A Fast Lock for Explicit Message Passing Architectures

Xiongchao Tang, Chen Zhang, Jidong Zhai, Xuehai Qian, Wenguang Chen, and Yong Jiang

Abstract—Synchronization is a crucial issue for multi-threaded programs. Mutex locks are widely used in legacy programs and are still popular for the intuition semantics. The SW26010 architecture, deployed on the supercomputer Sunway TaihuLight, introduces a 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, especially designed and optimized for architectures with this new feature. In this article, we propose mLock, a fast lock designed and optimized for architectures that support Explicit inter-core Message Passing (EMP). mLock uses partial cores as lock servers and leverages the fast on-chip network to implement high-performance mutual exclusive locks. In this article, we propose a series of novel techniques to improve the performance of EMP locks. First, we propose the concepts of chaining lock and hierarchical lock to reduce message count and mitigate network congestion. Second, we propose a fair lock approach to improve the fairness of EMP locks. Third, server reusing is introduced to reduce the number of lock servers. We implement and evaluate mLock on an SW26010 processor. Experimental results show that our proposed techniques can improve the performance of EMP locks by up to 16.2x over a basic design.

Index Terms—Lock, synchronization, on-chip network, inter-core message passing

1 INTRODUCTION

I t is common to use mutual exclusive locks to synchronize 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, locks are widely used in legacy programs and are popular for their intuitive semantics. Because of the importance, locks have been carefully designed and implemented for modern multi-core architectures [1], [2].

However, for emerging many-core processors, conventional coherent cache architecture has more and more complex and it is very hard to achieve high performance [3]. A novel architectural feature, Explicit inter-core Message Passing (EMP), has gained popularity in research and even been used in some product many-core processors, such as TILE-Gx8036 [4] and SW26010 [5]. The Sunway TaihuLight [6] 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 [7], [8]. Although researchers have showed EMP’s advantage over traditional coherent cache architectures on thread synchronization [4], the widely used lock mechanism has not been specially designed and optimized for EMP architectures like SW26010. It 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 [9] or partially give up the opportunity to accelerate synchronization with EMP.

In this paper, we propose mLock, a new lock mechanism specially designed for EMP supported architectures. mLock enables developers to leverage the benefits of fast EMP while preserving the conventional lock and critical section paradigm. We build mLock as a library that provides conventional lock/unlock interfaces and hides all complex architectural details. As a result, application developers can use mLock 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, the cores no longer need to check the status of the shared memory region repeatedly and the high contention caused by synchronization 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 fewer cores can be used for computation, the faster lock still improves the performance of 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 [11], 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 on 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 reduce the number of messages among cores. (2) The inter-core network of SW26010 architecture is non-uniform. 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 mLock, a high-performance lock library that addresses the above two key challenges. In the previous EMP-lock design, if a client wants to obtain or release a lock, it needs to communicate with the lock server. This mechanism introduces a large amount of communication between the lock server and clients. Instead of sending requests to the lock server each time, we transfer locks among clients directly in mLock. Therefore, mLock speeds up the process of requesting and releasing locks, and reduces communication traffic to mitigate network congestion. Furthermore, we adopt a hierarchical lock mechanism to improve the communication performance on a non-uniform network.

In summary, we make the following contributions.

- We propose a chaining lock method, which allows a client to bypass the 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, the 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 use it on EMP architectures.
- We propose a fair lock approach to improve the fairness of EMP locks, which can avoid starvation in highly-contended situations.
- We propose a server reusing mechanism to reduce the number of lock servers and allow more cores to do computation.
- Based on the proposed approach, we implement mLock on a product processor with EMP support, i.e., SW26010. We evaluate our approach with microbenchmarks and multi-threaded parallel programs.

The experiment results show that, compared with a basic design, mLock can reduce up to 44 percent of communication and reduce lock latency by 93 percent. A preliminary version of this work was presented at ASPLOS 2019[12], which is a combination of the chaining and hierarchical method mentioned above.

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 architectural scalability.

