]. Compared to hash by vertex id, hash by object address has more overhead of hash computation. For common graph traversal algorithms, both methods work well, while for algorithms that intensively request locks, such as PageRank, overhead of hash by address becomes significant. In the experimental evaluation, we shall show performance of both methods.

3.2.3 Internals

`vLock` consists of two parts, the lock virtualization layer and underlying lock implementation. An implementation of `vLock` is decided by three parameters: the hash function h, the lock space size $|\mathbb{L}|$, and the underlying implementation of locks. In our specific implementation of `vLock` for graph traversal, we made the following key technical choices.

- In default, we adopt hash by vertex id, while hash by object address can be configured by users.
- $|\mathbb{L}|$ is set as a constant value 8192 in default for our experimental system, no matter the scale of graph. In 4.1, we will give a theoretic proof that given the pattern of random access to vertices, when the scale of vertex set is large enough, the lock conflict rate no longer relies on the number of vertices but the size of $|\mathbb{L}|$ and the number of parallel threads which is at most the number of processor cores. For a given system whose number of cores is constant, to keep rate of lock conflicts under a threshhold, the minimal value of $|\mathbb{L}|$ can be decided. For a system with less than 16 cores, lock size of 8192 can ensure conflict rate under 1.4% in the worst case.
- The internal physical lock adopts CAS-based spin lock. For the fine granularity of vertex computation, punishment of waiting for a busy lock is trivial. For this reason, we adopt spin lock as the underlying implementation. In fact, as a comparison, both mutex of Pthreads which concerns potential system call cost and lightweight ticket lock can't achieve performance comparable to spin lock.

4. Theoretical Analysis

Compared to conventional fine-grained lock methods, `vLock` has the advantages of (1)less memory cost and (2)better cache performance of locks, as well as the disadvantage of (3)risks on punishment of extra lock conflicts. In this section, we give a theoretical proof to the conclusion that: given random access to large scale of virtual locks, there exist a proper physical lock space size and a good hash function so that the lock conflict rate can keep very low. Besides, we prove that the hash functions used in `vLock` ensure uniform distribution of objects(virtual locks) to physical lock space.

4.1 Lock Conflicts

In `vLock`, *false lock conflict* occurs when multiple different objects share and concurrently request the same lock, which never happens in conventional fine-grained locks.

THEOREM 4.1. *Assume all accesses to objects are uniformly random, then probability of false lock conflict is at most:* $\frac{A(|\mathbb{O}|,t)}{|\mathbb{O}|^t} - \frac{A(|\mathbb{L}|,t)}{|\mathbb{L}|^t}$ *,where t is the number of parallel threads and* $A(n,k)$ *is the number of different arrangements of k from n items.*

Proof. At any time point, for t threads that access elements of $\mathbb{O}$ independently, the number of all possible different cases is $|\mathbb{O}|^t$, while the number of cases that have no conflicts is $A(|\mathbb{O}|,t)$. Thus, the probability of non-conflicting access to $\mathbb{O}$ is $\frac{A(|\mathbb{O}|,t)}{|\mathbb{O}|^t}$, and further the probability of conflict access to $\mathbb{O}$ is $1 - \frac{A(|\mathbb{O}|,t)}{|\mathbb{O}|^t}$.

Meanwhile, as $\mathbb{O}$ is proportionally mapped to $\mathbb{L}$, access to $\mathbb{L}$ can be considered as uniformly random. Similarly, we thus conclude that the probability of non-conflicting access to $\mathbb{L}$ is $\frac{A(|\mathbb{L}|,t)}{|\mathbb{L}|^t}$, and further the probability of conflict access to $\mathbb{L}$ is $1 - \frac{A(|\mathbb{L}|,t)}{|\mathbb{L}|^t}$.

A false lock conflict means a conflict occurring on locks but meanwhile not on objects. Because in the hash mapping relation $h : \mathbb{O} \to \mathbb{L}$, $|\mathbb{L}| < |\mathbb{O}|$ always holds, a conflict access to $\mathbb{O}$ must induce a conflict on $\mathbb{L}$ but not vise versa.

