# pLock: A Fast Lock for Architectures with Explicit Inter-core Message Passing

Xiongchao Tang
Tsinghua University, Qatar Computing Research Institute
txc13@mails.tsinghua.edu.cn

Jidong Zhai
Tsinghua University, BNRist
zhaijidong@tsinghua.edu.cn

Xuehai Qian
University of Southern California
xuehai.qian@usc.edu

Wenguang Chen
Tsinghua University
cwg@tsinghua.edu.cn

## Abstract

Synchronization is a significant issue for multi-threaded programs. Mutex lock, as a classic solution, is widely used in legacy programs and is still popular for its intuition. The SW26010 architecture, deployed on the supercomputer Sunway Taihulight, introduces hardware-supported inter-core message passing mechanism and exposes explicit interfaces for developers to use its fast on-chip network. This emerging architectural feature brings both opportunities and challenges for mutex lock implementation. However, there is still no general lock mechanism optimized for architectures with this new feature.

In this paper, we propose pLock, a fast lock designed for architectures that support Explicit inter-core Message Passing (EMP). pLock uses partial cores as lock servers and leverages the fast on-chip network to implement high-performance mutual exclusive locks. We propose two new techniques – chaining lock and hierarchical lock – to reduce message count and mitigate network congestion.

We implement and evaluate pLock on an SW26010 processor. The experimental results show that our proposed techniques improve the performance of EMP-lock by up to 19.4× over a basic design.

***CCS Concepts*** • **Computer systems organization → Multicore architectures**; *Processors and memory architectures*; • **Software and its engineering → Multithreading**; **Mutual exclusion**; *Concurrency control.*

***Keywords*** lock; synchronization; on-chip network; inter-core message passing

**ACM Reference Format:**
Xiongchao Tang, Jidong Zhai, Xuehai Qian, and Wenguang Chen. 2019. pLock: A Fast Lock for Architectures with Explicit Inter-core Message Passing. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304030

## 1 Introduction

Mutual exclusive lock is a common mechanism of synchronization for multi-threaded programs. A lock protects a certain *critical section* (CS) and can only be held by a single thread at any given time. Only the thread which acquires the lock can execute the corresponding critical section. As a classic solution, lock is widely used in legacy programs and is popular for its intuition. Because of its importance, lock has been carefully designed and implemented for modern multi-core architectures [22, 27].

However, for emerging many-core processors, conventional coherent cache architecture has become more and more complex and it is very hard to achieve high performance [32]. A novel architectural feature, **E**xplicit inter-core **M**essage **P**assing (EMP), has gained popularity in research and even been used in some product many-core processors, such as TILE-Gx8036 [30] and SW26010 [13]. The Sunway TaihuLight [1] supercomputer is powered by SW26010 that uses EMP instead of coherent cache to share data among cores.

Since EMP provides both high bandwidth and low latency for inter-core communication, researchers have used EMP to accelerate programs on Sunway Taihulight [24, 38]. However, the widely used lock mechanism has not been well optimized for SW26010 architectures, which leads to sub-optimal performance for multi-threaded programs that frequently use locks to protect critical sections. Consequently, developers who want to port their multi-threaded programs to such new architectures with EMP support face a dilemma: they either need to rewrite their code using a new programming paradigm [25] or give up the opportunity to accelerate synchronization with EMP.

In this paper, we propose pLock, a new lock mechanism specially designed for EMP supported architectures, such as SW26010. pLock enables developers to leverage the benefits of fast EMP while preserving the conventional *lock and critical section* paradigm. We build pLock as a library that provides conventional *lock/unlock* interfaces and hides all complex architectural details. As a result, application developers can use pLock without knowing the underlying architectural features, and thus saves much time from porting and tuning.

Regarding thread synchronization, a key advantage of EMP over shared memory is that programmers can explicitly specify a core to share data with another. Therefore, high contention for a shared memory region can be efficiently avoided. The basic idea of an EMP-based lock is to use a dedicated core as a *lock-server*, and all the other cores as (*clients*) that can request locks from the lock-server [10]. Although less cores can be used for computation, the faster lock still improves the performance for lock-intensive programs. This server/client model seems to be similar to locks used in distributed systems. However, a distributed lock usually prioritizes reliability and availability [5], whereas on many-core processors, a lock mechanism designed for multi-threaded programs needs to focus on performance.

There are two main challenges to efficiently utilize EMP to implement a high-performance lock for Sunway Taihulight: (1) Though transferring data over EMP avoids memory contention, network congestion can become a new performance bottleneck with frequent communication. A fast lock based on EMP should minimize the number of messages among cores. (2) The inter-core network of SW26010 architecture is non-uniform and asymmetric. Thus the communication performance between different cores is different. As a result, we need to consider how an EMP-based lock works on a hierarchical network.

We propose pLock, a high-performance lock library that addresses the above two key problems. In previous EMP-lock design, clients need to communicate with lock server to request and release locks each time. However, this mechanism introduces a large amount of communication between a lock server and clients. Instead of sending requests to lock server each time, we *transfer lock* request among clients directly in pLock. Therefore, pLock speeds up the process of requesting and releasing locks, and reduces the communication traffic to mitigate network congestion. Furthermore, we adopt a hierarchical lock mechanism to improve the communication performance on a non-uniform network. We summarize our contributions as below:

- We propose novel chaining lock method, which allows a client to bypass lock server and transfer locks directly to another client. This design reduces the message amount thus mitigates the communication congestion caused by lock contention. To our best knowledge, chaining lock is a new and novel approach on EMP architectures.
- We propose a hierarchical lock method to fit the non-uniform nature of the inter-core network. This hierarchical method avoids unnecessary long-distance communications (which is slow) and replaces them with faster short-distance communications. Although hierarchical lock has been used in traditional shared-memory lock approaches, we are the first to adapt it on EMP architectures.
- Based on the proposed approach, we implement pLock on a product processor with EMP, *i.e.,* SW26010. We also evaluate our approach with micro-benchmarks and multi-threaded parallel programs.

The experiment results show that, compared with a basic design, pLock can reduce 53% of communication and reduce lock latency by 83%.

## 2 Background

### 2.1 EMP on SW26010 Architecture

On-chip inter-core network is not a new architectural feature and is already used by modern multi-core and many-core processors. However, on most current architectures, inter-core network hardware is invisible to developers. For example, the x86 architecture uses its inter-core network to transfer cache lines among cores for cache coherence. Both of the inter-core network and the cache are transparent in the x86 architecture. On the contrary, the SW26010 architecture exposes its inter-core network to developers for better architecture scalability.

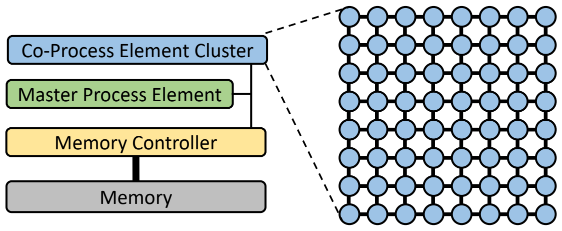


**Figure 1.** The architecture of SW26010.

SW26010 is a product processor that has been deployed on the supercomputer Sunway TaihuLight [13]. Figure 1 shows parts of its architectural features related to this work. The basic unit in SW26010 is a *core group (CG)*. A CG is composed of a *master processing element (MPE)* and a *co-processing element cluster (CPE cluster)*. While MPE is used for network communications and IO operations, most computation work is delivered to CPE cluster. A CPE cluster contains 64 CPE cores, which are connected by a 2D mesh inter-core network. Cores can use the following interfaces to communicate with others in the same row or the same column:

- `PUTR(destination, value)`. A core can send a message that contains *value* to another core in the same row, specified by *destination*. *Value* is 256 bits long, and developers can customize its content. We will show how to use the 256 bits for encoding messages in Section 4.
- `PUTC(destination, value)`. Similar to PUTR, but sends a message to the same column.
- `GETR(value)`. A core can receive a message from other cores in the same row, and then saves its content in *value*. The source core is unknown until the message is received. If no message is coming, the core will block and wait until there is a message.
- `GETC(value)`. Similar to GETR, but receives a message from the same column.

