Simulation Modelling Practice and Theory 100 (2020) 102067

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat

An efficient multi-threaded memory allocator for PDES applications

Tianlin Li, Yiping Yao, Wenjie Tang*, Feng Zhu, Zhongwei Lin

National University of Defense Technology, China

ARTICLE INFO

Keywords:
Multi-threaded
Memory allocator
PDES

ABSTRACT

Parallel Discrete Event Simulation (PDES) aims at providing high performance solutions for simulation with large and complex models. Like traditional multi-threaded applications, PDES applications are also confronted with large overhead produced by memory allocation. Especially, complex models and frequent inter-thread communications in PDES applications will lead to large amounts of memory allocation and deallocation requests. Inter-thread communications are implemented by sending and receiving events based on shared memory. Hence, an efficient memory allocator has great impact on the performance of such applications. Current well-known allocators are dedicated to be scalable and lock-free. However, they ignore the difference between thread local and shared memory and manage them in the same way, which cannot make good use of their own characteristics. To solve this problem, this paper proposes a high efficient multi-threaded memory allocator named HMalloc, which innovatively separates thread local memory from shared memory. Thus, local memory allocation and deallocation will not lead to false sharing and lock contention. Moreover, coalescence free is applied to free local memory blocks. Furthermore, a flag-based shared memory management method is proposed to achieve lock-free shared memory allocation and deallocation. Experimental results show that HMalloc can achieve significant performance improvement in both traditional memory allocation benchmarks and typical PDES benchmarks when compared with existing well-known memory allocators.

1. Introduction

Parallel Discrete Event Simulation (PDES) [18] is widely used in system evaluation and analysis in many areas including computer and telecommunication systems, biological networks, military war gaming, online games, and operational management. PDES aims at providing high performance solutions for simulations with large and complex models. Traditional PDES engines are mostly process-based (MPI), such as ROSS [3], WarpIV [28], GTW [23]. The high communication overhead and latency among processes will significantly decrease the application performance. With the development of multi-threading technology and multi-core platform, thread-based PDES simulators have become the mainstream simulators, because of low communication overhead and latency between threads. For example, ROSS-MT simulator [6,10,11] is the thread-based version of ROSS simulator, and can achieve several times of performance improvement when compared with ROSS.

A multi-threaded PDES application usually contains many simulation entities, and all these entities will be distributed to multiple threads. Interactions between entities will lead to frequent inter-thread communications. The models and frequent inter-thread

* Corresponding author.
E-mail address: tangwenjie@nudt.edu.cn (W. Tang).

https://doi.org/10.1016/j.simpat.2020.102067
Received 12 September 2019; Received in revised form 6 January 2020; Accepted 7 January 2020
Available online 15 January 2020
1569-190X/

communications will produce large amounts of memory allocation and deallocation operations. Hence, an efficient memory allocator will significantly influence the performance of PDES applications. Inter-thread communications depend on sending and receiving events based on shared memory. In general, a thread will not generate multiple events at one time and pass all of them to another thread. At a certain time or periodically, a thread will generate one event and send to another thread or generate multiple events and send to multiple threads. Each event has a timestamp and events are executed in the ascending order of timestamp. There is a global virtual time (GVT) among all threads, and an event will be freed if its timestamp is less than the GVT. A traditional well-known multi-threaded memory allocator may not be suitable for multi-threaded PDES applications, because PDES applications are different from traditional multi-threaded applications. The main difference is that the inter-thread communications in PDES applications are more complex and unpredictable. A thread in PDES applications may communicate with any other threads. However, the inter-thread communications in traditional multi-threaded applications are simpler. For example, a car entity may interact with many car entities in traffic simulation. These car entities may be distributed to multiple threads, which will lead to complex communications among threads. Frequent and complex inter-thread communications will bring challenges to multi-threaded allocators, such as false sharing and fierce contentions on shared memory, which will lead to uncertain performance improvement when a multi-threaded memory allocator is used for PDES applications.

Memory allocation and deallocation are the most basic operations in application programs, such as malloc and free functions in C or C++ applications. Memory allocators for single-thread or single-process applications have been very mature. With the emergence of multi-threading technology, most parallel simulation applications are built based on multi-core or many-core platforms. Therefore, multi-threaded memory allocators are necessary to provide efficient and scalable memory management solution. Allocation latency, access locality, and performance scalability have become three most important standards for multi-threaded memory allocators [21]. Hence, current multi-threaded memory allocators are committed to be scalable, high efficient and lock-free. Lock contention will happen when two threads operate the same memory block simultaneously. Large amounts of lock contentions will significantly decrease application performance and severely reduce scalability. Hence, existing multi-threaded memory allocators devote to eliminating lock contentions. Besides, false sharing and memory blowup should be avoided.

Currently, there exist multiple mature multi-threaded memory allocators, such as Hoard [7], SSMalloc [21] and SFMalloc [25]. Some of them have implemented lock-free and scalable memory allocation and deallocation. The core method of mainstream multi-threaded memory allocators is that they use local heap to perform thread isolation to achieve scalability and establish a lock-free queue or stack to eliminate lock contentions on shared memory. Each thread corresponds to a local heap, and all threads share one global heap. One thread only performs memory allocation and deallocation in its own local heap, reducing lock contentions on memory allocation as much as possible. When the available memory space in a local heap is not enough, the thread will first obtain memory from global heap. If the available memory blocks are exhausted in global heap, the thread will request virtual memory from OS. For example, there are multiple local heaps and one global heap in hoard and SSMalloc. Lock-free allocation and deallocation of shared memory is accomplished by establishing a lock-free memory block queue or stack, such as SFMalloc. Lock-free lists or queues are mostly implemented by compare and swap (CAS) atomic instructions, such as NBMalloc [2], SSMalloc, SFMalloc etc. False sharing is avoided by aligning memory block sizes to the multiples of cache line size. Furthermore, unbounded memory blowup problem is eliminated by remote free method. Remote free means that a shared memory block can be reused by its owner thread when it is freed by another thread.

Though current allocators are mature and efficient, there is still some room for improvement. They all ignore the difference between thread local memory and shared memory. In multi-threaded applications, memory blocks can be classified into two types, local or shared. Thread local memory represents the memory blocks which can only be accessed by its owner thread, while shared memory can be accessed by two or more threads. In the lifetime of a thread, it will allocate both local and shared memory blocks. Local memory blocks are used to satisfy internal storage requirements and shared memory blocks are responsible for transmitting messages to other threads. The key difference is that a local memory block can only be freed by its owner thread, while a shared block can be freed by different threads. However, current allocators manage thread shared memory and local memory in the same way and often store local memory blocks and shared memory blocks together. However, if inter-thread messages and thread local data are stored together, the memory requested size of local memory blocks need to be aligned to multiples of the cache line size to avoid false sharing, which will lead to fragment and waste of memory space. On the contrary, if local memory blocks are separated from shared memory blocks, there will be no lock contentions in local memory allocation and deallocation. Moreover, current allocators depend on CAS atomic instructions to implement lock-free shared memory allocation and deallocation. However, frequent CAS operations will decrease the performance of memory management. Even though, a completely lock-free memory allocator does not exist, because multiple threads will also contend on OS kernel when they perform virtual memory (VM) management operations at the same time. Therefore, an excellent memory allocator should also minimize the frequency of requesting memory from OS.