SW26010 is a product processor that has been deployed on the supercomputer Sunway TaihuLight [5]. Fig. 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 external IO operations, most computation work is delivered to the 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 256-bit message saved in value to another core in the same row, specified by destination.
- 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 save 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 a 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 interchangeably hereafter.

mLock is built on this 2D mesh network. Since communications across rows or columns are 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). Supposing there are two cores A and B, they are competing for a spin-lock L. Fig. 2 shows memory traffic caused by lock competition. Each core accesses memory and examines the status of L to check if it is available. If so, it acquires a 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 Fig. 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 here is used to demonstrate the main problem. More advanced shared-memory locks can leverage optimization techniques such as exponential back-off to mitigate, but not eliminate, the problem of memory contention. We will provide more discussion in Section 6.

EMP-lock is a drastically different design from SHM-lock. The status of locks is maintained by lock servers and cannot 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 the architecture. Therefore, the memory traffic is significantly reduced compared to SHM-lock.

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

1) Client A needs a lock L, so it sends a request message to server S;
2) S receives 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 appends B to the waiting list of lock L;
4) To free the lock, A sends a release message to S;
5) After receiving the release message, S picks a client (B) from the waiting list and grants it the lock;
6) 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.

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 section, the message count is

\[ M_{\text{basic}} = 3NS. \]  

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

Using an EMP-Lock, the core S is used as a dedicated lock server and no longer available for computation. The main limitation of this design is that it sacrifices compute resources for dedicated synchronization support, which may slow down programs that do not use locks extensively. We will describe how we solve this problem in Section 3.5.

3 MLOCK APPROACH

3.1 Chaining Lock

In this subsection, we introduce a chaining lock technique to reduce message count. In the basic design, clients only communicate with the lock server. When a client frees a lock, it sends a release message to the server. The server receives this message and grants the lock to the next client in the waiting list. The key idea of a chaining lock is: when a client frees a lock, instead of releasing it to the server, it passes the lock to a waiting client. Fig. 4 shows the idea of a 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 Fig. 4, the work is done in the 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 $B$ finishes its work, it passes both lock $L$ and the new waiting list $[D]$ to $C$;
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 a chaining lock depends on the behavior of programs. If the waiting list is long when the server grants a lock, the passing chain will be long too. In particular, 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 a 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.

Fig. 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. The only chain appears in clients $B$ and $C$. When the server grants the lock to $B$, client $C$ is pending in the waiting list, so it can be sent to $B$ together with the grant message.

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

$$M_{\text{chain\_worst}} = 3NS.$$  \hspace{1cm} (2)

And the message count in the best case is

$$M_{\text{chain\_best}} = S \times (N + 1 + N) = (2N + 1)S.$$  \hspace{1cm} (3)

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

The 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 overheads can be hidden with 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. Fig. 6 demonstrates the comparison of the basic design and the hierarchical design.

In Fig. 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 Fig. 6a, clients communicate with the server directly and generate a lot of long-distance communications.

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

1) Client $A$ sends a request message to a 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 responses to clients in its group. The message count of hierarchical lock depends on the network topology and the time sequence of client requests.

In Fig. 7, we can see that a local server manages to avoid releasing a lock. In particular, a local server will not release a lock to the global server unless its waiting list is empty.
Fig. 7. A local server prioritizes the lock locality.

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 starve for a while since a lock is repeatedly requested by clients belonging to another local server.

Supposing there are N clients, S critical sections, and T local servers, we analyze the message count for both the 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, one execution of a critical section involves two requests, two grants, and two releases

$$M_{\text{hier, worst}} = 6NS.$$  (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 communicates 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

$$M_{\text{hier, best}} = 3NS + 3T.$$  (5)

Equation (5) shows that, in the best case, if T is much less than NS, the hierarchical lock method is approximate to replace all long-distance communications by short-distance communications. This change of network distance can lead to performance improvement in communication.

### 3.3 Combining Two Techniques

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

Fig. 8 illustrates a combined design. Cores are organized hierarchically to reduce long-distance communication. The 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, and 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 finishes its work, it passes the lock to B;
6) B releases the lock to X after work is 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

$$M_{\text{combined, worst}} = 6NS$$  (6)
$$M_{\text{combined, best}} = (2N + T)S + 2T + 1.$$  (7)

Equation (6) is identical to Equation (4) because neither hierarchical nor chaining technique works. Equation (7) is a derivation of Equations (3) and (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 if T is much smaller than N.