When the on-chip network is free, the four instructions above can complete within 13 CPU cycles, and the throughput is 1 instruction per cycle. This fast on-chip network enables the fast lock approach presented in this paper. Sunway Taihulight is mainly used for High-Performance Computing (HPC) programs. Most HPC programs pin each thread to a fixed core, and a core serves only a thread. In other words, there is a one-to-one mapping between cores and threads. As a result, we use the term *core* and *thread* interchangeable hereafter.

pLock is built on this 2D mesh network. Since communications across rows or columns is not supported at the hardware level, they must be relayed in the software level by developers. Other architectures may have better hardware support for direct communication between arbitrary cores, but they may still have a non-uniform network topology due to a large number of cores.

## 2.2 A Basic Design of EMP-Lock

In this subsection we discuss the basic idea of using explicit message passing to implement a fast lock (*EMP-lock*), and its advantage over a traditional lock based on shared-memory (*SHM-lock*). Suppose there are two cores $A$ and $B$, they are competing for a spin-lock $L$. Figure 2 shows the memory traffic caused by the lock competition. Each core accesses memory and examines the status of $L$ to check if it is available. If so, it acquires the lock and sets its status to occupied; if the lock is already occupied by another core, it keeps checking the status until the current holder releases the lock. As shown in Figure 2, since $A$ already got the lock, $B$ keeps accessing memory, which produces a lot of memory traffic. This memory contention can lead to severe performance degradation. The simple implementation of SHM-lock is used to demonstrate the problem. More advanced shared-memory lock leverages optimization techniques like exponential back-off to mitigate memory contention, but they can only mitigate but not eliminate the problem. We will provide more discussion in Section 7.

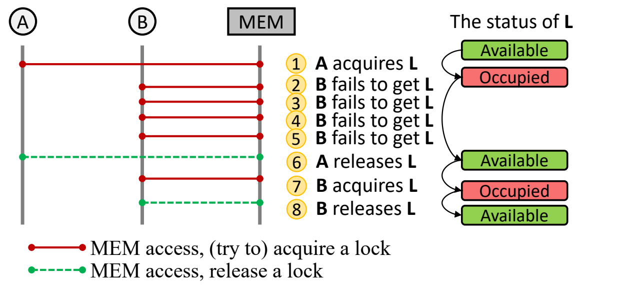


**Figure 2.** A spin lock based on shared memory.

EMP-lock is a design drastically different from SHM-lock. Status of locks are maintained by the lock server and can not be directly accessed from clients. Clients acquire and free locks via sending and receiving messages to and from a lock server, through the explicit message passing interfaces provided by architecture. Therefore, the memory traffic is significantly reduced compared to SHM-lock. Also, since only the lock server $S$ accesses the lock in memory, the memory locality is improved.

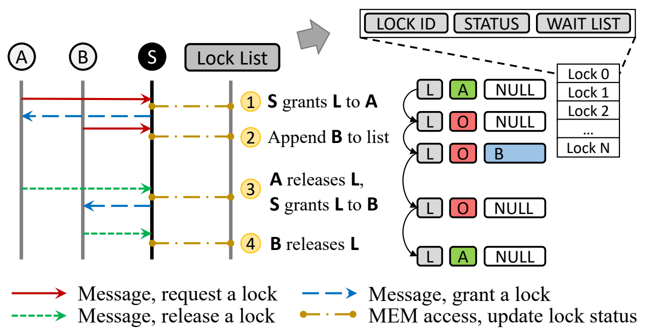


**Figure 3.** A basic design of EMP-lock

In a basic EMP-lock design shown in Figure 3, a core $S$ is used as a dedicated lock server, and all the other cores are clients ($A$ and $B$). Below is how two clients compete for a single lock $L$:

1. Client $A$ needs a lock $L$, so it sends a *request* message to server $S$;
2. Server $S$ received this request. Since the lock is currently *available*, $S$ sends a *grant* message back to client $A$;
3. $B$ also requests a lock, but the lock is now occupied by $A$. So $S$ append $B$ to the waiting list of lock $L$;
4. To free the lock, client $A$ sends a *release* message to server;
5. After receiving the release message, $S$ picks a client ($B$) from the waiting list and grants it the lock.
6. Client $B$ releases the lock, there is no client in the waiting list so $S$ changes the status of $L$ back to available.

For multiple locks, the lock server maintains a *lock list*, in which an entry represents the status of a lock. In a multi-lock situation, there can be one lock server that holds all locks or multiple lock servers that each holds part of locks. A previous study on simulator shows that a lock server can handle many locks without becoming a performance bottleneck [10]. For simplicity, in the rest of this paper, we only discuss the scenario that there is only one lock server and all clients compete for the same lock.

Now we analyze the communication overhead of this basic design. To execute a critical section, a client needs to (1) send a request message to the server; (2) receive the grant message from the server; (3) send a release message to the server. As a result, for $N$ clients each executes $S$ critical session, the message count is

$$M_{basic} = 3NS \tag{1}$$

In other words, three messages are passed among cores for each client and each critical session.

Using an EMP-Lock, the core $S$ is used as a dedicated lock server and no longer available for computation, only the client cores are used for computation. This design principle sacrifices the compute resources for dedicated synchronization support, which may slow down lock non-intensive programs. However, as we will see in Subsection 5.2, for lock-intensive programs, the lock performance will compensate the disadvantage of less computing cores.

## 3 pLock Approach

### 3.1 Chaining Lock

In this subsection, we introduce chaining lock technique to reduce message count. In the basic design (Figure 3), clients only communicate with the server. When a client frees a lock, it sends a release message to the server, then the server receives this message and grants the lock to the next client in waiting list. The key idea of chaining lock is: when a client frees a lock, instead of releasing it to the server, it passes the lock to a waiting client. Figure 4 shows the idea of chaining lock.

To enable lock passing among clients, the client that currently holds a lock must know who is waiting for the lock. To do so, when a lock server sends a grant message to a client, it also piggybacks the waiting list of the lock. As the example in Figure 4, the work is done in following steps:

1. All clients ask $S$ for lock $L$;
2. $S$ grants lock $L$ to client $A$, and also sends the waiting list *[B,C,D]* to $A$;
3. When client $A$ finishes its work and wants to free $L$, it checks the waiting list received with the lock;
4. Then $A$ notices that $B$ is waiting for the lock, so it passes the lock $L$ along with the updated waiting list *[C,D]* to client $B$.
5. After $B$ finishing its work, it passes lock $L$ and the new waiting list *[D]* to $C$;

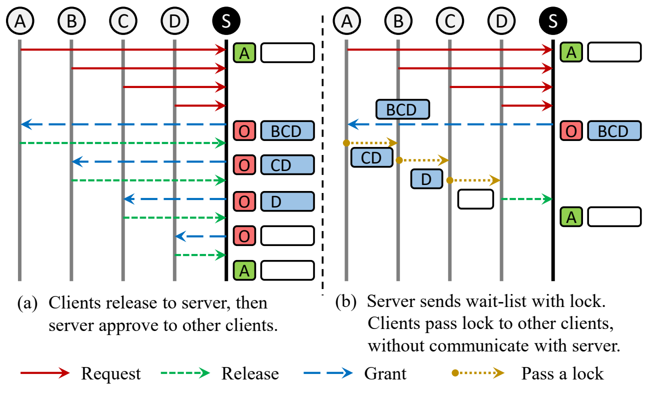

A
B
C
D
S
A
BCD
O
CD
D
(a) Clients release to server, then server approve to other clients.
(b) Server sends wait-list with lock. Clients pass lock to other clients, without communicate with server.
Request
Release
Grant
Pass a lock

**Figure 4.** Chaining lock mechanism allows clients to pass a lock to others, thus reduces message count.

6. Client $C$ finishes its work and passes an empty waiting list to *[D]*;
7. At last, client $D$ sees nothing in the waiting list so it sends a release message to server $S$.