If thread local memory blocks are separated from shared memory blocks, the above mentioned problems can be solved well. This paper proposes a hybrid, scalable, and lock-free multi-threaded memory allocator named HMalloc, aimed at providing a more efficient memory management solution for multi-threaded applications, including PDES applications. To make full use of the difference between thread local and shared memory, HMalloc innovatively separates local memory from shared memory and manage them in different ways. Based on this, HMalloc is implemented to be scalable, synchronization-free and lock-free. False sharing and memory blowup are avoided too. The contributions of this paper are summarized as follows.

(1) Thread local memory and shared memory are stored and managed separately. Hence, there is no need to consider the issues, such as false sharing and lock contention, in local memory allocation and deallocation.
(2) Coalescence free is adopted to free local memory blocks. When the allocation number and deallocation number in a large memory

block are equal, the large memory block will be cleared. Compared with traditional deallocation process, the process is greatly simplified.

(3) A flag-based shared memory management method is proposed to implement lock-free allocation and deallocation for shared memory, achieving asynchronous remote free. Each shared memory block reserves 8B memory space at the starting position to store a flag value. The flag represents whether the memory block is occupied or freed. When a shared memory block is allocated, the flag is set to 1 by the owner thread. When another thread frees it remotely, the flag is reset asynchronously, which does not require atomic operations. A flag occupies 8 bytes is to ensure byte alignment.

The rest of this paper is organized as follows. Section II describes the background and the related work. Section III presents the framework of HMalloc. The experimental results and analysis are provided in Section IV. Finally, Section V concludes the paper.

2. Background and related work

This section will first introduce multi-threaded parallel discrete event simulation, and then lists the related work of multi-threaded memory allocators.

2.1. Multi-threaded parallel discrete event simulation

The concept of PDES is proposed in 1990 [18]. Since then, PDES has attracted large amounts of researches, because of its high performance. At first, PDES simulators are all process-based, such as ROSS, YH-SUPE [31], depending on Message Passing Interface (MPI) or TCP/IP to implement parallelism. However, the communication overhead and latency between processes is high. The emergence of multi-core platform and multi-threading technology provides a more efficient solution for PDES applications. In order to enhance PDES performance, some researchers have designed multi-threaded PDES engines. Chen et al. [12] proposed a multi-threaded PDES implementation that uses a global event scheduling mechanism. Vitali et al. [22] proposed a different multi-threaded PDES simulator that employed a load-sharing scheme. Park and Fishwick [9] designed a GPU-based application framework which could support discrete-event simulation. Tang et al. [26,27] also proposed a hierarchical PDES kernel for GPU and multi-core platform. However, all these simulators focus on optimizing the event scheduling or workload balance of PDES applications. Memory allocation and deallocation in these simulators still depends on the thread-safe allocator in glibc.

Among multi-threaded PDES simulators, ROSS-MT is a very representative one. Here, we take ROSS-MT as an example to introduce the mechanism of thread-based PDES simulator. Fig. 1 shows the simplified architecture of ROSS-MT [11]. In a multi-threaded PDES application, there are usually many simulation entities. Logical process (LP) is a terminology which is used to represent a simulation entity, such as a car in traffic simulation, and thread is the Process Element (PE). All LPs are mapped onto multiple threads in sequential or Round-Robin (RR) ways. In process of simulation execution, each LP will produce events and send events to other LPs, and state updates of LPs depend on scheduling events. When the sender LP and receiver LP are mapped onto different threads, inter-thread communications will happen. One thread communicates with another thread by sending and receiving events. Each thread maintains an output event queue to store events that will be sent to other threads, and an input event queue to store remote events from other threads. These two queues only store the address of the events, because threads share the same address space. After one thread push the address of an event into input event queue of another thread, the event sending is finished. In ROSS-MT, a free event queue is pre-allocated before the simulation starts. When a thread needs to create a new event, it will grab one from the free event queue and initialize it. Moreover, an event will be pushed into the processed event queue when the event is processed. When global virtual time (GVT) is updated, the processed events whose time stamps are less than GVT will be recycled and

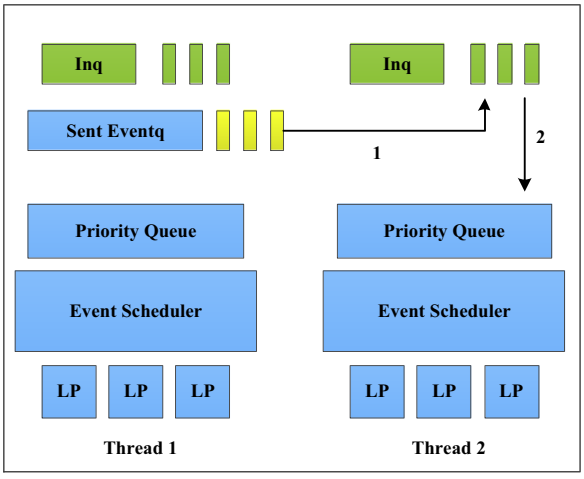


Fig. 1. ROSS-MT architecture [11].

pushed into the free queue. Multi-threaded PDES applications are different from traditional multi-threaded applications. There are frequent time synchronizations among threads. Besides, the inter-thread communication is more complex and unpredictable, which will affect the performance of memory allocator. Hence, a traditional memory allocator may not be suitable for multi-threaded PDES applications.

In this article, we focus on designing a more efficient memory allocator for multi-threaded applications, including PDES applications. To compare our proposed memory allocator with other multi-threaded memory allocators, the generation and fossil collection of events are accomplished by memory allocation and deallocation operations. That is to say, producing an event needs memory allocation to obtain memory space for the event. Besides, the event should be freed by the allocator when a processed event is recycled. Here, the reason why we need a dynamic allocator is that the length of pre-allocated event queue is difficult to determine for different PDES applications, because there are usually many kinds of simulation entities. Hence, there is no queue length that can be suitable for all applications. Irrational length of event queue will significantly reduce the application performance. Therefore, an efficient allocator is necessary for PDES applications. Events are used as mediums to transmit messages between LPs. The event sizes depend on detail models. Usually, the normal size of an event will not be very large. For example, the size of an event is 8 bytes in Phold, and the event size is 36 bytes in PCS benchmark.

2.2. Multi-threaded memory allocators

With the development of multithreading technology, scalable and efficient multi-threaded memory allocators have attracted a lot of attention. There have been many excellent multi-threaded memory allocators, such as Hoard, TCMalloc [8], SFMalloc, SSMalloc, XMalloc [29], and so on.