To sum up, combine the above two partial conclusions, we get the probability of locks' false conflicts: $(1-\frac{A(|\mathbb{L}|,t)}{|\mathbb{L}|^t})-(1-\frac{A(|\mathbb{O}|,t)}{|\mathbb{O}|^t}) = \frac{A(|\mathbb{O}|,t)}{|\mathbb{O}|^t} - \frac{A(|\mathbb{L}|,t)}{|\mathbb{L}|^t}$. □

In reality, $|\mathbb{O}|$ is very large (e.g., greater than 2^{24}) so that $\frac{A(|\mathbb{O}|,t)}{|\mathbb{O}|^t}$ is tightly close to and can be simply treated as 1. We then have the following lemma.

LEMMA 4.2. *If space of objects is large enough, then*
(1) probability of false lock conflicts is approximately equal to that of overall lock conflicts, i.e., $\frac{A(|\mathbb{O}|,t)}{|\mathbb{O}|^t} - \frac{A(|\mathbb{L}|,t)}{|\mathbb{L}|^t} = 1 - \frac{A(|\mathbb{L}|,t)}{|\mathbb{L}|^t}$.
(2) lock conflict probability is no longer relevant to the size of object space but only the size of locks ($|\mathbb{L}|$*) and the number of threads (*t*).*

Lemma 4.2-(2) is one of the theoretical foundations of vLock. It reveals that for a system that has a given number of hardware threads (t in the above analysis), we can choose a constant value for lock space size to make sure the rate of lock conflicts low enough, no matter how large scale of the object space $\mathbb{O}$ would be.

Figure 6 shows the curve plotted according to the formula in Lemma 4.2-(1). We can see that for given number of parallel threads, with increasing lock space size, the conflict rate falls quickly. For today's typical multi-core processors whose number of cores(or equivalently the maximum number of parallel threads) is less than 32, 8192 (or 8191) is a nearly perfect value of lock size that can avoid most conflicts(e.g., conflict rate is lower than 5.88% for 32 threads and lower than 1.45% for 16 threads) while keep memory cost small enough. Actually, 8192 is the default value in our experiments. Note that the above analysis assumes that all threads issue lock requests simultaneously, which is more harsh than real situations where lock requests are much less intensive and thus has a lower conflict rate.

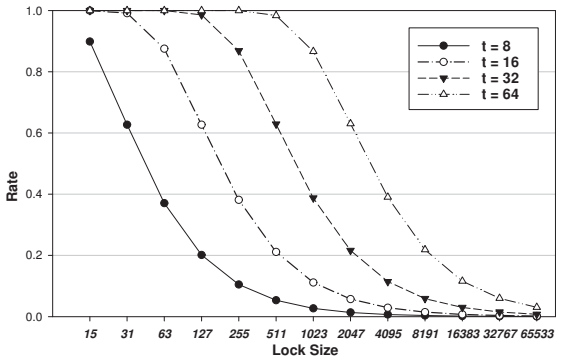


Figure 6. Conflict probability, plot by the formula in Lemma 4.2-(1). In the figure, t is the number of threads.

4.2 Hash Function

A good hash method should uniformly map virtual locks to physical locks. In vLock, object space is far larger than physical lock space, i.e., $|\mathbb{O}| \gg |\mathbb{L}|$. In order to reduce lock races, the distribution of objects mapped to each lock should be closely uniform.

For hash by vertex id where the vertex id set consists of continuous integers, it is straightforward that a $module$ function maps virtual locks to physical locks uniformly. For hash by object address, however, it needs more concerns. Here we only consider the object set implemented as an array, and the size of object element in memory is fixed. Unlike integer index, memory addresses of a set of objects are not continuous but incremental with a fixed stride. For example, size of object in type integer is 4 bytes while that of long type is 8 bytes. We here proved that with a dedicated hash function, vLock can automatically handle variance of object sizes and map objects uniformly to lock space.

THEOREM 4.3. *Given (1) any object type whose size is fixed as n bytes and (2) a lock space $\mathbb{L}$, where $|\mathbb{L}|$ is a prime number and $|\mathbb{L}| \gg n$, hash function:*

$$h(addr(o)) = addr(o)\ mod\ |\mathbb{L}|, where\ o \in \mathbb{O}$$

maps objects in $\mathbb{O}$ uniformly to the lock space $\mathbb{L}$, no matter the value of n.