The message count of chaining lock depends on the behaviors of programs. If the waiting list is long when the server grants a lock, the passing chain will be long too. More particularly, all clients must send a request and a release (or pass), but there is only one grant message in the best case. The chain length is limited by the number of clients since a waiting list cannot be longer than the full set of clients. On the contrary, if no client is in the waiting list for each time the server grants a lock, there will be no lock passing.

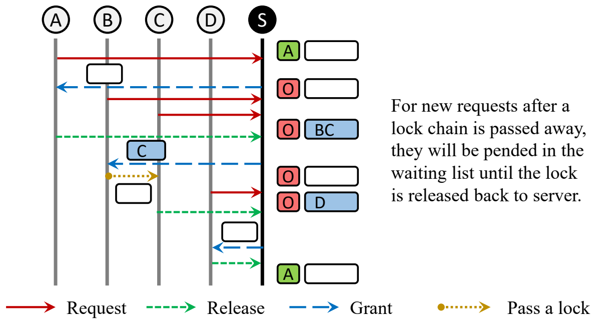

A
B
C
D
S
A
O
BC
C
D
For new requests after a lock chain is passed away, they will be pended in the waiting list until the lock is released back to server.
Request
Release
Grant
Pass a lock

**Figure 5.** Chaining lock mechanism allows clients to pass lock to others, thus reduces message count.

Figure 5 shows a non-ideal case of lock passing. In this example, the server $S$ has granted the lock $L$ to client $A$ before more requests come, so client $A$ cannot pass the lock to anyone, instead it needs to release the lock to server $S$. The only chain appears in client $B$ and $C$. When server grants the lock to $B$, the client $C$ is already pending in waiting list, so it can be sent to $B$ together with the grant message.

Suppose there are $N$ clients and $S$ critical sections, the message count in the worst case is

$$M_{chain\_worst} = 3NS \tag{2}$$

And the message count in the best case is

$$M_{chain\_best} = S \times (N + 1 + N) = (2N + 1)S \tag{3}$$

From Equation 2 and 3 we can see that, in the worst case, chaining lock does not introduce any extra messages, and in the best case, when $N$ is very big, chaining lock reduces about $1/3$ communications.

Chaining lock also has some drawbacks. All clients need to allocate space for storing a waiting list, and they need to check the list to decide where to send the release message, which also introduces additional computation overhead. Besides, the message content in this approach needs to include a waiting list, which may enlarge the message size. Nevertheless, both benefits and overheads depend on programs and platforms. Fortunately, in Section 4 we will see that, on the SW26010 processor, these overhead can be hidden with a careful implementation.

## 3.2 Hierarchical Lock

As discussed in previous sections, the network performance of a many-core processor can be different for nearby and faraway cores. Based on that, we introduce a technique named *hierarchical lock* to remove slow long-distance communication. Figure 6 demonstrates the comparison of the basic design and the hierarchical design.

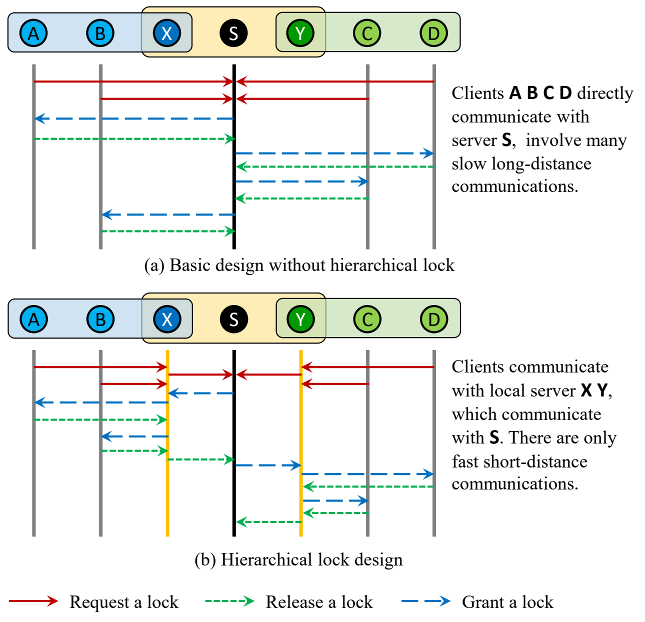

A
B
X
S
Y
C
D
Clients A B C D directly communicate with server S, involve many slow long-distance communications.
(a) Basic design without hierarchical lock
Clients communicate with local server X Y, which communicate with S. There are only fast short-distance communications.
(b) Hierarchical lock design
Request a lock
Release a lock
Grant a lock

**Figure 6.** Hierarchical lock strategy reduces long-distance inter-core communication.

In Figure 6 there are seven cores. According to the distance of each pair, we cluster the cores into three groups: $\{S, X, Y\}$, $\{X, A, B\}$, and $\{Y, C, D\}$. Communications inside a group are short-distance and fast, while communications across groups are long-distance and slow. We can see that $X$ and $Y$ are clustered into two groups, we call these cores *ad-core*. Core $A$, $B$, $C$, $D$ are clients and $S$ is the lock server. In Figure 6 (a), clients communicate with the server directly and generate a lot of long-distance communications.

The key idea of hierarchical lock is to use ad-cores as local servers to avoid long-distance communication. Figure 6 (b) demonstrates the principle of hierarchical lock, the work flow can be listed as below:

1. Client $A$ sends a request message to local server $X$ for lock $L$;
2. $X$ does not own the lock, so it asks $S$ for it;
3. $S$ grants $L$ to $X$, and then $X$ grants $L$ to $A$;
4. $A$ releases $L$, and $X$ grants $L$ to $B$. At this time, since $X$ already owns $L$, it does not need to ask $S$ again;
5. After $X$ releases $L$ to $S$, $Y$ gets $L$ and the process repeats.
6. At last, all clients have done their work and the lock $L$ is released back to server $S$.

In the hierarchical lock design, a local lock server acts as a lock cache to provide fast response to clients in its group. The message count of hierarchical lock depends on the network topology and the time sequence of client requests.

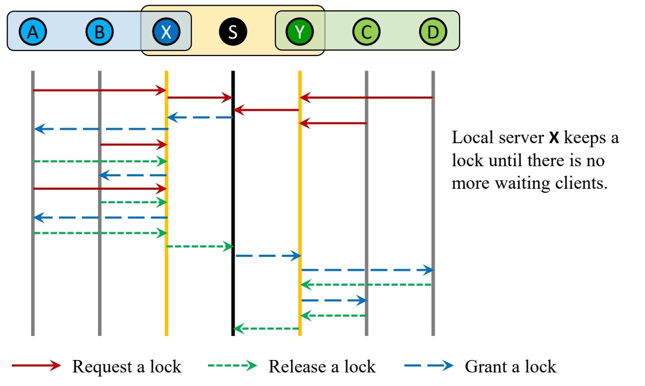

A
B
X
S
Y
C
D
Local server X keeps a lock until there is no more waiting clients.
Request a lock
Release a lock
Grant a lock

**Figure 7.** A local server prioritizes the lock locality.

In Figure 7 we can see that a local server manage to avoid releasing a lock. More particularly, a local server will not release a lock to the global server unless its waiting list is empty. So, although clients $C$ and $D$ sent requests earlier than the second request of $A$, they can only get the lock after $A$ finishes its work. There can be some fairness problems with this design. In a highly-contended situation, some threads may be starved for a while since a lock is repeatedly requested by clients belonging to another local server. However, most HPC programs prioritize throughput and care little about which thread finishes first, so we choose to minimize the message count between the global server and local servers to improve the throughput.

Suppose there are $N$ clients, $S$ critical sections, and $T$ local servers, we analyze the message count for both best and worst cases.