Hoard, proposed by Berger et al., is a typical scalable memory allocator for multi-threaded applications [7]. Each thread corresponds to a local heap and all thread share a common global heap. Once the mapping relationship between a thread and a local heap is determined, the thread will only access the corresponding local heap. The basic memory unit that Hoard allocates from OS is superblock, and all superblocks are of the same size and fixed at 64 KB. All blocks in a superblock are in the same size class. When the requested size exceeds half the size of a superblock, the request will be satisfied by allocating memory from OS directly. Each superblock maintains an empty ratio. When the ratio is exceeded, it will be freed to the global heap and can be reused by other threads to prevent memory blowup. When multiple threads allocate memory simultaneously, the requests will be satisfied by different superblocks. However, Hoard is not lock-free. When multiple threads access the freed superblock list in global heap at the same time, lock contentions will happen. Besides, heap contention will also happen when a thread allocates memory from one heap and other threads also free memory blocks which belongs to the same heap. Typically, producer-consumer application will lead to many heap contentions in Hoard. Large amount of lock contentions will degrade the performance.

Dice and Garthwaite [5] introduced remote-free mechanism to improve Hoard. Lock contentions on local heap are eliminated. Lock operations are used only for operations on freed superblock lists in the global heap. Compared to Hoard, the number of lock operations is reduced. Michael implemented a lock-free dynamic memory allocator by introducing CAS atomic instructions [15]. On the basis of Hoard, CAS instructions are used to implement lock-free freed superblock list. Leite et al. also proposed a lock-free allocator, which still depends on CAS atomic instructions to implement the lock-free memory management process [20].

Cho et al. proposed a memory allocator for multi-core embedded devices [30]. Similarly, the allocator also distributes a thread private heap for each thread, and all threads share a thread global heap (TGH). The TGH contains a global block (GB), a lock-free circular queue (LCQ), and memory block informations (MBIs). GB is also composed of some fixed-size memory blocks. LCQ is used to store the freed MB. LCQ is lock-free by means of CAS atomic instructions. Each MB maintains an element called free_count to record the number of freed objects, and free_count is initially set to the maximum number of objects contained in the MB. Whenever a thread triggers a free operation, free_count is decremented by 1 through atomic operations. When free_count is reduced to 0, the MB will be placed in the LCQ for subsequent memory requests.

Liu et al. designed a memory allocator named SSMalloc. Similar to Hoard, each thread owns a private heap, and all threads share the global heap. Each private heap owns several fixed-size memory chunks. A memory chunk contains many objects of the same size class. Global heap maintains a lock-free freed chunk list. When empty memory chunks in a thread local heap are superfluous, some of them will be recycled to this list for other threads to use. The list is also implemented to be lock-free with CAS atomic instructions. SSMalloc mainly achieves the following three improvements: (1) reducing the memory management overhead through minimized critical path; (2) Reducing the VM management system calls; (3) adopting lock-free and most wait-free algorithms to optimize the memory management process.

CAS atomic instructions are often used to implement lock-free list or queue in current multi-threaded allocators. However, CAS will lead to some problems [29]. For example, there exists a critical section which begins by reading a shared variable and the variable is updated by CAS instructions. A thread tries to update the variable, if the update CAS instructions fail, meaning that the variable has been modified by another thread. The thread needs to repeat the critical section. What's worse, one thread updates the variables first and other lagging threads will repeat the section when multiple threads execute this critical section simultaneously. Hence, CAS may leads to a lot of extra overhead. Huang et al. proposed a two level allocation algorithm, which can avoid the problems induced by CAS instructions. However, this algorithm is only suitable for SIMD applications. Besides, each superblock header contains a number of flag bits to represents whether the basicblocks in the superblock are occupied or freed. The updates of these flag bits also depend on CAS instructions. Springer et al. also proposed a parallel memory allocator [16]. However, this allocator is only responsible for SIMD applications on GPUs.

Areias et al. proposed a multi-threaded memory allocator based on table space organization [13]. Similar to the superblock in

Hoard, the unit of memory allocated from OS is page. Each page is divided into some fixed-size contiguous memory blocks. Each memory page is responsible for storing only one type of data structure. A page can be used as a thread local page or global page. Through table space organization, these pages are connected as a list for memory allocation and deallocation. When multiple threads access a global page, lock contention will happen.

Seo et al. proposed a lock-free and mostly synchronization-free dynamic memory allocator named SFMalloc. In SFMalloc, superpage is the basic unit of memory block allocated from OS. The size of a superpage is 64 OS pages, which is 256KB. A superpage consists of a header and several page blocks (PB). The page blocks may be of different sizes, but they are all multiples of 4KB. The superpage header occupies the first page and the page blocks occupy the remaining 63 pages. Each PB also has its own header (PBH), similar to superblock. PB is divided into fixed-sized sub-blocks and the size is determined by memory requested size. Each thread corresponds to a local heap and there is no global heap. A global freed superpage list is responsible for storing the superpage freed by each thread. In order to support remote free, a remote list is added to each PBH, and a remote PB list is added to the superpage header. They are also implemented through lock-free stacks. When a thread frees a memory block allocated by another thread, it will push the memory block to the remote list in the PB of the owner thread. This algorithm has some disadvantages. First, it uses a large number of data structures, which will consume more memory space. Second, the critical path of the algorithm is long. Moreover, the lock-free stack needs large amount of atomic operations.

In summary, existing multi-threaded memory allocators usually ignore the difference between thread local memory and shared memory, leading to more memory resource are wasted during the local memory allocation and deallocation. Besides, the implementation of lock-free memory management must rely on atomic instructions. These allocators do not utilize the asynchronous characteristic of memory allocation and deallocation, which is that the allocation of a memory block must precede the deallocation.

3. Framework of HMalloc

This section mainly presents the design and implementation of HMalloc. Fig. 2 illustrates the architecture of HMalloc. The most important design goal of HMalloc is also scalability and high performance, and lock contention is the biggest challenge for scalability. HMalloc creatively separates thread local memory from shared memory, and it also draws on the advantages of other excellent allocators. When thread local memory is separated from shared memory, the allocation and deallocation of thread local memory is obviously lock-free and will not lead to false sharing. To achieve lock-free shared memory allocation and deallocation, a flag-based memory management method is proposed. Based on this method, remote free on shared memory blocks becomes lock-free and