Proof. Assume the base address of $\mathbb{O}$ is b and index of o in $\mathbb{O}$ is k, then $addr(\mathbb{O}[k]) = b + k * n$. Further,

$$\begin{aligned} h(addr(o)) &= (b + k * n)\ mod\ |\mathbb{L}| \\ &= c + k * n\ mod\ |\mathbb{L}| \\ &= c + X_k \end{aligned}$$

, where $c = b\ mod\ |\mathbb{L}|$ and $X_k = k*n\ mod\ |\mathbb{L}|$. Note that X_k is a circular series as k is incremental one by one and $n * |\mathbb{L}|$ must be one of its periods. Since $|\mathbb{L}|$ is a prime number, the period of X_k is $|\mathbb{L}|$, which means the distribution of $o \in \mathbb{O}$ on $\mathbb{L}$ is uniform. □

5. Evaluation

In this section, we have two objectives: (1) to verify vLock's advantage on performance over traditional fine-grained locks, and (2) experimentally explain why and how vLock works well in graph traversal algorithms.

5.1 Experimental Setup

Our benchmark consists of four classic graph algorithms: Breadth First Search (BFS), Single Source Shortest Path (SSSP) [7, 19], Connected Component (CC) [20], and PageRank [8]. All of them are implemented by the graph traversal framework in Algorithm 1. All graphs used in this section are scale free and generated by the R-MAT [9] generator in Graph500 [1]. The graph scale is 2^{24} vertices with

2^{28} edges for PageRank, and 2^{25} vertices with 2^{29} edges for other three algorithms.

Our experimental platform is a shared memory multi-core system whose configuration is shown in Table 1.

Table 1. Experimental Platform Configuration

Node	SMP
# of processors	2
Memory size	24GB
Processor	Intel Xeon X5650
# of cores	6
Frequency	2.67GHz
L1 Cache size	384K
L2 Cache size	1536K
L3 Cache size	12M
Memory Type	DDR3-1333
QPI speed	6.4GB/s
Operating System	CentOS 5.5
Compiler	GCC 4.1.2

The performance evaluation is performed with three sets of fine-grained parallel programs:

- `FineLock`: Using conventional fine-grained locks for synchronizing current updates on vertices, i.e., each vertex is associated with a single spinlock, as in most multi-threaded graph libraries.
- `vLock-Vtx`: Using our `vLock` library configured with hash-by-vertex-id.
- `vLock-Add`: Using our `vLock` library configured with hash-by-vertex-$address$.

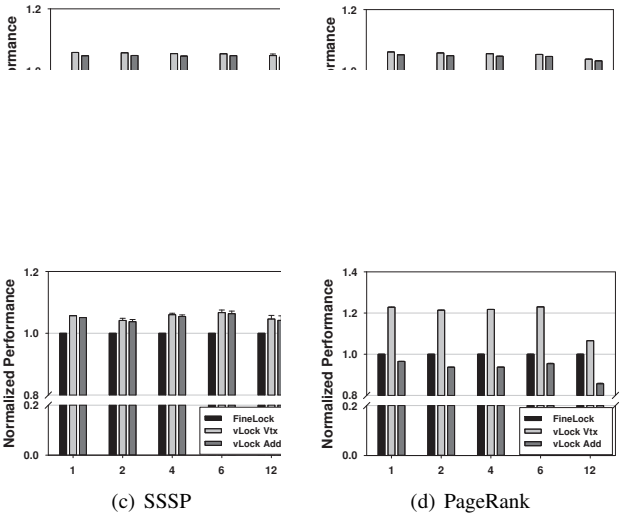
Both `vLock-Vtx` and `vLock-Add` adopt the default physical lock space size, 2^{13}, while `Finelock` can be considered as a special case of `vLock-Vtx` where physical lock space size is number of vertices. All raw experimental results are collected from 16 runs. Specifically, every time BFS and SSSP begin from a different source vertex, while CC and PageRank always begin iterating from all vertices.