In the worst case, a local server serves only one client then frees the lock to the global server. As a result, an execution of a critical section involves two requests, two grants, and two releases.

$$M_{hier\_worst} = 6NS \tag{4}$$

Equation 4 indicates that the hierarchical lock method doubles the message count of the basic design. However, since the short-distance communication here is faster than long-distance ones in the basic design, it is not necessary to take double time.

In the best case, a local server only communication with the global server once, i.e., it frees the lock after all local clients finish their work. In this case, the message count is

$$M_{hier\_best} = 3NS + 3T \tag{5}$$

From Equation 5 we can see that, in the best case, if $T$ is much less than $NS$, the hierarchical lock method is approximately to replace all long-distance communications by short-distance communications. This change of network distance can lead to a performance improvement in communication.

### 3.3 Combining Two Techniques

We can combine chaining lock with hierarchical lock together to get an even faster design for EMP architectures.

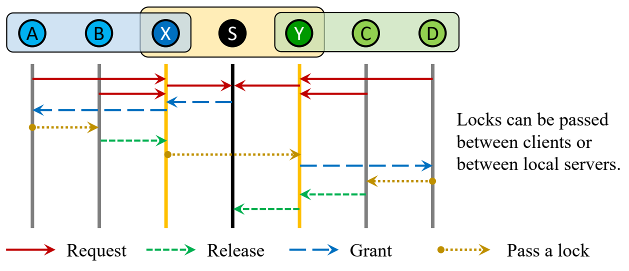

A
B
X
S
Y
C
D
Locks can be passed between clients or between local servers.
Request
Release
Grant
Pass a lock

**Figure 8.** A hierarchical chaining lock

Figure 8 illustrates a combined design. Cores are organized hierarchically to reduce long-distance communication. Chaining lock technique is applied to servers and clients. A client can pass a lock to another client belonging to the same local server. A local server can also pass a lock to another local server. In this example:

1. Client $A$ and $B$ send requests to their local server $X$, $C$ and $D$ send requests to local server $Y$;
2. Local server $X$ and $Y$ hold no lock, so each of them sends a request to global server $S$;
3. $S$ sends the granting message, along with the waiting list *[Y]* to $X$;
4. $X$ sends the granting message, along with the waiting list *[B]* to $A$;
5. After $A$ finishing its work, it passes the lock to $B$;
6. $B$ releases the lock to local server $X$ after work done.
7. $X$ passes the lock to another local server $Y$;
8. $Y$ grants the lock to $D$ and the process repeats;
9. Finally, local server $Y$ releases the lock to global server $S$.

With this combined design, the message count for the worst case and the best case will be:

$$M_{plock\_worst} = 6NS \tag{6}$$

$$M_{plock\_worst} = (2N + 1)S + 2T + 1 \tag{7}$$

Equation 6 is identical to Equation 4 because neither hierarchical or chaining technique works. Equation 7 is a derivation of Equation 3 and Equation 5. In the best case, compared with the basic design, the combined design can reduce 1/3 communications and improve the performance of the rest.

## 4 Implementation

In this section, we describe the implementation of pLock on the SW26010 architecture. According to its architecture, as shown in Figure 1, we map servers and clients to cores in a manner shown in Figure 9.

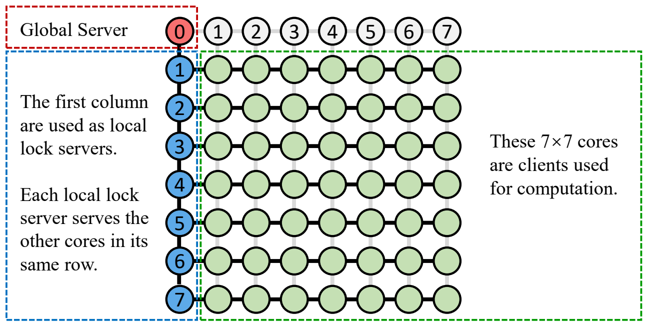
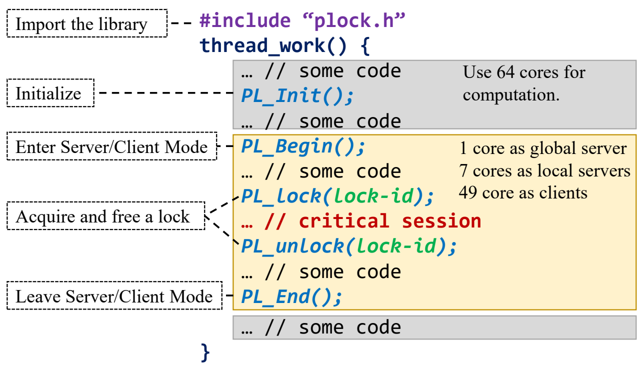

Global Server
0
1
2
3
4
5
6
7
The first column are used as local lock servers.
Each local lock server serves the other cores in its same row.
These 7×7 cores are clients used for computation.

**Figure 9.** Mapping the cores to global lock server, local lock servers and clients.

We use core 0 as the global lock server, and use the first column (except core 0) as local servers. Cores in other columns are used as clients. The communication latency between any two cores (in the same row or same column) are the same on SW26010. Therefore, the choice of server column has no impact on the performance. We select the edge column for convenience, but it can be any other column as well. A local server serves the clients in the same row. For basic design without a local server, the local server cores act for relaying messages. This is because the hardware messaging is only supported within a row or a column, so we have to dedicate the first column for software message relaying. Moreover, the server cores are saturated by the relaying work and are unable to do computation (as a client) simultaneously. As a result, this *49 clients + 8 servers* design is the only feasible solution of implementing EMP lock on SW26010, and we cannot implement *63 clients + 1 server* due to the on-chip network constraints of SW26010. We give a discussion for more flexible EMP architectures in Section 6.

In practice, we can also include the first row as clients. To do so, core 0 will be both a global server and a local server at the same time. However, core 0 will handle more messages than others and become a performance bottleneck. For higher performance and more transparent implementation, we let each core plays only one role, and we leave core 1-7 in the first row unused. Using 15 cores for lock sacrifices computational power, but benefits the synchronization. We discuss the tradeoff in the discussion under Figure 13, where the data show that *49 cores + pLock* is better than *64 cores + SHM-Lock* when the critical section consumes more than 0.88% of total time.

When using multiple SW26010 processors for parallel computing, a CG is bound to a process, so the change of the number of available computing cores only affects the thread-level parallelism, but does not affect the process-level decomposition. In other words, we do not need to allocate more CG or introduce more network communication for using pLock.



```
#include "plock.h"
thread_work() {
  … // some code
  PL_Init();
  … // some code
  PL_Begin();
  … // some code
  PL_lock(lock-id);
  … // critical session
  PL_unlock(lock-id);
  … // some code
  PL_End();
  … // some code
}
```

**Figure 10.** The APIs and usage of pLock

We have implemented pLock as a library for developers. The core APIs and usage of pLock are demonstrated in Figure 10 with C-style pseudo-code. Leaving 15 cores for non-computational purpose raises a question that whether fewer cores lead to lower performance. To efficiently utilize the resource of a CG, we do not enforce developers to use this core mapping strategy at all time. Instead, developers can enter server/client mode before a critical section intensive area and back to normal mode after that. This switching allows developers to use 64 cores for embarrassing parallel workload and then switch to 49 cores for workload that needs synchronization. Developers can use `PL_Begin` to assign roles to cores as servers or clients, then use `PL_End` to go back to the 64-core mode. Inside the surrounding area from `PL_Begin` to `PL_End`, developers can use `PL_lock` and `PL_unlock` to acquire and free locks. Also, a more conservative way is to use 49 cores from the beginning to avoid potential load re-balancing among threads due to mode switching. In Section 5 we will see that pLock is much faster than a shared-memory lock. For a program with highly contended critical sections, using 49 cores plus pLock can be faster than using 64 cores plus shared-memory lock.

The next step of implementation is message encoding. The inter-core communication APIs in SW26010 support sending and receiving 256-bit messages. We encode a message as shown in Figure 11.