### 3.4 Fair Lock

In the hierarchical design, a local server would continue granting the lock to its clients without releasing it until its waiting list is empty. Clients that belong to other local servers cannot get the lock before that, so they would wait for a long time in a highly-contended situation.

For the combined design, we can improve fairness while not bringing any additional computation. A local server does not grant the lock to its clients after the lock is released from its clients. Instead, it passes the lock to another local server or releases the lock to the global server. If the waiting list of the local server is not empty, it needs to send another request to the global server. With this strategy, we can solve the unfairness without bringing any additional computation. As shown in Fig. 9, the workflow contains the following steps:

1) Clients A and B send requests to their local server X, C and D send requests to local server Y;
2) Local servers X and Y send requests to the global server S;
3) S grants the lock to X and X grants the lock to A;
4) After A finishes its work, it passes the lock to B;
5) A sends another request to its local server X;
6) B releases the lock to X after work is done;
7) X passes the lock to another local server Y despite client A is waiting for the lock;
8) X sends a request to global server Y despite client A is waiting for the lock;
9) C and D get the lock successively;
10) Y releases the lock to S;
S grants the lock to X, X grants the lock to A;
12) A releases the lock to X, X releases the lock to S.

In this fair approach, the message count for the worst case is

\[
M_{\text{fair, worst}} = 6NS. \quad (8)
\]

In the best case, a local server sends S requests to the global server and the message count is

\[
M_{\text{fair, best}} = (2N + 3T + 1)S. \quad (9)
\]

Compared with the combined design, the fair lock design needs more communications between the global server and local servers. However, the additional request from the local server is not on the critical path of lock-passing. In Fig. 9, the request from X to S (Step 8) and the lock granting to C (Step 9) are in parallel. And thanks to the fairness, clients of different local servers can do the work outside the critical section concurrently. Therefore, in Section 5, we will see that the performance of the fair lock design is better than the combined design.

### 3.5 Server Reuse

In an EMP-Lock, lock servers only manage lock transfer and are not in charge of computation, which can lead to a waste of computation resources. Therefore, we propose a server reuse mechanism to use one local server as the global server to reduce the number of lock servers and make more cores available for computation.

Fig. 10 shows an example of server reuse. Server cores are clustered into four groups: \{S, X, Y\}, \{X, A, B\}, \{Y, C, D\}, and \{S, E, F\}. Communications inside each group are fast. We use X, Y, S as local servers of clients \{A, B\}, \{C, D\}, \{E, F\} respectively and reuse S as the global server. When requesting a lock, clients A, B, C, and D send messages to their local servers X or Y, X and Y then send messages to global server S, while clients E and F can directly send the request to the global and local server S. Similarly, server S can directly grant the lock to clients E and F while it needs to grant the lock to a local server which grants the lock to clients A, B, C, and D.

Supposing there are N clients, M of which can communicate with the global server efficiently, T local servers including the global server and S critical sections, the message count for different cases are listed in Table 1.

In this design, S handles more messages than other local servers, but if message handling is fast enough, this will not become a performance bottleneck. Moreover, for tasks that need more than one lock, we can divide the locks to different lock servers to balance the communications.

In Fig. 11, there are two locks, one is owned by server X and the other is owned by server Y. In addition, X and Y act as the local server of \{A, B\} and \{C, D\} respectively. To request the lock owned by X, clients A and B can directly send messages to the global and local server X, while clients C and D need to send the message to their local server Y first. On the contrary, to request the lock owned by Y, clients C and D can send messages directly while clients A and B need the help of their local server X.

### 4 Implementation

In this section, we describe the implementation of mLock on the SW26010 architecture.

**Mapping Strategy.** According to the SW26010 architecture, as shown in Fig. 1, we map servers and clients to computation cores in a manner shown in Fig. 12.

We use the first column as lock servers and other columns as clients. The lock server in row \(k\) serves the clients in the same row as a local server and works as the global server for locks numbered \(8n + k\). For the basic design

---

**TABLE 1**

<table>
<thead>
<tr>
<th>Type</th>
<th>Best Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hierarchical Lock</td>
<td>(3NS + 3(T - 1))</td>
<td>(6NS - 3MS)</td>
</tr>
<tr>
<td>Combined Lock</td>
<td>((2N + T)S + 2T - 1)</td>
<td>(6NS - 3MS)</td>
</tr>
<tr>
<td>Fair Lock</td>
<td>((2N + 3T - 1)S)</td>
<td>(6NS - 3MS)</td>
</tr>
</tbody>
</table>

---

Fig. 9. The fair lock can release the lock in a highly-contended situation.