5.2 Performance Analysis

We use the four benchmark programs to evaluate the performance of `vLock` and `FineLock`, and further analyze with detailed profiling data of lock conflicts and cache misses.

Figure 7 shows results of normalized performance of `vLock` as well as `FineLock`, where `FineLock` serves as baseline.[3] Overall, `vLock` outperforms `FineLock`. For BFS, CC and SSSP, `vLock` has an advantage of 4%∼8% from 1 to 12 threads. Particularly, when all threads run on the same socket (shown in Figure 7 as number of threads is no more than 6), PageRank of `vLock` shows interesting results that `vLock-Vtx` has considerable advantage of about 20% while its counterpart of `vLock-Add` degrades performance. The lost performance of `vLock-Add` is due to its high cost of hash computation which is 70∼80 times higher than that in `vLock-Vtx`. Meanwhile, lock requests in PageRank are much more frequent than others. Thus, in PageRank, overhead of hash by address is rather expensive and exceeds the benefits of lock virtualization itself.

Now we come to investigate the L3 cache(LLC) misses. As noted in the motivation section, `vLock` is expected to improve cache performance by reducing the size of physical lock space to be small enough to reside in cache. As shown in Figure 8, compared to the conventional `FineLock`, `vLock` significantly reduces number of LLC misses. The reduction of misses is due to the improved hit rate of lock accesses. Comparing Figure 8 and Figure 7, we found that LLC miss times are closely matched with performance for all four applications. Take PageRank as an example. When all threads are in one processor socket, LLC miss times of `vLock` is only about half of that in `FineLock`. Considered the expensive cost of LLC miss, it is not surprising that

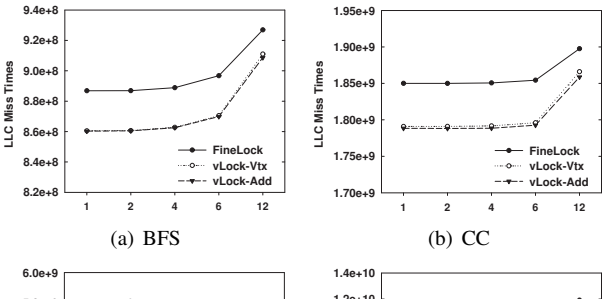


(c) SSSP (d) PageRank

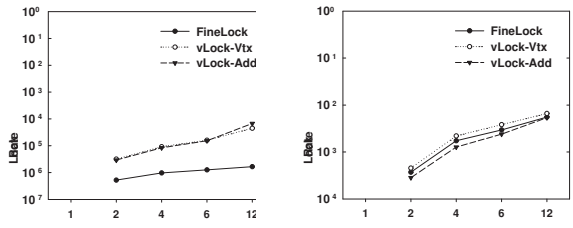
Figure 7. Comparison of performance normalized to case of FineLock. The x-axis represents the number of threads.

Besides, we know that compared to `FineLock`, the main disadvantage of `vLock` is the potential higher lock conflict rate. Figure 9 summarizes the rates of lock conflicts. For BFS, SSSP and CC, the conflict rates of `FineLock` are obviously lower than that of `vLock`. However, their absolute values keep very low (e.g., $< 0.01\%$ here), which is far smaller than the estimated bound by Lemma 4.2-(1). This means the overall overhead of lock races is low, and thus the performance difference between `vLock` and `FineLock` is primarily mainly decided by the difference of LLC misses, which is demonstrated by the accordance of changes between LLC miss times and performance.

[3] For BFS and SSSP, normalized performance of each run is first calculated and then used to compute their harmonic mean and deviation. For CC and PageRank, however, the mean and standard deviation of runtime in all 16 runs are first calculated and then handled with normalization.



(a) BFS (b) CC



(c) SSSP (d) PageRank

Figure 9. Comparison of lock conflict rates. The x-axis represents the number of threads.