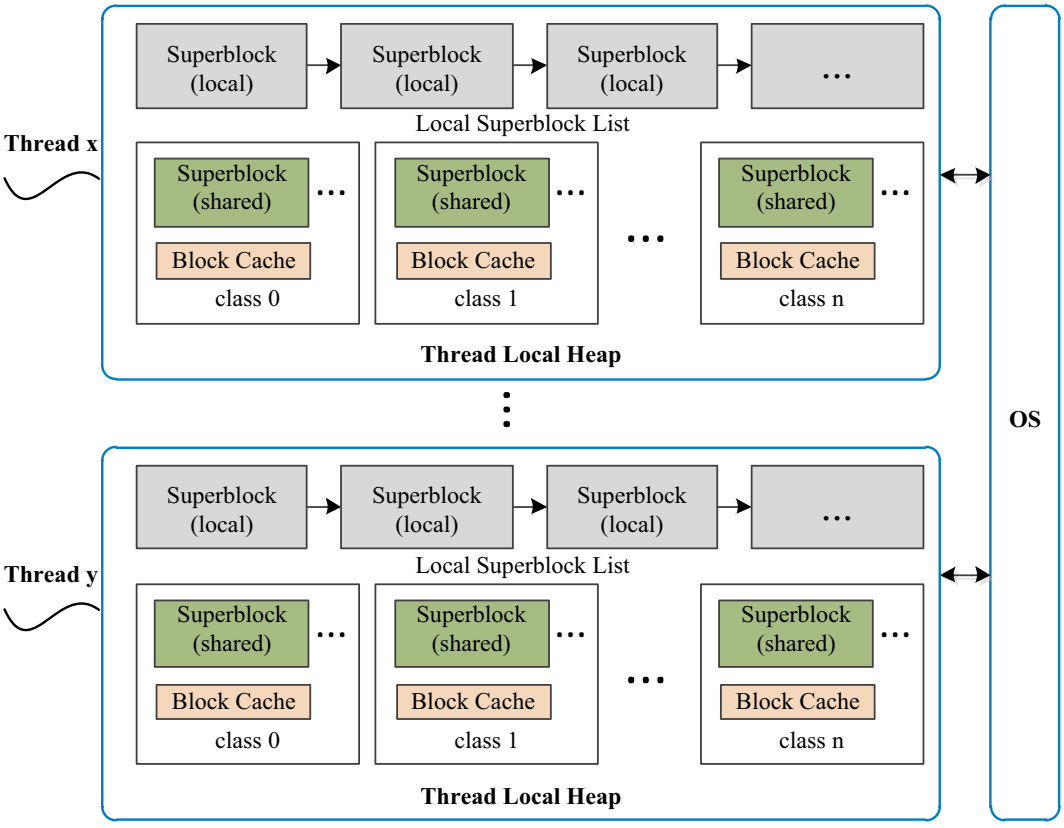


Fig. 2. HMalloc framework.

asynchronous without atomic instructions. To be efficient, HMalloc adopts coalescence free method to minimize the critical path length of local memory allocation and deallocation, and cache method to reduce the allocation overhead of shared memory. Hence, lock contentions will happen only when multiple threads allocate memory from OS simultaneously, which is inevitable. Besides, non-fixed-size memory superblocks are introduced to accommodate various memory requests and reduce the frequency of allocating memory blocks from OS.

As shown in Fig. 2, each thread corresponds to a local heap, which reduces lock contentions between threads as many as possible. A local heap is only responsible for the memory allocation and deallocation of its owner thread. Different from other multi-threaded allocators, there is no global heap. In other allocators, global heap is used to store superfluous freed memory blocks of some threads and these memory blocks can be reused by other threads. However, global heap will lead to additional lock contentions. Take Hoard as an example, a superblock will be moved to the global heap when its fill level is low. Hence, Hoard needs to maintain the empty ratio of each superblock and fill level of each local heap. Lock contentions on local heap will happen when two threads free the memory blocks in same local heap at the same time. Besides, contentions on global heap will also happen when multiple threads request superblocks from the global heap simultaneously. To avoid thread lock contentions on global heap, HMalloc abandons the global heap. When current superblock is exhausted, HMalloc will first traverse the old superblocks in local heap to find empty memory blocks. If all superblocks in the local heap are exhausted, the thread will generate new superblocks by requesting virtual memory from OS directly. A memory block will remain in the local heap until its owner thread terminates, because a thread will periodically allocate and free large amounts of memory blocks in some cases, such as PDES applications. Besides, abandoning global heap will not lead to memory blowup. Assuming that a thread will consume a maximum memory space of size M in its lifecycle, an N-thread application will cost at most N*M memory size in the worst cases when it adopts HMalloc as its memory allocator. Moreover, HMalloc adopts remote free method to free shared memory blocks. Hence, memory blowup will not happen even there is no global heap. However, this will sacrifice the memory space efficiency. When a thread is terminated, a new thread can inherit its local heap to reuse its freed memory space.

Like Hoard, superblock is the basic unit of memory allocated from OS. There are two types of superblocks, local and shared. Local superblock is responsible for thread local memory allocation and shared superblock is used for shared memory allocation, so thread local memory is separated from shared memory. Then, a local superblock can only be accessed by its owner thread, while a shared superblock may be accessed by multiple threads. A local superblock only performs memory allocation and deallocation requests from its owner thread, while a shared superblock can perform allocation and deallocation requests from multiple threads. Non-fixed-size superblocks are introduced. Unlike the fixed-sized superblock or superpage in other memory allocators, the sizes of superblocks in HMalloc are not fixed, aims at accommodating various memory requests and minimizing the frequency of requesting memory from OS directly. Besides, a thread local superblock can contain many blocks of different sizes. However, a shared superblock contains a number of sub-blocks of the same size. Because the message size between two threads, such as the event size in PDES application, is usually fixed-size. However, there may be many different sizes of thread local data structures.

Each local heap consists of a local superblock list (LSL), a number of shared superblock lists (SSL) and block cache lists (BCL). When the memory space of a local superblock is exhausted, it will be inserted into LSL. Shared superblocks are classified into multiple size classes, and the size class of a shared superblock is determined by the size of its sub-block. Shared superblocks of the same size class are inserted into the same SSL list. Besides, a SSL list corresponds to a block cache list. A Block cache list is used to store the addresses of newly freed shared memory blocks, and the sizes of the blocks should be same. In order to avoid memory blowup in case of producer-consumer pattern, the length of block cache list is limited. When a shared block needs to be freed, the allocator first checks whether the block cache list is full. If the list is full, the block will be freed by our proposed flag-based memory management method. Otherwise, the address of the shared chunk will be inserted into block cache list. The cache mechanism can greatly enhance the performance when a thread executes frequent free/malloc pairs.

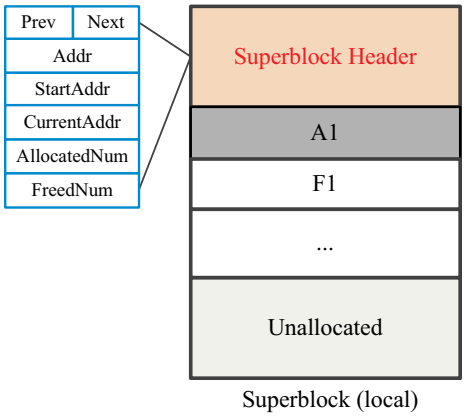


Fig. 3. Thread local superblock.