Fig. 10. Use a local server to serve as a global server.

Fig. 11. Requests of locks owned by different global servers.
without a local server, the local server cores act for relaying messages. This is because the hardware communication is only supported within the same row or 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, we cannot implement 63 clients + 1 server due to the on-chip network constraints of SW26010.

Using 8 cores for locks sacrifices certain computational power, but benefits the synchronization. We will discuss this trade-off in Section 5.2, where the data show that 56 cores + mLock is better than 64 cores + SHM-Lock when the critical section accounts for more than 0.44 percent of total time.

When using multiple SW26010 processors for parallel computing, a CG (core group with 64 cores) 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 when using mLock.

**APIs and Usage.** We have implemented mLock as a library for developers. The core APIs and usage of mLock are demonstrated in Fig. 13 with C-style pseudocode. Leaving 8 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 embarras-sing parallel workloads and then switch to 56 cores for workloads that need synchronization. The switching is a simple state-setting and its time is negligible. Developers can use ML_Begin to assign roles to cores as servers or clients, then use ML_End to go back to the 64-core mode.

Inside the surrounding area from ML_Begin to ML_End, developers can use ML_lock and ML_unlock to acquire and free locks. Also, a more conservative way is to use 56 cores from the beginning to avoid potential load re-balancing among threads due to mode switching.

**Hardware Limitations.** For a CPE on SW26010, only the memory access to its 64 KB **Local Data Memory** (LDM) is fast. Limited by the small size of LDM, we only support locks with id 0 to 63. These locks use less than 1 KB LDM in total and other LDM is reserved for applications. In Section 5.4, we will see that 64 locks are enough for typical applications.

The inter-core communication APIs in SW26010 support sending and receiving 256-bit messages. We use 64 bits to specify the lock to operate on, 64 bits to encode the operation, 64 bits to save the waiting list of the chaining lock and reserve the remaining 64 bits. More details are elaborated in our previous work [12].

At the end of Section 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.

**Message Count Analysis.** Based on this implementation, we revisit the message count of approaches described in Section 3. In this discussion, we suppose that all the 56 clients are used, which means N = 56. On SW26010, a long-distance message passing is implemented by two short-distance message passing and 7 clients in the same row with the global server can communicate with the global server with one short-distance message passing, i.e., M = 7. The 8 cores in the first column serve as local servers, therefore T = 8. We summarize the message count of mLock implemented on SW26010 in Table 2.

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

$$C_{sw_{basic}} : C_{sw_{Hier_{best}}} : C_{sw_{Chain_{best}}} : C_{sw_{Fair_{best}}} = 1 : 0.53 : 0.38 : 0.43.$$  

(10)

### 5 Evaluation

#### 5.1 Methodology

We have performed a series of experiments to evaluate the performance of mLock. Experiments are done on the Sunway TaihuLight system. Since this work focuses on thread synchronization inside a process, all experiments are done using a single CG of SW26010. Four approaches, (1) the Basic

**TABLE 2**

<table>
<thead>
<tr>
<th></th>
<th>Best Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Lock</td>
<td>315S</td>
<td>315S</td>
</tr>
<tr>
<td>Hierarchical Lock</td>
<td>168S + 21</td>
<td>315S</td>
</tr>
<tr>
<td>Combined Lock</td>
<td>120S + 15</td>
<td>315S</td>
</tr>
<tr>
<td>Fair Lock</td>
<td>135S</td>
<td>315S</td>
</tr>
</tbody>
</table>

**Fig. 12.** Mapping computation cores to lock servers and clients.

**Fig. 13.** The APIs and usage of mLock.
design (Basic), (2) Hierarchical lock (Hier), (3) the combination of hierarchical and chaining (Chain), and (4) the Fair design (mLock) are evaluated. Note that the Fair design is implemented on top of the chain lock and includes all optimization techniques, so we use mLock to denote it.

All experiments for EMP lock approaches are done with 8 local server threads and 56 client threads. In Sections 5.2 and 5.3, all clients compete for the same lock. In Section 5.4, clients compete for multiple locks.