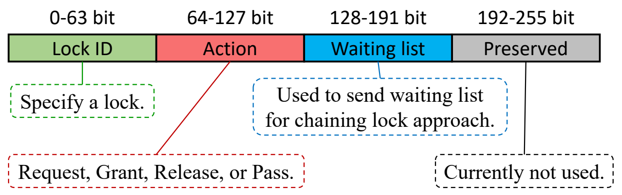


**Figure 11.** Message encoding for SW26010

A 256-bit hardware message is divided into four segments, and each has 64 bits. These four segments are used as:

- **Lock ID.** This field is used to specify which lock to operate on.
- **Action.** This field can be *request*, *grant*, *release*, or *pass*.
- **Waiting List.** As discussed in Subsection 3.1, chaining lock needs to transfer waiting lists among cores.
- **Preserved.** This field is currently not used.

As SW26010 supports 256-bit hardware-level message, although we can use fewer bits to encode fields list above, we use 64 bits for each to simplify encoding/decoding operations for a high-performance implementation on SW26010. Also, it is possible to implement non-trivial routing algorithms, e.g., non-minimal routing, to support more complex server-client layouts. However, on SW26010, the message encoding/decoding/addressing operations are more expensive than sending the message itself. Therefore, we keep the logic as simple as possible.

At the end of Subsection 3.1 we discuss possible drawbacks of chaining lock. For SW26010, since the hardware-supported message length is long enough to contain a waiting list, chaining lock does not introduce any overhead for message passing.

Based on this implementation, we revisit the message count of approaches described in Section 3. In this discussion, we suppose that all the 49 clients are used. On SW26010, a long-distance message passing is implemented by two short-distance message passing. So the message count for the basic design becomes

$$C_{sw\_basic} = 6NS = 294S \tag{8}$$

For hierarchical lock design, we have $T = 7$, and the message count of the best case is

$$C_{sw\_hier\_worst} = 6NS = 294S \tag{9}$$

$$C_{sw\_hier\_best} = 3NS + 3T = 147S + 21 \tag{10}$$

We build chaining lock on the base of hierarchical lock, so its message count is

$$C_{sw\_plock\_worst} = 6NS = 294S \tag{11}$$

$$C_{sw\_plock\_best} = (2N + 1)S + 2T + 1 = 99S + 15 \tag{12}$$

We can see that the worst cases of hierarchical and chaining lock have the same message count with the basic design. For a critical section inside a loop with many iterations, $S$ can be quite large ($S \rightarrow \infty$), and we have extreme case

$$C_{sw\_basic} : C_{sw\_hier\_best} : C_{sw\_plock\_best} = 6 : 3 : 2 \tag{13}$$

Equation 13 indicates that, compare with the basic design, hierarchical lock can reduce at most 50% inter-core communications and chaining lock can reduce at most 67%. Reducing communication have potential benefits to improving the lock performance, but additional logic work in hierarchical and chaining lock may undermine this advantage. We evaluate our approaches in Section 5.

## 5 Evaluation

### 5.1 Methodology

We have done experiments to evaluate the performance of pLock. Experiments were done on the Sunway TaihuLight system. Since this work focuses on the thread synchronization inside a process, all experiments are done using a single CG of SW26010. Three approaches, (1) the Basic design (*Basic*), (2) Hierarchical lock (*Hier*) and (3) the combination of hierarchical and chaining, i.e., our design (*pLock*) are evaluated. All experiments of EMP lock approaches were done with 1 global server thread, 7 local server threads, and 49 client threads.

Note that we do not use shared-memory lock approaches or atomic-based approaches as a baseline, because SW26010 has no coherent cache, shared-memory lock approaches can only use atomic operations that directly access main memory, which results in long latency and low throughput. Only for some test cases, we compared EMP lock with shared-memory lock or atomic approaches to validate our idea.

In Subsection 5.2, we design several micro-benchmarks to validate our ideas. Experiments in this subsection include latency, throughput, scaling performance of three EMP-lock designs. We also record the message count to verify our analysis in previous sections.

In Subsection 5.3, we evaluate the performance benefits of pLock on five benchmarks. *Counter* and *Stack* are two popular data structures used in multi-threaded programs. *BFS* (breadth-first search) and *Tri* (triangle count) are benchmarks from Graph500 and CRONO [2]. Despite above four common test programs for lock performance evaluation, we design an *Imbalanced Counter* test program, to evaluate the performance under a highly imbalanced workload.

As mentioned in Section 3, the performance of our approach depends on the workload. However, in our experiments, for a given workload, the performance of our approach is quite stable. This is partially due to the in-order design of SW26010 architecture. We use the average of three times measurement as the results for this section.

### 5.2 Validation

Figure 12 shows the latency of a critical section using three EMP-lock approaches. In this experiment, threads are doing critical sections repeatedly. There is nothing inside the critical section, only lock and unlock. The interval between two critical sections varies from 100 cycles to 100,000 cycles.

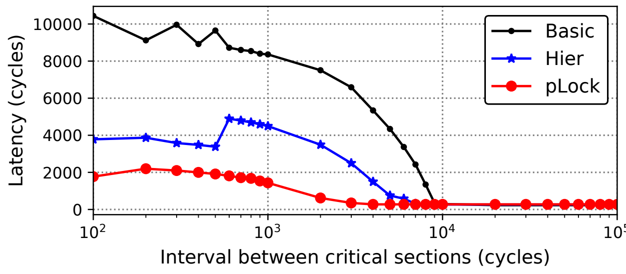


**Figure 12.** The average latency of a critical section using three EMP-lock approaches

From Figure 12 we can see that, In a lightly contended situation (interval > 10,000 cycles), all approaches have short latency. In a highly contended situation, where there is communication congestion, the benefit of reducing message count is obvious. With an interval of 100 cycles, *pLock* is 5.89× faster than *Basic*. Even more, with an interval of 4000 cycles, *pLock* is 19.41× faster than *Basic*. To explain the sudden shoots up around 600 cycles for *Hier*, we can look at the X-axis from the right. With a decreasing interval, the latency of *Hier* increases due to higher network traffic. However, when the interval is short enough, a local lock server will always have pending requests, so it keeps serving local clients without giving the lock back to the global server. In this case, the latency drops due to lock locality.

Latency shown in Figure 12 represents the average time cost from entering *lock()* to leaving *unlock()*. Throughput shown in Figure 13 represents the total operation rate of the whole program, i.e., how many critical sections are completed within a period. The throughput is measured in Mops (Million Operations Per Second). With a long interval, the throughput is limited by the parallelism of the programs itself, so we focus on the results with short intervals.

From Figure 12 and Figure 13 we can see that, although the latency of *Hier* is much shorter than *Basic*, its throughput improvement is less impressive. *pLock*, on the other side, has much higher throughput than *Basic*. With a 200-cycle interval, *pLock* has 3.84× improvement over *Basic*. We can

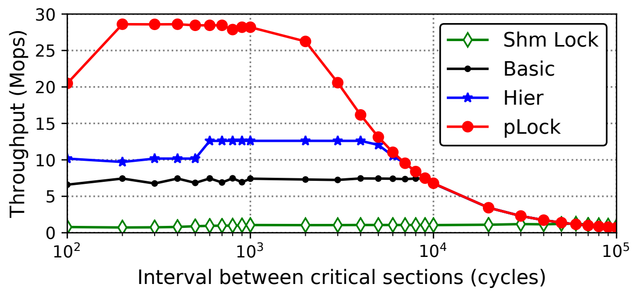


**Figure 13.** The throughput of three EMP-lock designs and SHM-lock.

also see that the throughput of shared-memory lock with 64 threads (*Shm Lock*) is much lower than pLock (1.03 Mops vs. 28.19 Mops).