3.1. Local and shared superblocks

Fig. 3 represents the data structure of a local superblock. A local superblock only performs memory allocation and deallocation requests from its owner thread. Hence, false sharing will not happen, and any allocation and deallocation operations on such superblocks are lock-free. As described above, the sizes of local superblocks in HMalloc are not fixed. This is different from traditional multi-threaded memory allocators. For example, the superblock size in Hoard is fixed to 64 KB and the superpage size in SFMalloc is 256 KB. To better satisfy memory requests of different sizes, HMalloc adopts two sizes of local superblocks, 64 KB and 4 MB. The reason why we choose these two sizes is based on previous researches. Usually, a local superblock of 64 KB per thread is enough to satisfy the memory allocation requests in a light-weight multi-threaded application. When a local superblock of 64 KB is exhausted, a new local superblock of 4 MB will be allocated from OS to satisfy the subsequent memory allocation requests. Through introducing larger superblock, HMalloc can efficiently reduce the frequency of allocating memory from OS, which may reduce lock contentions in OS kernel.

A local superblock consists of a superblock header and many sub-blocks of different sizes. In traditional allocators, a superblock is divided into many blocks of the same size, such as Hoard, SSMalloc. However, a thread may produce many local memory allocation requests of various sizes in its lifecycle. If a superblock only contains fixed-size sub-blocks, the allocator needs to allocate a new superblock for a new memory allocation request in the worst case, which will lead to waste of memory space. To tackle this problem, HMalloc adopts superblock with variable sub-block sizes as the local superblock. The sizes of sub-blocks in a local superblock are not fixed, and the sizes depend on the corresponding memory requested sizes.

The superblock header maintains the key information of its owner superblock, and local superblocks in the same heap are connected into a doubly linked list with headers. Moreover, it stores the starting address of the superblock (Addr), and the number of allocated blocks (AllocatedNum) and the number of freed blocks (FreedNum). Because the sizes of sub-blocks are different, the superblock header does not record the address of each sub-block. It only records the address of the first sub-block (StartAddr) and the starting address of the unallocated block (CurrentAddr). Hence, CurrentAddr can be used as the return address for the next memory allocation request directly, and it should be updated based on the requested size when a memory allocation is finished. AllocatedNum records the number of allocated blocks, and FreedNum records the number of freed blocks. Coalescence free method is introduced to free the memory blocks in local superblocks. When AllocatedNum equals to FreedNum, coalescence free will be triggered. Then the superblock will be cleared and all the elements in the header will be reset. Among them, AllocatedNum and FreedNum will be set to 0 and CurrentAddr will be set to StartAddr. Through updating these elements, thread local memory allocation and deallocation requests can be satisfied.

Fig. 4 shows the structure of thread shared superblock. Thread shared superblock is responsible for performing shared memory allocation and deallocation. Hence, the sub-blocks in a shared superblock will be accessed by different threads. Moreover, the size of a shared superblock can also be 64KB or 4MB. The motivation is same to that of local superblock.

In multi-threaded applications, especially in PDES application, shared memory blocks are mainly used to store messages or events between threads, and the sizes of messages or events are usually fixed. Hence, all the sub-blocks in a shared superblock should be of the same size, and the size should be a multiple of the cache line size. The aim is to avoid false sharing between threads. At the same time, the shared memory requested sizes should also be aligned to multiples of the cache line size. Hence, shared superblocks can be classified into multiple size classes according to the sub-block sizes, class 0 is 64B, class 1 is 128B and so on. When a shared memory request occurs, only the superblock of the same size class can perform the allocation. Shared superblocks of same size class are stored in a list named Shared Superblock List (SSL). When a shared superblock is exhausted, it will be inserted into SSL list. Then, HMalloc will first traverse SSL list to obtain a superblock which has enough freed chunks, and the superblock becomes the new current superblock.

Similarly, the superblock header is responsible for connecting shared superblocks of same size class into a list. Besides, the header also contains the starting address of the superblock (Addr), the address of the first sub-block (StartAddr), the total number of sub-blocks (TotalObjects), the sub-block size (ObjectSize). Since the sizes of all sub-blocks in a shared superblock are all same, the address

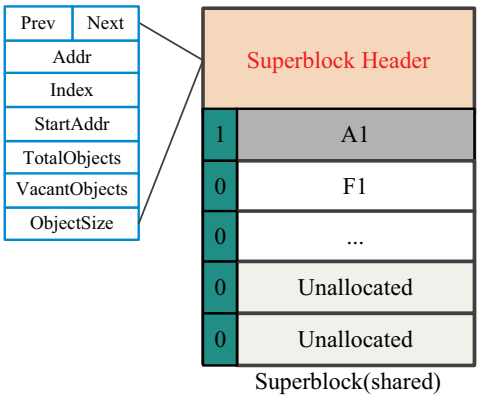


Fig. 4. Thread shared superblock.

of each block can be inferred according to ObjectSize and its ranking. VacantObjects records the number of objects that have not been allocated so far. Initially, VacantObjects equals to TotalObjects. When an object is allocated, VacantObjects will be subtracted by 1 until 0. On the contrary, VacantObjects will not be increased by 1 when a block in the superblock is freed. The reason is to avoid lock contentions on VacantObjects, because multiple threads may free blocks in the same superblock simultaneously. Element "Index" is an integer and set to 0 initially. Index is used to determine whether a shared superblock has been exhausted. When a memory request comes and VacantObjects in current shared superblock has become 0, HMalloc will first check whether the Index-th block has been freed. If the Index-th block is freed, HMalloc assumes that the superblock is still empty, and the block can be used to satisfy the allocation. After the allocation, Index should be increased by 1 for next use. Otherwise, HMalloc assumes the superblock has been exhausted and begins to find a new empty superblock.

To achieve lock-free shared memory allocation and deallocation, a flag-based lock-free shared memory management method is proposed. Different from the traditional flag bit, each sub-block in the shared superblock reserves 8 bytes to store the flag value. The address of the flag is the starting address of its owner memory block, and the return address for memory allocation can be obtained by adding 8 bytes to the starting address. The flag indicates whether the memory block is occupied or freed. There are two reasons why a flag occupies 8 bytes, one is that the OS owns a 64-bit address space, and the other is to ensure byte alignment. Since the size of each shared memory block should be aligned to the multiple of the cache line size (64 bytes) to avoid false sharing, 8 bytes per flag usually will not produce additional memory cost. When a memory block is allocated, the corresponding flag will be set to 1, and reset to 0 when the block is freed by another thread. Obviously, the allocation of a shared memory must precede the deallocation. Hence, the flag-based remote free method is synchronization-free and lock-free. After the flag value of a shared memory block is reset, the block can be reused by its owner thread for later memory requests, which can avoid the memory blowup problem.