Note that we do not use shared-memory lock approaches as a baseline, since 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.

In Section 5.2, we design several micro-benchmarks to validate the design of EMP-Lock. Experiments in this subsection include latency, throughput, scaling performance, and message count for the above four approaches.

In Section 5.3, we evaluate the performance benefits of mLock on CRONO [13], a benchmark suite containing several multithreaded graph algorithms.

In Section 5.4, we analyze the approach of server reuse by comparing the 56-client design of mLock with a 49-client design. We first use micro-benchmarks to evaluate our approach, including latency, throughput, and message count. We then show the performance of server reuse on CRONO.

We use the average of three times measurement as the results for this section.

5.2 Validation

5.2.1 Different Density

Figs. 14 and 15 show the latency and throughput using four EMP-lock approaches respectively. In these experiments, threads are doing critical sections repeatedly. There is nothing inside the critical section, and the interval between two critical sections varies from 100 cycles to 100,000 cycles. Latency shown in Fig. 14 represents the average time cost from entering lock() to leaving unlock(). The throughput shown in Fig. 15 represents the number of critical sections completed within a period. The throughput is measured in Million Operations Per Second (Mops).

In a lightly contended situation (interval > 10000 cycles), all approaches have short latency, and the throughput is limited by the small demand for locks. In a highly contended situation, the benefit of reducing message count is obvious.

From the view of latency, Chain is 16.2× faster than Basic with an interval of 4,000 cycles and the speed of mLock and Chain is similar. To explain the sudden shoots up around 400 cycles for Hier in Fig. 14, 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.

Although the latency of Hier is much shorter than Basic, its throughput improvement is less impressive. Chain and mLock, however, have much higher throughput than Basic. With a 200-cycle interval, Chain and mLock have 3.41× and 3.49× improvement over Basic respectively. And the throughput of the shared-memory lock with 64 threads (Shm Lock) is much lower than mLock (1.03 Mops versus 29.96 Mops).

We use an example to show that 56 cores plus mLock can work better than 64 cores plus shared memory lock, for a program with a highly contended critical section. Supposing there is a program using mLock and it contains a critical section with 200 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

$$T = \frac{29.96}{1.03} S + \frac{56}{64} (1 - S) = 28.21 S + 0.875.$$  

We can have $T < 1$ when $S < 0.0044$. In other words, mLock is better than Shm Lock when the critical section consumes more than 0.44 percent of the time.

In the case of different intervals between critical sections, Chain can pass lock in the following three modes:

100 Cycles. The local server can always receive new requests from its clients before the lock is released from clients to the local server. Therefore, the lock is only passed among clients in that row and other clients are starved. In this micro-benchmark, clients release the lock ones they get it. However, for real applications, there are some workloads inside the critical section, so starvation may occur in a lower lock density.

200 Cycles to 2,000 Cycles. In this situation, all clients are in the waiting list of their local server when the local server receives the lock from the global server, so the local server will grant the lock to all clients. As the interval between critical sections is longer than the time of a lock passing chain of 7 clients, the local server will not receive new requests before the lock is released from its clients. Therefore, the local server will release the lock to the global server once the chain is ended. The actual lock passing route is the same as using mLock, but Chain is slightly slower as the local servers in Chain need to confirm that its waiting list is empty before releasing the lock.

More Than 3,000 Cycles. When the local server was granted the lock, only some of its clients are in the waiting list. Therefore, during the lock-passing among these clients, the local server may receive new requests from other clients.
and grants the lock to them in the next iteration. As a result, Chain uses fewer messages than mLock and is slightly faster. However, the lock is no longer the bottleneck of the whole system in this lightly contended situation, so mLock can achieve performance similar to Chain.

5.2.2 Scalability Analysis

Fig. 16 shows the scaling performance of the four approaches. We set the interval between critical sections to 200 cycles, and vary the client number from 1 to 56.

Generally, throughput increases with the number of clients with some drop-down points. As we discussed in Section 3, 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 Fig. 16 we can also see that, the scalability of Chain and mLock is better than the other two. With an interval of 200 cycles, Basic scales up to only 3 clients, while Chain and mLock scale up to 20 clients.