We use an example to show that 49 cores plus pLock can work better than 64 cores plus shared memory lock, for a program with a highly contended critical section. Suppose there is a program using pLock and it contains a critical section with 1000 cycles interval, which occupies $S$ of total time and the rest $1 - S$ is embarrassing parallel workload. If we use 64 cores plus *Shm Lock*, the time will be

$$T = \frac{28.19}{1.03}S + \frac{49}{64}(1 - S) = 26.60S + 0.7656 \qquad (14)$$

We can have $T < 1$ when $S < 0.0088$. In other words, pLock is better than *Shm Lock* when the critical section consumes more than 0.88% of time.

Figure 14 and Figure 15 show the scaling performance of the three approaches. We fixed the interval between critical sections to 200 cycles, and vary the client number from 1 to 49.

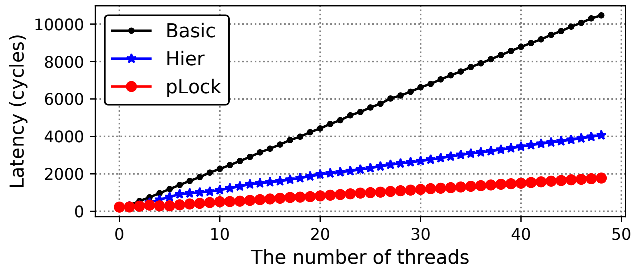


**Figure 14.** Latency scales with client numbers

For all of the three approaches, the latency is increasing proportionally to the number of threads. The increasing latency is due to the increasing length of the waiting list. For example, if there are 10 clients, a client will need to wait for 5 other clients on average; but if there are 40 clients, a client will need to wait for 20 clients on average.

Generally, throughput increases with the number of clients with some drop-down points. As we discussed in Section 3,

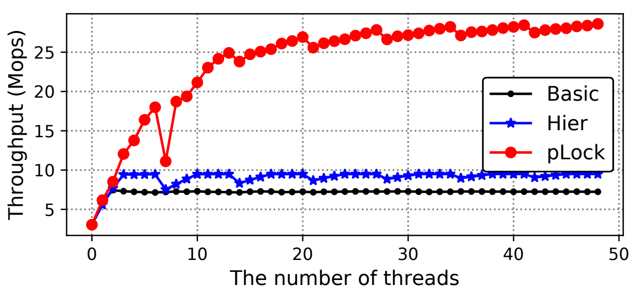


**Figure 15.** Throughput scales with client numbers

lock performance depends on the behavior of programs. When the lock acquisition has good locality, the communication also has good locality then the performance is better. The lock acquisition behavior depends on the time sequence of clients, which is further affected by the number of clients. From Figure 15 we can also see that, the scalability of *pLock* is better than the other two. With an interval of 200 cycles, *Basic* scales up to only 3 clients and *pLock* scales up to 20 clients. So pLock has better scalability than a basic EMP-lock implementation.

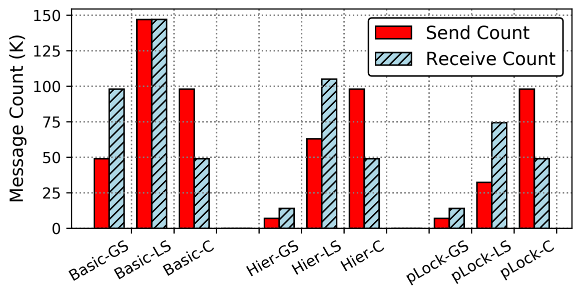


**Figure 16.** The send and receive message counts of the global server, local servers, and clients

To verify our analysis in Section 3 and Section 4, we let each client do critical sections for 1000 times, and record the message counts at runtime. The results are shown in Figure 16. Message counts are listed for the global server (GS), local servers (LS), and clients (C), respectively. We also distinguish send count and receive count. From Figure 16 we can see that the bars for *Basic* is symmetric. It is because that each lock operation involves a client-relay-server communication path. The total message count of LS is twice as GS and C, and a local server needs to receive then send for each message. Message count for GS and LS are reduced a lot with hierarchical lock. Chaining lock further reduces the message count for LS. The message counts of three approaches in this experiment are

$$C_{exp\_basic} : C_{exp\_hier} : C_{exp\_plock} = 6.00 : 3.43 : 2.80 \qquad (15)$$

Compare Equation 13 and 15, we see that the actual impact of hierarchical and chaining techniques is between the best case and the worst case.

When the global server and local servers handle fewer messages, they can respond much more quickly than before, As a result, while *pLock* only reduce 25% messages of *Hier* with an interval of 4000 cycles, its performance is much better than *Hier* (4.98× for latency and 2.10× for throughput).

### 5.3 Case Studies

We use three EMP-lock designs on five multi-threaded programs: *Stack*, *Counter*, *Imbalanced Counter*, *BFS*, and *Tri*. The speedup over basic design is shown in Figure 17. In this figure we also show the geometric mean of speedup.

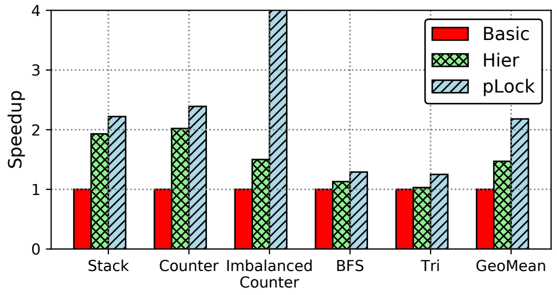


**Figure 17.** Speedup over basic design. The speedup of *pLock* for Imbalanced Counter is 5.76 and its bar is truncated.

Figure 17 shows that *pLock* approach has the best performance. In average, *Hier* and *pLock* improve performance by 47% and 118%, respectively. For *Imbalanced Counter*, *pLock* achieves the highest speedup, 5.76× over *Basic*. In this subsection, we analyze the characteristics of these programs to explain the results.

*Counter* and *Stack* are two synthesized benchmarks. Threads keep doing a critical section for many times, inside the section, they increase a global counter or push new elements into a global stack, respectively. These two benchmarks were run with 49 clients and 200 cycles interval, their performance is measured by throughput (Mops). The throughput of three approaches and two benchmarks, along with the results in Figure 13, are shown in Table 1.

**Table 1.** The critical section duration and throughput of *micro-test*, *Counter*, and *Stack* benchmarks.

| | The Duration of a Critical Section | Throughput (Mops) | | |
|---|---|---|---|---|
| | | Basic | Hier | pLock |
| Micro-Test | 0 cycles | 7.44 | 9.71 | 28.60 |
| Counter | 419 cycles | 1.72 | 3.47 | 4.12 |
| Stack | 460 cycles | 1.71 | 3.31 | 3.80 |

As we can see in Table 1, the throughput of *Counter* is much lower than *micro-test* in Subsection 5.2. This is because the global memory access in SW26010 is quite slow and the long duration of critical section limits the parallelism of threads. Since there are more memory access operations in *Stack*, its critical section is even longer, so the throughput is lower. As the proportion of memory accessing increases, the performance impact of lock optimization becomes smaller. So the speedup for *Stack* is lower than *Counter*.

We also implemented an *atomic counter*, which uses 49 threads to concurrently increase a counter using the atomic increment instruction. Since the atomic operations on SW26010 directly access the main memory, the atomic increment was so slow that the atomic-based counter is 2.01× slower than *pLock* version.

Another test program *Imbalanced Counter* has non-uniform workload for clients. A client $k$ has $k$ times workload of client 1. We also record the message counts of using three approaches and the ratio is 6.00 : 4.26 : 2.92. However, *pLock* got speedup even more than 3×. This is because the imbalanced workload causes network congestion in Column 0 (the server column) of the chip, and our chaining lock technique can efficiently reduce the inter-server communication and avoid network congestion. In this experiment, the ratio of inter-server message count was 100.53 : 30.56 : 1.00. In other words, pLock reduced 51% overall communication and 99% inter-server communication.

BFS and Tri are two graph computing programs. We run BFS with 245K vertexes with an average degree of 16. For Tri, the test graph has 100K vertexes and the average degree is 16. From Figure 17 we can see that pLock improved the performance of BFS and Tri by 29% and 25% respectively.