The final question is why `vLock-Vtx` can improve performance of PageRank so much greater than other three benchmarks? The answer is the different frequency of lock requests. For BFS, SSSP and CC, not all edges trigger computation on their target vertices. For example, in BFS, an edge (u, v) triggers lock request and computation on v only when v is not visited. Similarly, in SSSP and CC, a comparison before lock requesting filters most of the redundant requests from edges. Thus, for these three benchmarks the frequency of lock requests is comparatively low. However, in PageRank there are no such filters to locks and each edge triggers a vertex computation. Therefore, frequency of lock requests is much higher than the others, which makes performance of the lock operation itself critical. On one hand, as shown in Figure 9, for both `FineLock` and `vLock`, lock conflict rate in PageRank is higher than others by one order. However, the absolute value is still low enough($< 0.7\%$), which means the overhead of lock conflicts is still low. On the other hand, as described before, for PageRank `vLock` reduces the LLC miss times by a half, which leads to significant memory performance boost. To sum up, for `vLock`, benefits of cache performance improvement exceeds the increased overhead of comparatively higher lock conflicts, which lets `vLock-Vtx` beat `FineLock`.

In summary, `vLock` performs better than conventional `FineLock` in graph traversal algorithms, especially in the situations where concurrent lock requests are frequent.

5.3 Impact of `vLock` Parameters

In this section, we use the BFS benchmark to experimentally illustrate the characteristics of `vLock` and quantitatively verify its design foundation. Particularly, we investigate how different choices on physical lock space sizes and hash functions affect performance of `vLock` over different number of threads.

The results are shown in Figure 10. The x-axis represents size of physical lock space. For hash by vertex, x in X-axis means number of physical lock entries is 2^x, while for hash by object address it is $2^x - 1$.[4]

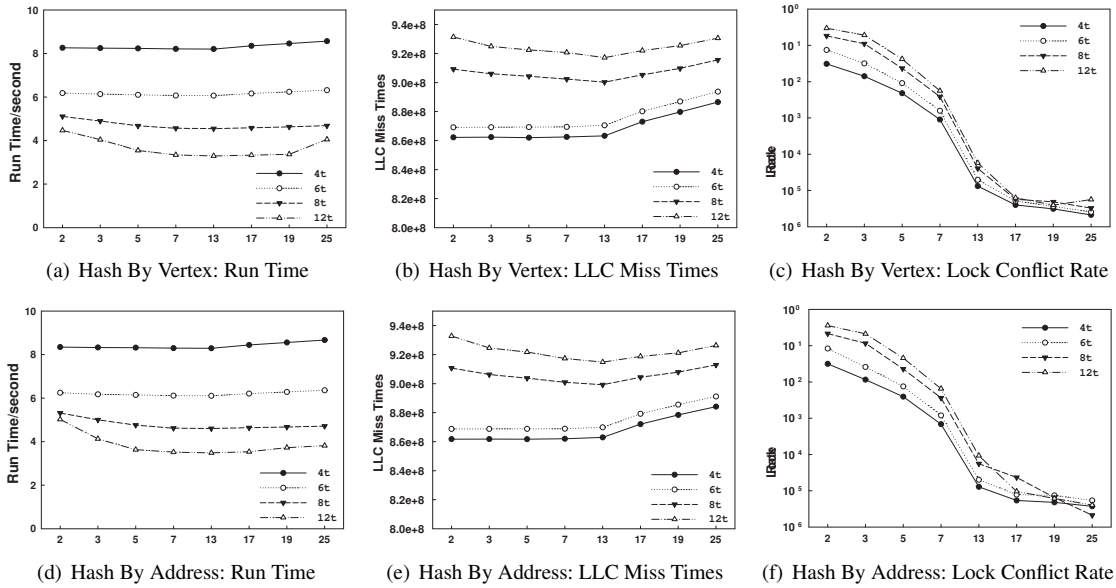
Fact 1: We first observe the execution time in Figure 10-(a) and 10-(d). For different number of threads and hash methods, the trends of run time are identical. With increasing lock entries, curve of run time first falls, achieves an optimal value, and then rises. This fact implies that with respect to performance both coarse-grained lock and fine-grained lock are not the optimal solution.

Fact 2: LLC misses, as shown in Figure 10-(b) and 10-(e), have the same trend with run time over increasing physical lock space size. When the number of lock entries is greater than 2^{13}, the LLC miss times increase nearly lineally. As the cost of LLC miss leads to expensive external memory access, it is a crucial factor affecting performance.