5.2.3 Message Count Analysis

To verify our analysis in Sections 3 and 4, we let each client do critical sections for 1,000 times, and record the message counts. The results are shown in Fig. 17 and Table 3. Message counts are listed for the reused server (RS) which serves as both a global server and a local server, local servers expect the reused one (LS), and clients (C), respectively. As mentioned in Section 4, Basic needs some cores to relay messages between the lock server and clients. We regard these cores as LS in this experiment.

The bars for Basic is symmetric. It is because that each lock operation involves a client-relay-server or a client-server communication path.

Message count for RS and LS is reduced a lot with the hierarchical lock. The chaining lock further reduces the message count for LS. However, fair lock increases the message count of RS and LS.

The message counts of four approaches are

$$\frac{C_{exp,basic}}{C_{exp,hier}} : \frac{C_{exp,chain}}{C_{exp,fair}} = 1 : 0.58 : 0.48 : 0.56.$$  \hspace{1cm} (12)

Comparing with Equation (10), we can see that the actual impact of these 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 Chain only reduces 17 percent messages of Hier with an interval of 200 cycles, its performance is much better than Hier (1.47 × for latency and 2.55 × for throughput).

Despite mLock increases message count by 17 percent over Chain, it outperforms Chain by 2.7 percent for latency and 2.4 percent for throughput. And for more complex benchmarks, mLock can have a larger speedup over Chain due to its better fairness. The results will be shown in Section 5.3.

5.3 Case Studies

In this section, we use the CRONO benchmark suite to evaluate the proposed techniques in mLock. The time complexity of benchmarks in CRONO varies greatly, so we use different graph sizes in different benchmarks, limiting the execution time between 100 milliseconds and 10 seconds. The description of each algorithm and the number of vertices we choose are listed in Table 4. TSP is run on a complete graph and other benchmarks are run on graphs with an average degree of 16.

Fig. 18 shows the speedup over the basic design in each benchmark. We can see that mLock has the best performance in the benchmark suite. From the view of geometric mean, mLock improves performance by 9.0 percent, while Hier and Chain decrease performance by 17.8 and 7.6 percent respectively.

COMM does not use locks. ASAP, BC, BFS, PR, and TSP are not lock-intensive benchmarks. Therefore, the lock-
passing strategy has little influence on the execution time of these benchmarks and the execution time difference of these approaches is within 3 percent.

In DFS, mLock is slightly slower than Chain, indicating that the improvement of fairness may harm the performance in some situations. It is because mLock passes more messages than Chain (4.0 percent in this case).

As mentioned in Section 5.2, Hier and Chain outperforms the basic design on micro-benchmarks. However, Hier and Chain are slower than Basic on CC, Tri, and SSSP. This is because to improve the locality, these two designs prefer to grant locks to clients belonging to the same local server and clients owned by other local servers can be suspended for a long time.

We further profile the execution time of SSSP and Tri to verify that idea. Fig. 19 shows the time consumption of critical sections and global synchronization (Barrier) of an SSSP run and a Tri run. The time consumption is measured for each core. The bars in Fig. 19 represent the average time of all client cores, and the error bars show the minimum and maximum values.

We can see that while Hier and Chain reduce the average time for critical sections over Basic, they increase the variance of it. In the same benchmark, the work inside each critical section is almost the same, so the large variance means that the latency for each client to get the lock varies greatly. As a result, although these two benchmarks have a uniform workload for clients respectively, different clients have a great difference in the time to finish their work, indicated by the large variance on barrier time. mLock, however, reduces the mean time for critical sections over basic and keeps the variance. Therefore, mLock has the best performance.

5.4 Server Reuse

mLock can leverage more computation cores (56 cores on SW26010) as clients while our previous work, pLock [12] without server reuse mechanism can only use 49 cores as clients, which can lead to a waste of computation resources. In this section, we will show the performance improvement brought by the approach of server reuse proposed in mLock.

In the following micro-benchmarks, the threads repeatedly select a mutex lock randomly, lock it and unlock it without doing anything in the critical section. In Fig. 20, there are eight available locks and the interval between two critical sections varies from 100 cycles to 100,000 cycles.