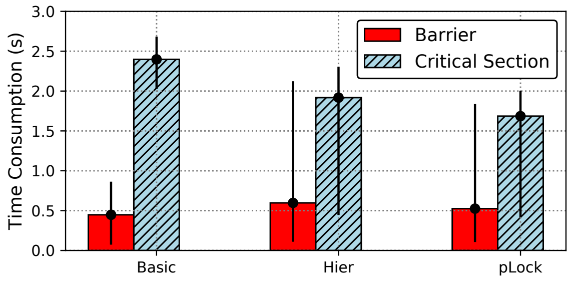


**Figure 18.** Time consumption of global synchronization (barrier) and critical sections

We further profile the execution time of BFS, for both critical sections and other parts. Figure 18 shows the time consumption of global synchronization (barrier) and critical sections of a BFS run. The time consumption was measured for each core, the bars in Figure 18 represent the average time of all client cores. The error bars show the minimum and maximum values. We can see that while pLock reduces the time for critical section, the variance of barrier time was increased. The reason is that while both hierarchical lock and chaining lock techniques improve the locality of lock acquiring, the basic design has better fairness. Nevertheless, pLock still has better overall performance than the basic design.

## 6 Discussions

In this work, we present pLock, a lock approach that focuses on throughput. Some programs may have other priorities such as fairness and security. A fair and safe lock on EMP architectures remains as future work. Allowing clients to pass locks to others may introduce potential security issues. As a temporal solution for security concern, we can fall back to a centralized server-client solution by forcing lock servers always to return an empty waiting list.

The two techniques used in pLock, chaining lock and hierarchical lock, are general techniques for any architectures that support EMP. For a new architecture that supports arbitrary pairwise communication, there will be more possible EMP lock hierarchies and the optimal hierarchical solution can be different from the one of SW26010. Also, if an EMP architecture supports coherent shared cache, then cores can use cache to share waiting list, and there is no need to pass a waiting list among client cores explicitly.

## 7 Related Work

Multi-thread programming is widely used to efficiently utilize the computational ability of multi-core and many-core processors. Since many threads share a region of memory, synchronization is necessary to avoid data race and ensure the correctness of concurrent operations [8, 15, 23, 36].

Using locks to protect critical session is a popular solution for synchronization. Many techniques can be used to implement locks. Spin lock [4] is a classic and straightforward lock approach. A spinlock keeps checking a shared variable until it gets the lock. Spinlocks have poor scalability since all threads spin on a single memory location, which can lead to heavy contention. Lock contention can cause significant performance loss [37], so researchers have investigated many approaches to reduce or avoid lock contention.

When a thread cannot get a lock immediately, it can wait for a period then retry. A spin lock with exponential back-off algorithm [22] has much better performance than a naive implementation. Queue locks like MCS [27] and CLH [7] were proposed to address the lock contention problem. Threads using queue locks append themselves to the end of a queue. When a thread releases a lock, it will pass the lock to its successor. Queue locks have good performance in a highly contended situation. However, if a thread is scheduled to the background by OS, the lock passing chain may break and harm performance. In modern operating system (OS), threads can also suspend until the lock is available. This is the approach used by POSIX mutex (`pthread_mutex_lock`), a better choice when processors are highly over-subscribed. Since queue locks and POSIX mutex introduce additional overhead, the most straightforward spinlock has the best performance under no-contended or lightly-contended situations. QOLB [14, 20] uses a hardware-supported queue and cache-line associated *syncbits* to organize waiting processes and critical data. With the hardware support, QOLB can simplify and accelerate lock transferring among processes on cache-coherent architectures.

The lock approaches above are all trying to transfer the permission of executing critical sessions among threads. As a result, cache lines containing lock variables and shared data are also transferring between cores. In nowadays processors, cache transfer can be expensive. Some researchers propose new cache coherence protocol extension to mitigate the contention in cache [16], and some others propose software solutions called *delegation* technique to improve memory locality. Instead of transferring permission via locks, delegation synchronization approaches transfer computation, i.e., the work of a critical section. With *Flat-Combining (FC)* [12, 17] technique, threads that do not get a lock appends their requests to an additional request queue, then the thread which gets the lock not only finishes its job but also reads the request queue and does the work for others threads. The *service threads* in FC is dynamically selected, in RCL [25], service threads are dedicated and pinned to cores and have better cache performance. Some optimization in traditional locks are also be applied to delegation methods. Queue-delegation [21] use a queue to organize client threads. Server threads only need to communicate with the client thread in the queue head, without polling over all client threads. While delegation methods have good performance in highly-contended situations, their latency is longer than most locks due to remote-procedure-call (RPC) operation, so is not the best choice for lightly-contended programs. Besides, modification in the source code is required to adopt delegation approaches, and it can consume considerable manual efforts.

Recent architectural evolution also affects thread synchronization techniques. Many modern computers have Non-Uniform-Memory-Access (NUMA) architecture. NUMA-aware and hierarchical synchronization methods were proposed for both locking and delegation [12, 33]. HCLH [26] and Cohort-Locks [9] are hierarchical locks, and SANL [40] is a NUMA-aware delegation approach.

Another trend in a many-core processor is the undetermined future of cache coherence (CC) and the emerging of explicit message passing (EMP) [31, 39]. For inter-core communication, EMP is much faster than CC-based data sharing. A processor with EMP is somehow similar to a cluster or a distributed system. Although locks are also widely used in distributed systems [5], they often consider high availability and reliability, while locks for multi-thread programs put the performance (throughput and latency) in the first place. As

**Table 2.** A brief comparison of state-of-the-art synchronization approaches.

| Synchronization Approaches | Lock-Based or Lock-Free | Required Hardware Feature | Compatibility for Legacy Code | Performance in Highly-Contended Situations |
|---|---|---|---|---|
| Lock-free Data Structures [19] | Lock-free | Atomic instructions | Very Low | High |
| Delegation (shm version) [25] | Lock-free | Atomic instructions | Low | High |
| Transactional Memory [29] | Lock-free | Transactional memory instructions | Medium | Conflict rate dependent |
| Spinlock [4] | Lock-based | Atomic instructions | High | Low |
| POSIX mutex lock | Lock-based | OS dependent | High | Medium |
| Queue-lock [27] | Lock-based | Atomic instructions | High | Medium |
| Delegation (EMP version) [30] | Lock-free | Explicit message passing | Low | High |
| **pLock (our work)** | Lock-based | Explicit message passing | High | High |

a result, instead of using distributed locks directly on EMP processors, developers should carefully design new synchronization mechanisms to meet performance requirements. EMP has been used to accelerate the request sending routine in RCL [30], and the performance is improved by 4.3×. On Sunway Taihulight [13], concurrent data classification obtains a speedup of 19.15× after changing synchronization method from shared-memory based locking to EMP based delegation [24]. Dogan et al. designed a server-client lock model to accelerate multi-thread programs while preserving the original lock interface [10]. Our work also uses the server-client lock model used in Dogan's. Nevertheless, our hierarchical lock and chaining lock techniques improved the performance of EMP-based lock by up to 10.38× over the intuitive method used in Dogan's work.

There are also some alternative synchronization methods without explicit locking. Simple and common data structures like queue [35], stack [18], hash table [28], have been redesign to be *lock-free*. Universal construction approaches for wait-free or lock-free data structures have also been proposed [3, 11]. Nevertheless, auto-constructed lock-free data structures can introduce additional memory space usage and performance overhead. Transactional memory has been proposed to support more aggressive parallel execution. Software transactional memory (STM) [34] is too slow thus is not practical [6]. Hardware transactional memory (HTM) is much faster than STM, while still can be further improved [29]. HTM-based programs have good performance with low conflict rate. With frequent memory access conflict, HTM-based programs often fall back to using a traditional synchronization method.