Fact 3: As shown in Figure 10-(c) and 10-(f), the lock conflict rate falls sharply with increasing physical lock entries. When the lock entries increase to more than 2^{13}, the lock conflict occurs with a probability of less than 0.01%. Obviously, lower lock conflict rate leads to higher degree of parallelism and further better performance. However, as shown, when the number of lock entries is more than 2^{17}, the curve becomes smooth, which means benefits of further increasing lock entries becomes trivial. Actually, as shown in Figure 10, the conflict rate of 2^{17} entries has been close to conventional fine locks (one lock per vertex, 2^{25} entries here) with only a gap of 0.001%.

Combine the above three facts, we can conclude that (1) when number of lock entries is small(e.g., $< 2^{13}$ in our experiments), increasing it leads to slow extra LLC cache miss

[4] The size of physical lock space should be a prime number for hash by address and be power of 2 for hash by vertex.



(a) Hash By Vertex: Run Time (b) Hash By Vertex: LLC Miss Times (c) Hash By Vertex: Lock Conflict Rate

(d) Hash By Address: Run Time (e) Hash By Address: LLC Miss Times (f) Hash By Address: Lock Conflict Rate

Figure 10. Results of `vLock` in BFS. The x-axis represents physical lock space size that is 2^x in (a)∼(c) and $2^x - 1$ in (d)∼(f).

rising and fast lock conflict falling, which improves performance, (2) when number of lock entries is big enough(e.g., $> 2^{13}$), further increasing it leads to fast extra LLC cache miss rising and slow lock conflict falling, which degrades performance, and (3) the best lock performance is achieved when the number of the lock entries(e.g., $> 2^{13}$) gets a balance on LLC miss and lock conflict rate. In practice, we only need to keep the lock conflict rate low enough(generally $< 0.1\%$) , and the nearly optimal value of physical lock space size can be estimated by Lemma 4.2-(2).

6. Related Work

There is a consensus that it is necessary to exploit fine-grained parallelism of applications in the era of multi-/many-core. The consequent question is how to support efficient fine-grained synchronization. In communities of both academics and industry there have been a lot of hardware and software attempts.

One class of hardware-based echnologies supports fine-grained data synchronization in memory [4, 11, 24]. Among these solutions, the most successful one is word-level Full-Empty Bits(FEB) implemented in Cray XMT [2] (previously MTA [11]) machines. In fact, Cray XMT performs excellent for graph algorithms thanks to the effective fine-grained data synchronization. However, the Cray solution requires a customized DIMM structure that is too expensive to afford for most users. Recently, on many-core architecture, Zhu et.al. [24] proposed a clever technique called Synchronization State Buffer (SSB). SSB was motivated by an observation, similar with ours, that at any instant only a small fraction of memory locations involve in data synchronization. Based on this observation, SSB adds a small piece of buffer to each bank of on-chip memory (i.e., SRAM-like fast memory), recording and managing states of frequently synchronized data. In other words, SSB uses a small "cache" to perform the similar function of FEB. Besides, as these fine-grained data synchronization mechanism needs to modify either memory or cache structure, it is not widely used in commodity machines.

As noted in the motivation section, our observation also supports feasibility of several emerging execution/programming models like speculative parallelism [21], optimistic parallelism [16] and transactional memory [14, 15]. Undoubtedly, these approaches can achieve bettter performance than Pthreads library more or less because of non-blocking synchronization with speculative execution. However, a common problem is that they require significant changes to the program structure when we port a Pthreads-based parallel program. Another efforts of software approaches primarily focus on designing lock-free algorithms [3, 12] and concurrent data structures [17, 18, 23] that leverage hardware atomic instructions. Most lock-free algorithms are designed for specific situations and need complex algorithmic proof to guarantee their correctness. In practice, lots of lock-free algorithms even perform worse than lock-based implementations. Actually, except for some common concurrent data structures such as queues [18], linked list [23] and hash table [17], lock-free approach is far from an alternative to lock synchronization.