The definition of latency and throughput is the same as Figs. 14 and 15 respectively. In a lightly contended situation, the latency and throughput show a similar performance regardless of whether we enable server reuse. However, in a highly contended situation, the system with server reuse has much lower latency and higher throughput. With a 1000-cycle interval, server reuse reduces 56 percent of latency and brings a 1.82× improvement for throughput.

In Fig. 21, threads acquire random locks repeatedly with 200 cycles interval. When the number of available locks is small, throughput increases with available locks, because more threads can visit their critical sections simultaneously. When the number of locks is large, throughput decreases with the increase of available locks. mLock performs better when the lock acquisition has better locality, i.e., threads of the same local server want the same locks. In this experiment, lock_id is generated randomly, so the increase of available locks hurts the locality and reduces the throughput.

We let each of the 49 threads in the design without server reuse acquire 1,000 random locks and each of the 56 threads in mLock acquire 875 random locks, so that there are 49,000 lock requests in each run. There is a 200-cycle interval between two requests of one thread. We record the average message counts for the global server (GS) and local servers (LS) in the design without server reuse plus all servers in mLock (RS). Fig. 22 shows that the message count increases as the number of available locks due to the poorer locality. The message count of the global server in the design without server reuse grows significantly faster than its local servers and lock servers in mLock. When the number of available locks is large, the global server becomes the bottleneck, so in Fig. 21 we can see that, the design without server reuse has a bad performance when the number of available locks is large.

We then compare mLock with the design without server reuse on the CRONO benchmark suit. Each program is run
with 32 available locks (32L) and 64 available locks (64L) respectively. We use the same graph size as Section 5.3. The results are shown in Fig. 23.

The server reuse mechanism increases the number of working threads from 49 to 56, but for most benchmarks, the performance under different thread numbers are similar. It is because the bandwidth of global memory is limited in SW26010, while these benchmarks are memory bounded, so that the increase of working thread cannot improve the performance. In TSP, we save the intermediate results in the fast LDM of SW26010, so mLock can be faster than the design without server reuse.

The speedup on the CC benchmark and the SSSP benchmark is larger than $\frac{49}{46} \approx 1.14$, showing the effect of communication-balancing of mLock. These two benchmarks are quite lock-intensive, so in the design without server reuse, the single global lock server becomes the bottleneck of the whole system. However, in mLock, we distribute the workload of the global server to the 8 reused lock servers and thus reduce the maximum number of messages that need to be processed by each lock server.

As mentioned in Section 4, we only provide 64 locks. In general, fine-grained locks can bring more chance of parallelism and improve performance. However, Fig. 23 shows that the performance of the 64-lock version is similar to that of the 32-lock version, and we infer that using more than 64 locks will not bring better performance.

6 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 [17], [18], [19], [20].

Using locks to protect critical sections is a popular solution for synchronization. Many techniques can be used to implement locks. Spinlock [16] 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 [21], 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 spinlock with an exponential backoff algorithm [1] has much better performance than a naive implementation. Queue locks like MCS [2] and CLH [22] 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 a 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 heavily over-subscribed. Since queue locks and POSIX mutex introduce additional overhead, the most straight-forward spinlock has the best performance under no-contended or lightly-contended situations. QOLB [23] uses a hardware-supported queue and cache-line associated synchbits to organize waiting processes and critical data. With the hardware support, QOLB can accelerate lock transferring among processes on coherent cache architectures.

All the lock approaches above are trying to transfer the permission of executing critical sections 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. Therefore, researchers propose the 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) [24], [25] technique, threads that do not get a lock append 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 [9], service threads are dedicated and pinned to cores and have better cache performance. Some optimization in traditional locks is also be applied to delegation methods. Queue-delegation [26] uses 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 [25], [27]. HCLH [28] and Cohort-locks [29] are hierarchical locks, and SANL [30] 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], [32]. 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 [11], they often consider high availability and reliability, while locks for multi-thread programs put the performance (throughput and latency) in the first place. As a result, instead of using distributed locks directly on EMP processors, developers should carefully design new synchronization methods to meet performance requirements. EMP has been used to accelerate the request sending routine in RCL [4], and the performance is improved by 4.3 x. On Sunway TaihuLight [5], concurrent data classification obtains a speedup of 19.15 x after changing synchronization method from shared-memory based locking to EMP based delegation [7]. HybLock [4] and Dogan’s work [10] use EMP to accelerate multi-thread programs while preserving original lock interfaces. Their work illustrates the advantage of EMP over cache coherence on locking. Nevertheless, the usage of EMP mechanism in previous work is relatively straightforward. Our work shows that a lock mechanism especially designed for EMP can have further performance improvement over a simple EMP-based lock mechanism. The mLock approach improves the performance of EMP-based lock by up to 16.2 x over the basic approach.