There are so many synchronization approaches that many other works are not introduced in this paper. We summarize the pros and cons of state-of-the-art synchronization approaches in Table 2. As we can see, previous solutions either do not perform well in highly contended situations or lack of compatibility to use on legacy code. Our work, pLock, preserves the lock interfaces, so it is easy to be applied for legacy programs. It also leverages explicit message passing for high performance. Since *"every locking scheme has its fifteen minutes of fame"* [8, 15], several tools [30, 40] have been proposed to help developers to choose the best synchronization solution according to program workload and hardware platform.

## 8 Conclusion

In this paper, we discuss the principles of using explicit message passing (EMP) interface to design and implement mutex lock. We also proposed two techniques – hierarchical lock and chaining lock – to reduce message count and further improve the performance of EMP-lock. We implement our approach on SW26010, a product processor with EMP and is used in the supercomputer Sunway Taihulight. Experimental results show that our optimization techniques improve the performance by over 10× compared with the basic implementation.

## Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and suggestions. We also thank Bowen Yu, Heng Lin, Xiaosong Ma, and Xu Ji for their valuable feedback and suggestions. This work is partially supported by the National Key R&D Program of China (Grant No. 2016YFB0200100), National Natural Science Foundation of China (Grant No. 61722208), and National Science Foundation (Grant No. CCF-1657333, CCF-1717754, CNS-1717984, CCF-1750656). Jidong Zhai is the corresponding author of this paper (Email: zhaijidong@tsinghua.edu.cn).

## References

[1] 2018. top500 website. http://top500.org/. (2018).
[2] Masab Ahmad, Farrukh Hijaz, Qingchuan Shi, and Omer Khan. 2015. CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores. In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization (IISWC '15).*

[3] James H Anderson and Mark Moir. 1995. Universal constructions for large objects. In *International Workshop on Distributed Algorithms*. Springer, 168–182.

[4] Thomas E. Anderson. 1990. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 1, 1 (1990), 6–16.

[5] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 335–350.

[6] Calin Cascaval, Colin Blundell, Maged Michael, Harold W Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. 2008. Software transactional memory: Why is it only a research toy? *Queue* 6, 5 (2008), 40.

[7] Travis Craig. 1993. *Building FIFO and priorityqueuing spin locks from atomic swap*. Technical Report. Technical Report TR 93-02-02, University of Washington, 02 1993.

[8] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 33–48.

[9] David Dice, Virendra J Marathe, and Nir Shavit. 2015. Lock cohorting: A general technique for designing NUMA locks. *ACM Transactions on Parallel Computing* 1, 2 (2015), 13.

[10] Halit Dogan, Farrukh Hijaz, Masab Ahmad, Brian Kahne, Peter Wilson, and Omer Khan. 2017. Accelerating Graph and Machine Learning Workloads Using a Shared Memory Multicore Architecture with Auxiliary Support for In-hardware Explicit Messaging. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 254–264.

[11] Panagiota Fatourou and Nikolaos D Kallimanis. 2011. A highly-efficient wait-free universal construction. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 325–334.

[12] Panagiota Fatourou and Nikolaos D Kallimanis. 2012. Revisiting the combining synchronization technique. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 257–266.

[13] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, et al. 2016. The Sunway TaihuLight supercomputer: system and applications. *Science China Information Sciences* 59, 7 (2016), 072001.

[14] James R Goodman, Mary K Vernon, and Philip J Woest. 1989. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third Symposium on Architectural Support for Programming Languages and Operating Systems*. ACM, 64–75.

[15] Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. 2016. Multicore Locks: The Case Is Not Closed Yet.. In *USENIX Annual Technical Conference*. 649–662.

[16] Syed Kamran Haider, William Hasenplaugh, and Dan Alistarh. 2016. Lease/release: Architectural support for scaling contended data structures. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 17.

[17] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 355–364.

[18] Danny Hendler, Nir Shavit, and Lena Yerushalmi. 2010. A scalable lock-free stack algorithm. *J. Parallel and Distrib. Comput.* 70, 1 (2010).

[19] Moshe Hoffman, Ori Shalev, and Nir Shavit. 2007. The baskets queue. In *International Conference On Principles Of Distributed Systems*. Springer, 401–414.

[20] Alain Kägi, Doug Burger, and James R Goodman. 1997. Efficient synchronization: Let them eat QOLB. In *ACM SIGARCH Computer Architecture News*, Vol. 25. ACM, 170–180.

[21] David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. 2014. Brief announcement: Queue delegation locking. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*. ACM, 70–72.

[22] Byung-Jae Kwak, Nah-Oak Song, and Leonard E Miller. 2005. Performance analysis of exponential backoff. *IEEE/ACM transactions on networking* 13, 2 (2005), 343–355.

[23] M. LeBeane, S. Song, R. Panda, J. H. Ryoo, and L. K. John. 2015. Data partitioning strategies for graph workloads on heterogeneous clusters. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[24] Heng Lin, Xiongchao Tang, Bowen Yu, Youwei Zhuo, Wenguang Chen, Jidong Zhai, Wanwang Yin, and Weimin Zheng. 2017. Scalable Graph Traversal on Sunway TaihuLight with Ten Million Cores. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 635–645.

[25] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia L Lawall, Gilles Muller, et al. 2012. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications.. In *USENIX Annual Technical Conference*. 65–76.

[26] Victor Luchangco, Dan Nussbaum, and Nir Shavit. 2006. A hierarchical CLH queue lock. *Euro-Par 2006 Parallel Processing* (2006), 801–810.

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

[28] Maged M Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 73–82.

[29] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M Michael, and Hisanobu Tomari. 2015. Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 144–157.

[30] Darko Petrović, Thomas Ropars, and André Schiper. 2014. Leveraging hardware message passing for efficient thread synchronization. *ACM SIGPLAN Notices* 49, 8 (2014), 143–154.

[31] Carl Ramey. 2011. Tile-gx100 manycore processor: Acceleration interfaces and architecture. In *Hot Chips 23 Symposium (HCS), 2011 IEEE*. IEEE, 1–21.

[32] Sabela Ramos and Torsten Hoefler. 2013. Modeling communication in cache-coherent SMP systems: a case-study with Xeon Phi. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. ACM, 97–108.

[33] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. 2017. ffwd: delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 342–358.

[34] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L Hudson, Chi Cao Minh, and Benjamin Hertzberg. 2006. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 187–197.

[35] Michael L Scott and William N Scherer. 2001. Scalable queue-based spin locks with timeout. In *ACM SIGPLAN Notices*, Vol. 36. ACM, 44–52.

[36] Shuang Song, Xu Liu, Qinzhe Wu, Andreas Gerstlauer, Tao Li, and Lizy K. John. 2018. Start Late or Finish Early: A Distributed Graph Processing System with Redundancy Reduction. *PVLDB* 12, 2 (2018), 154–168.

[37] Nathan R Tallent, John M Mellor-Crummey, and Allan Porterfield. 2010. Analyzing lock contention in multithreaded applications. In *ACM Sigplan Notices*, Vol. 45. ACM, 269–280.

[38] Chao Yang, Wei Xue, Haohuan Fu, Hongtao You, Xinliang Wang, Yulong Ao, Fangfang Liu, Lin Gan, Ping Xu, Lanning Wang, et al. 2016. 10M-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics. In *High Performance Computing, Networking, Storage and*

*Analysis, SC16: International Conference for*. IEEE, 57–68.

[39] Zhiyi Yu, Ruijin Xiao, Kaidi You, Heng Quan, Peng Ou, Zheng Yu, Maofei He, Jiajie Zhang, Yan Ying, Haofan Yang, et al. 2014. A 16-core processor with shared-memory and message-passing communications. *IEEE Transactions on Circuits and Systems I: Regular Papers* 61, 4 (2014), 1081–1094.

[40] Mingzhe Zhang, Haibo Chen, Luwei Cheng, Francis CM Lau, and Cho-Li Wang. 2017. Scalable Adaptive NUMA-Aware Lock. *IEEE Transactions on Parallel and Distributed Systems* 28, 6 (2017), 1754–1769.