`vLock` is a virtualization solution of the fine-grained lock synchronization on current commercial multi-core systems. Compared to the traditional fine-grained lock synchronization approach, it provides the same semantics and programmability while requires only a small amount of memory and ensures better cache performance on multi-core processors. Compared to the above hardware approaches, `vLock` provides the same synchronization functions without extra programming efforts or special hardware support.

7. Conclusion and Future Work

In this paper, we presented a novel solution of fine-grained locks—vLock, whose essential idea is lock virtualization. Given large size of lock space and random access to locks, compared to traditional fine-grained lock(e.g., spin lock in Pthreads), vLock only needs a small amount of memory cost and however achieves higher performance on multi-core platforms. We evaluated vLock with four classic graph traversal algorithms. As shown in experimental section, vLock outperforms conventional fine-grained lock methods with obvious advantage. Particularly, for applications with frequent lock requests, e.g., PageRank, vLock-Vtx shows more than 20% improvement on performance.

vLock is completely a software approach and provides the same programming style with the traditional lock methods in Pthreads. Unlike other new programming models, the adoption of vLock does not need to change program structure of the original Pthreads codes. Besides, vLock-Vtx is more specific to graph algorithms while vLock-Add is more general-purpose. However, vLock-Add's hash function incurs high cost when lock request is frequent. In the future, we shall investigate the possibility of hardware assisted implementation.

Acknowledgments

This work is supported by National 863 Program(2009AA 01A129), the National Natural Science Foundation of China (60803030, 61033009, 60921002, 60925009, 61003062) and 973 Program (2011CB302500 and 2011CB302502).

References

[1] Graph500 benchmark. *http://www.graph500.org*.

[2] Cray-XMT. *http://www.cray.com/products/xmt*.

[3] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proceedings of the twenty-seventh annual ACM Symposium on Theory of Computing*, STOC '95, 1995.

[4] B. E. Akgul and V. J. Mooney. The system-on-a-chip lock cache. *International Journal of Design Automation for Embedded System*, 7:139–174, 2002.

[5] D. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, W. Mann, and T. Meuse. HPCS scalable synthetic compact applications #2 graph analysis (ssca#2 v2.2 specification). September 2007.

[6] D. A. Bader and K. Madduri. SNAP, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *IPDPS*, pages 1–12, 2008.

[7] D. P. Bertsekas, F. Guerriero, and R. Musmanno. Parallel asynchronous label correcting methods for shortest paths. *Journal of Optimization Theory and Applications*, 89, 1996.

[8] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Computer networks and ISDN systems*, pages 107–117. Elsevier Science Publishers, 1998.

[9] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, 2004.

[10] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual. April 2012.

[11] J. Feo, D. Harper, S. Kahan, and P. Konecny. Eldorado. In *Proceedings of the 2nd Conference on Computing Frontiers*, CF '05, pages 28–34, New York, NY, USA, 2005.

[12] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), May 2007.

[13] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st annual International Symposium on Computer Architecture*, ISCA '04, 2004.

[14] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2010.

[15] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, 1993.

[16] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, 2007.

[17] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '02, 2002.

[18] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th annual ACM symposium on Principles of distributed computing*, PODC '96, 1996.

[19] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, 2010.

[20] Y. Shiloach and U. Vishkin. An o(log n) parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[21] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, 2000.

[22] D. Tu and G. Tan. Characterizing betweenness centrality algorithm on multi-core architectures. *International Symposium on Parallel and Distributed Processing with Applications*, 0: 182–189, 2009.

[23] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th annual ACM symposium on Principles of distributed computing*, PODC '95, 1995.