There are also some alternative synchronization methods without explicit locking. Simple and common data structures like queue [33], stack [34], hash table [35], have been redesign to be lock-free. Universal construction approaches for wait-free or lock-free data structures have also been proposed [36]. 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) [37] is too slow thus is not practical [38]. Hardware transactional memory (HTM) is much faster than STM, while still can be further improved [15]. HTM-based programs have good performance with low conflict rates. 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 5. 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, mLock, 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” [17], [18], several tools [4], [30] have been proposed to help developers to choose the best synchronization solution according to program workload and hardware platform.

### Conclusion and Discussions

In this paper, we discuss the principles of using explicit message passing (EMP) interfaces to design and implement a mutex lock. We propose a series of techniques, including hierarchical lock, chaining lock, fair lock, and server reuse, to improve the performance and fairness of mutex locks. We implement these techniques on SW26010, a product processor with EMP and is used in the supercomputer Sunway TaihuLight. Experimental results show that our optimization techniques improve performance by up to 16.2x compared with the basic implementation.

The four techniques used in mLock: hierarchical lock, chaining lock, fair lock, and the idea of server reuse, can be generally used for any architectures that support EMP. For a new architecture that supports arbitrary pairwise communication, there are 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, computation cores can use cache to share the waiting list, and there is no need to pass a waiting list among client cores explicitly.

Allowing clients to pass locks to others may introduce potential security issues. As a temporal solution for security concerns, we can fall back to a centralized server-client solution by forcing lock servers always to return an empty waiting list. And a safe lock on EMP architectures remains as future work.

### Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by the National Key R&D Program of China under Grant 2016YFB0200100, in part by National Natural Science Foundation of China under Grant 61722208, in part by Beijing Natural Science Foundation under Grant 4202031, and in part by National Science Foundation under Grant CCF-1657333, Grant CCF-1717754, Grant CNS-1717984, and Grant CCF-1750656.
REFERENCES


[38] C. Cascaval et al., “Software transactional memory: Why is it only a research toy?” *Queue*, vol. 6, no. 5, 2008, Art. no. 40.

Xiongchao Tang received the BS and PhD degrees from Tsinghua University, Beijing, China, in 2013 and 2019, respectively. He is currently a postdoc with Tsinghua Shenzhen International Graduate School and Sangfor Technologies Inc. His major research interests include performance analysis and performance optimization for multi-core systems, and large-scale clusters.

Chen Zhang is currently working toward the undergraduate degree with the Department of Computer Science and Technology, Tsinghua University, Beijing, China. Her research interest include parallel computing.

Jidong Zhai received the PhD degree from Tsinghua University, Beijing, China, in 2010. He is an associate professor with the Department of Computer Science and Technology, Tsinghua University. His research interests include high-performance computing, compilers and performance analysis, and optimization of parallel applications. He is currently on the editorial board of the *IEEE Transactions on Parallel and Distributed Systems*. 
Xuehai Qian received the PhD degree from the Computer Science Department, University of Illinois at Urbana-Champaign, Champaign, Illinois. He is an assistant professor with the Ming Hsieh Department of Electrical Engineering and the Department of Computer Science, University of Southern California. He leads the ALCHEM Research Group.

Wenguang Chen received the BS and PhD degrees in computer science from Tsinghua University, Beijing, China, in 1995 and 2000, respectively. He was the CTO in Opportunity International Inc. from 2000 to 2002. Since January 2003, he has joined Tsinghua University. He is a professor with the Department of Computer Science and Technology, Tsinghua University. His research interests include parallel and distributed computing and programming model.

Yong Jiang received the BS and PhD degrees in computer science from Tsinghua University, Beijing, China, in 1998 and 2002, respectively. He is a professor with Tsinghua Shenzhen International Graduate School. His research interests include the architecture of network, mobile network, network application, and the next generation Internet.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.