3.2. Allocation process

Fig. 5 shows the thread local memory allocation process. Thread local heap needs to be obtained first when its owner thread requests a local memory block (line 1). Next, the requested size sz needs to be aligned to a multiple of 8B (line 2), which ensures the byte alignment. If the requested size sz is larger than the half size of current superblock, the memory allocation request should be satisfied by allocating memory from OS directly (lines 16–18). Otherwise, the request will be handled by the local superblocks in the local heap (lines 4–15). If the unallocated space of current superblock is larger than the requested size, current superblock is able to satisfy the memory request. CurrentAddr can be used as the return address directly, and then CurrentAddr and AllocatedNum should be updated for the next allocation request (lines 4–7). If not, current superblock will be inserted into local superblock list for later use and HMalloc will first traverse LSL to obtain a suitable local superblock whose available space is larger than the requested size (lines 9–10). If no suitable superblock is found, a new superblock should be allocated from OS and replace the old current superblock (lines 11–13). Then, the memory allocation can be satisfied by the new current local superblock and the elements in superblock header should be updated (lines 14). Obviously, there is no lock contention in thread local memory allocation process. Besides, there is no need to worry about false sharing.

Algorithm 1: Thread local memory Allocation Process

```
Data: sz, the memory requested size
Data: tlh, thread local heap
Data: current, the current local superblock in tlh
Data: LSL: local superblock list
Result: ptr: the address of allocated memory block
1   tlh ← getLocalHeap();
2   sz ← align8(sz);
3   if sz < (current.size)/2
4       if AvailableSpace(current) > sz
5           ptr ← current. CurrentAddr;
6           CurrentAddr ← CurrentAddr + sz;
7           AllocatedNum++;
8       else
9           Insert current into LSL;
10          current ← traverse LSL to obtain an empty superblock;
11          if current == NULL
12              current ← GetNewSBfromOS();
13          end
14          same to Lines 5-7;
15      end
16  else
17      ptr ← allocate memory from OS;
18  end
19  return ptr;
```

Fig. 5. Thread local memory allocation process.

```
Algorithm 2: Thread shared memory Allocation Process
Data: sz, the memory requested size
Data: tlh, thread local heap
Data: current, the current shared superblock in tlh
Data: SSL: shared superblock list
Result: ptr: the address of allocated memory block
1   tlh ← getLocalHeap();
2   sz ← align64(sz + 8);
3   if sz < 32 KB
4       class ← SizeToClass(sz);
5       current ← getCurrentSuperblock(class);
6       if block cache list is not empty
7           ptr ← obtain an element from block cache list;
8           return ptr;
9       end
10      if current. VacantObjects > 0
11          ptr ← current. CurrentAddr;
12          CurrentAddr ← CurrentAddr + ObjectSize;
13          VacantObjects --;
14      else
15          ptr ← the Index-th block in current superblock;
16          if ptr == NULL
17              Insert current into SSL of class;
18              current ← traverse SSL to obtain an empty superblock;
19              if current != NULL
20                  ptr ← current. StartAddr + Index* ObjectSize;
21                  Index ++;
22              else
23                  current ← GetNewSBfromOS(class);
24                  same to Lines 11-13;
25              end
26          end
27      end
28      double * flag ← ptr;
29      flag = 1;
30      ptr ← ptr + 8;
31  else
32      ptr ← allocate memory from OS;
33  end
34  return ptr;
```

Fig. 6. Thread shared memory allocation process.

Fig. 6 shows the thread shared memory allocation process. The allocation process is different to that of local memory. Thread local heap needs to be obtained first when a thread performs a memory request (line 1). Next, the requested size sz needs to be aligned to the multiple of 64 bytes, aimed at avoiding false sharing (line 2). Here, the cache line size is assumed to be 64 bytes. Before the alignment, the value of sz is increased by 8 (line 2). The 8 bytes are reserved to store the flag value. Usually, a flag occupying 8 bytes will not produce additional memory cost when the requested size is aligned to a multiple of 64 bytes. If the requested size sz is larger than 32 KB, the request will be performed by allocating memory from OS directly (lines 31–33). Otherwise, the allocation will be handled by shared superblocks in the heap. If sz is less than 32 KB, HMalloc should first determine the size class that the requested size belongs to (line 4). Next, a shared superblock of the size class can be obtained (line 5). If there is no superblock of this class, a new superblock will be allocated and initialized as this size class. HMalloc will first check whether the block cache list (BCL) is empty or not. If not, HMalloc will obtain an element from the list and return (lines 6–9). If the list is empty, HMalloc begins to obtain a block from current superblock. If VacantObjects is larger than 0, there is at least one unallocated memory block and the address of the block can be used as the return value, and VacantObjects should be updated (lines 10–13). If VacantObjects equals to 0, HMalloc will first determine whether current superblock is empty or not. Because there may be some memory blocks which have been freed by other threads. HMalloc will then check the Index-th block in current superblock (line 15). If the Index-th block has been freed, the block address can be used as return address and Index will be increased by 1. If Index equals to TotalObjects, it will be set to 0. Otherwise, HMalloc assumes that current superblock has been exhausted, and current superblock will be inserted into SSL list of the corresponding size class. Next, HMalloc first traverses the SSL list to obtain an empty one. If there exists an empty superblock in SSL, the superblock will replace the old current superblock and perform the allocation (lines 16–21). Otherwise, a new shared superblock should be allocated from OS, and then the request can be satisfied by the new superblock and the elements in superblock header will be updated (lines 23–25). At last, the flag value should be set to 1 when the suitable address ptr is obtained. The return address will be