[24] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In ISCA '07, 2007.

vLock:图遍历算法中利用细粒度并行性的锁虚拟化机制严杰 1,2 谭光明 1 张秀霞 1,2 姚二林 1 孙宁晖 1
1 中国科学院计算技术研究所计算机体系结构国家重点实验室
2 中国科学院大学
yanjie,tgm,zhangxiuxia,yaoerlin,snh}@ict.ac.cn

对于图遍历应用，需要精细同步才能利用大规模精细并行性。然而，在使用细粒度锁的传统解决方案中，锁本身会承受巨大的内存成本，并且对于固有的不规则顶点访问，锁的局部性较差。在本文中，我们提出了一种新颖的细粒度锁解决方案 -vLock。关键思想是锁虚拟化，它将巨大的逻辑锁空间映射到可以在程序生命周期内驻留在缓存中的小得多的物理锁空间。锁虚拟化有效地减少了锁带来的内存成本和缓存未命中的开销。它还在旧式图形程序中实现了高可用性，因为从用户的角度来看，vLock 与 Pthreads 中的锁定方法相同。我们将 vLock 实现为类似 Pthreads 的库，并在四种经典图形算法（BFS，SSSP，CC，PageRank）中评估其性能。在具有两个 Intel Westemere 六核处理器的 SMP 系统上进行的实验表明，与传统的细锁相比，vLock 显着减少了锁缓存未命中，并且具有竞争力的性能。特别是，带有 vLock 的 PageRank 具有约 20％的性能提升。类别和主题描述符

D.1.3 [编程技术]：并发编程并行编程一般术语算法，性能关键词图算法，精细同步，vLock

1.简介大规模图分析已成为数据密集型应用中挖掘有价值信息的重要程序，例如网络挖掘，社交网络分析，生物信息学允许免费复制本作品的全部或部分用于个人或课堂使用，前提是复制或分发不是为了盈利或商业利益，并且副本在第一页上带有此通知和完整引文。否则，复制，重新发布，发布在服务器上或重新分发到列表，需要事先获得特定许可和/或付费。

CGO 13
2013 年 2 月 23-27 日，中国深圳。
978-1-4673-5525-4/13/$31.00 c 2013 IEEE.$15.00

信息学、信息检索等。图遍历问题由于其固有的不规则计算行为而难以优化。首先，现实世界中的图是稀疏和无标度的。通常，图顶点的邻接关系由稀疏矩阵或列表数据结构描述。稀疏数据结构的访问模式是随机的，因此很难利用局部性。在现实世界的应用中，由于图由数十亿个顶点和边组成，因此不规则内存访问的操作是密集的。其次，由于不规则的访问模式和数据依赖性，图很难划分为粗粒度并行。一个共识是，在图的过程中存在大量的细粒度并行性遍历。然而，我们观察到图算法的细粒度并行实现中存在两个问题。效率低下：为了解决并行线程间对同一顶点的并发更新冲突，大多数并行图库（如 SNAP [6]等）都使用细粒度锁进行同步。然而，在图遍历应用中，对顶点或边的有用工作仅涉及一些琐碎的操作，例如更改其状态或累积。在[22]中，Tu 等人分析了 SSCA#2 基准测试[5]的执行情况，其中遍历无标度图以计算每个顶点的中介中心性。他们表明临界区的有用工作太小，无法摊销锁开销。因此，高效的线程同步机制对于细粒度并行图算法的性能至关重要。一种解决方案是架构支持，例如 Cray XMT [2]上的字级全空位和 IBM Cyclops64 [24]上的 SSB。然而，这种特殊的架构特性在商用多核架构上不具备通用性。因此，支持多核细粒度并行性的其他解决方案都依赖于软件优化，包括编程模型和运行时系统。可用性低：为了克服锁机制，最近的热门研究集中在无锁算法[12,18]、事务内存[14,15]和乐观并行[16]上。这些方法致力于推测性地消除冗余同步，以期望获得更高的性能。然而，我们注意到在图遍历中采用这些方法并不容易。例如，通常难以开发和推理无锁算法的正确性。此外，无锁算法的编程比传统的基于锁的算法复杂得多。从概念上讲，事务内存和乐观并行在某种程度上都通过避免显式锁同步来简化并行编程。为了有效地支持线程级推测执行，这些方法通常需要对现有架构进行特定的硬件修改[13]，因此需要手动工作或特殊的编译工作来匹配硬件。事实上，如果我们将并行软件移植到这些新的编程模型上，仍然有大量遗留代码需要进行重大修改。显然，低可用性限制了它们的普及。我们试图寻求一种在性能和可编程性之间取得平衡的方法，以便在多核架构上开发高效的细