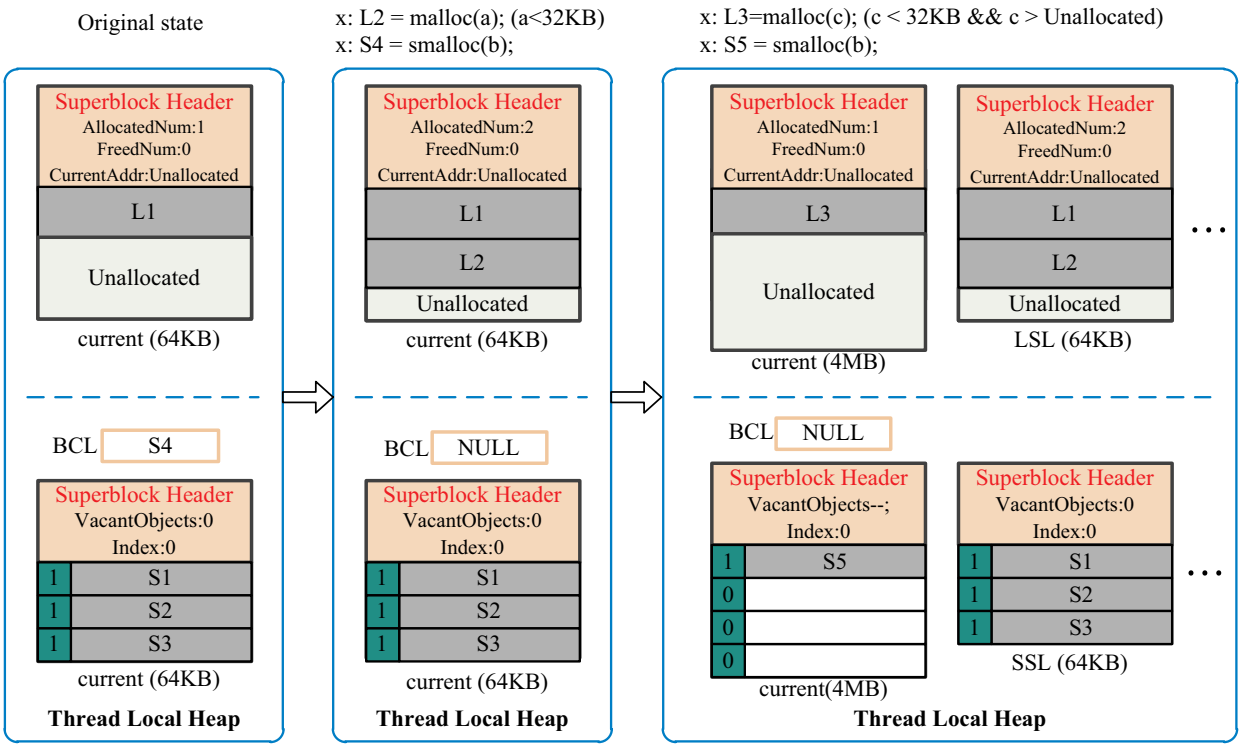


Fig. 7. A simplified process of memory allocation.

updated as ptr + 8 (lines 28–30).

Fig. 7 presents the simplified memory allocation process. We can clearly see that memory blocks in the same local heap can only be allocated by their owner thread no matter they are local or shared memory. At initial state, the current local superblock has allocated a memory block L1 and the current shared superblock of *b* size class has allocated three memory blocks. Besides, the block cache list of *b* size class also contains a block address. Next, thread x requests a local memory block of *a* size, and *a* is less than 32 KB and the size of unallocated space is larger than *a*. Hence, the current local superblock can satisfy the memory request. At the same time, thread x also requests a shared memory block of size *b*. HMalloc will first check whether the block cache list of the size class is empty or not. Since the block cache list has one element. Hence, the element can be used as the return address. Next, thread x requests a local memory block of *c* size, and *c* is larger than the unallocated space of current local superblock. Then a new local superblock of 4 MB size is allocated from OS to perform the memory request. Thread x also requests a shared memory block of *b* size again. Since there is no available chunk in current shared superblock and the block cache list is empty, a new shared superblock should be allocated from OS to satisfy the memory allocation.

In summary, there is no lock contention or atomic synchronization instructions in thread local or shared memory allocation process. All local superblocks in a local heap can only be accessed by their owner thread, and a thread can only obtain shared memory blocks from its own local heap. Lock contentions will only happen in OS kernel level when multiple threads allocate superblocks from OS simultaneously. Besides, false sharing is avoided.

3.3. Deallocation Process

The deallocation process of thread local memory is presented in Fig. 8. When a deallocation request comes, HMalloc will first determine whether the memory block is allocated from OS directly or not. If true, the block will be freed to OS directly (lines 1–2). If not, the address of the corresponding superblock can be deduced according to the address to be freed. Further, the superblock can be obtained (line 4). Next, FreedNum in superblock header needs to be increased by 1, indicating that a memory block has been freed. Then, FreedNum will be compared with AllocatedNum (line 6). If FreedNum equals to AllocatedNum, coalescence free will be executed. Because the superblock performs the same number of allocation and deallocation requests. When coalescence free is triggered, CurrrentAddr will be set to the starting address of the first memory block in the superblock (line 9). Besides, both FreedNum and AllocatedNum will be reset to 0 (lines 7–8). The next memory allocation will be allocated from StartAddr again.

The deallocation process of a shared memory block is shown in Fig. 9. HMalloc will first determine whether the memory block is allocated from OS or not. If true, the block will be freed to OS directly (lines 1–2). If not, HMalloc will check whether the block cache list is full or not (line 4). If the list is full, the allocator will calculate the address of flag value based on the address to be freed (line 5). Then, the flag value will be set to 0, indicating that the shared block has been freed. Otherwise, HMalloc will insert the address into

```
Algorithm 3: Thread local memory Deallocation Process
Data: ptr, the address of the freed memory block
Data: tlh, the thread local heap
Data: sb, the superblock contains ptr
1    if the memory block is allocated from OS
2        free it to OS directly;
3    else
4        sb ← ExtractSuperblock(ptr);
5        sb.FreedNum++;
6        if sb.FreedNum == sb.AllocatedNum
7            sb.AllocatedNum ← 0;
8            sb.FreedNum ← 0;
9            sb.CurrrentAddr ← sb.StartAddr;
10       end
11   end
```

Fig. 8. Thread local memory deallocation process.

```
Algorithm 4: Thread shared memory Deallocation Process
Data: ptr, the address of the freed memory block
1    if the memory block is allocated from OS
2        free it to OS directly;
3    else
4        if the block cache list is full
5            double * flag ← (size_t)ptr − 8;
6            * flag ← 0;
7        else
8            insert ptr into block cache list;
9        end
10   end
```

Fig. 9. Thread shared memory deallocation process.

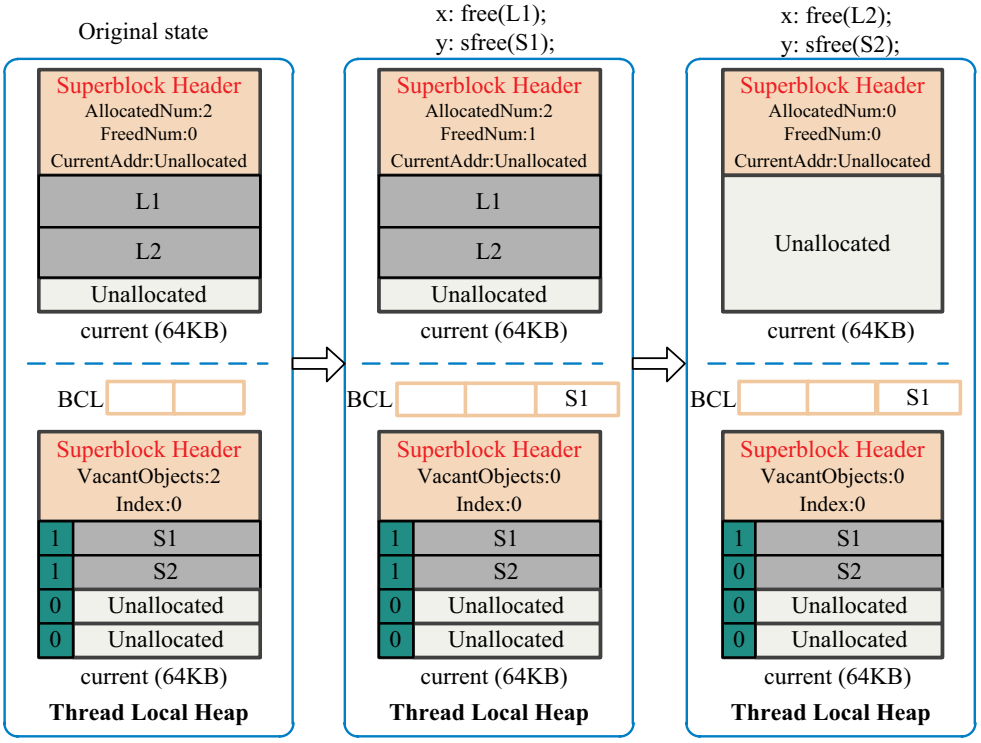


Fig. 10. A simplified process of memory deallocation.

the block cache list of the same size class (line 8). Through updating the flag value, the deallocation of share memory block become lock-free and synchronization-free. Moreover, the critical path is minimized.

Fig. 10 presents the simplified memory deallocation process. At initial state, the current local superblock has allocated memory block L1 and L2, and the current shared superblock has allocated two memory blocks, S1 and S2. Thread local memory can only be freed by its owner thread, while shared memory can be freed by other threads. Step 1, thread x frees L1 block and thread y frees S1 block. In this step, only element FreedNum in current local superblock is increased by 1. Since the block cache list is still not full, the address S1 will be inserted into the list. Then, the block cache list becomes full. Here, assuming the maximum length of block cache list is 3. Step 2, thread x frees block L2 and thread y frees block S2. In this step, we can see that block L1 and L2 are freed at the same time because of coalescence free method. Since the block cache list has been full, the flag value of S2 block should be reset to 0.

Traditional remote free methods usually depend on CAS atomic instructions to implement lock-free deallocation process. For example, SSMalloc adopts CAS instructions to implement lock-free remote deallocation, while SFMalloc designs a stack based on atomic instructions to implement remote free without lock contentions. Compared with traditional allocators, our proposed flag-based remote free method abandons atomic instructions and has minimum critical path.

4. Experimental results and analysis

There are two goals when we design the experimental scenarios, one is to prove that HMalloc performs better in traditional multi-threaded applications when compared with existing well-known memory allocators, and the other is to validate that HMalloc is more suitable for PDES applications. Here, HMalloc is compared with several well-known allocators, SFMalloc, SSMalloc and the thread-safe allocator in glibc (2.12). SSMalloc and SFMalloc are the latest multi-threaded memory allocators, and they have been proved more efficient than Streamflow [24], Hoard and TCMalloc. The performance of thread-safe allocator in glibc plays a role of baseline. Moreover, HMalloc-shared is also used as a baseline to validate the benefit of separating local memory from shared memory. HMalloc-shared is the shared memory management part of HMalloc, which perform all memory allocations in shared memory. Since the memory in Producer-consumer, Larson and the PDES benchmarks all belongs to shared memory, HMalloc-shared equals to HMalloc, so the results of HMalloc-shared are not illustrated. The experiments are mainly divided into two types:

(1) In order to validate the high performance and scalability of HMalloc, eight famous benchmarks are used to test these allocators. They are Threadtest [7], Linux-scalability [4], Largetest [25], SHbench [14], Larson [17], Producer-consumer [25], Active-False [7], and Passive-False [7]. These benchmarks can be used to verify the performance of these allocators in multiple aspects, including speed and scalability, large memory blocks, producer-consumer patterns, and false sharing. Table 1 introduces these benchmarks in detail.
(2) To verify that HMalloc is more suitable for PDES application, three famous PDES benchmarks, Phold [19], PCS [3] and Airport [3], are used to compare HMalloc with other three allocators. The time synchronization mechanism adopted in these PDES applications is conservative time synchronization [1].

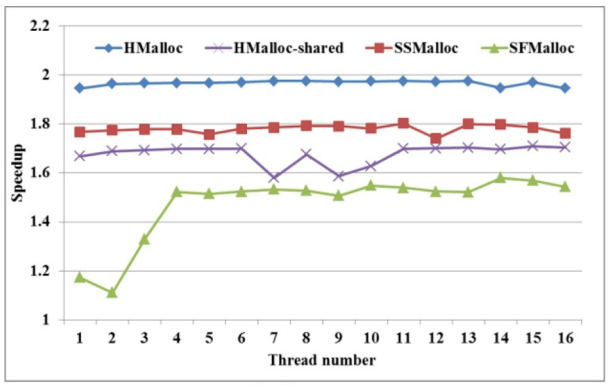
The experimental environments are as follows. The operating system is CentOS release 6.4, and the compiler version is gcc 4.4.7. The processor is Intel(R) Xeon(R) CPU E5-2650 and the size of cache line is 64B. The processor owns 16 cores and supports 16 simultaneous threads. The maximum length of block cache list is set to 100.

4.1. Speed and scalability

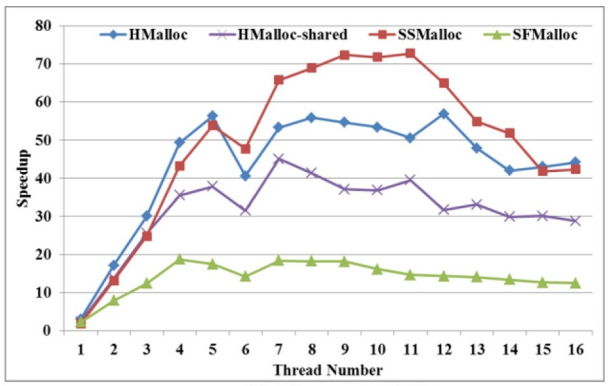
To verify the performance of allocation speed and scalability of HMalloc, three benchmarks, Threadtest, Linux-scalability and SHbench, are adopted as experimental scenarios in this section. The performance of thread-safe allocator in glibc plays a role of baseline.

Table 1
Benchmark introduction.

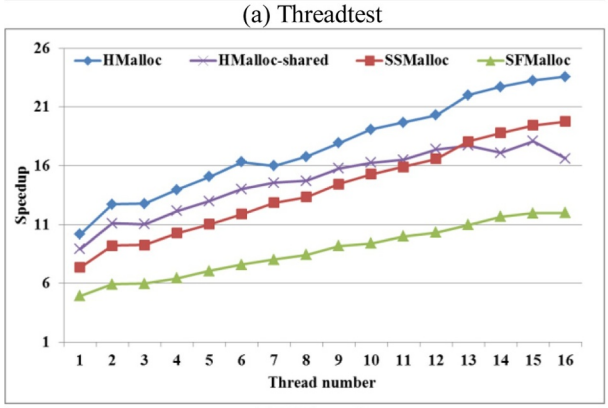
Benchmark	Description	Input	Memory type
Threadtest	Fixed-size blocks	5000 iterations, 100,000 blocks, 64B	100% local
Linux-scalability	Fixed-size blocks	512B, 1000,000 blocks/thread	100% local
Largetest	Large-size blocks	5000 iterations, 100,000 blocks, 64B, 64KB	100% local
SHbench	Random-size blocks	200,000 iterations, 1B - 1KB	100% local
Producer-consumer	Producer-consumer pattern	5000 iterations, 60,000B buffer	100% shared
Larson	Multi-threaded server	10 s, 8B - 512B, 10,000 blocks	100% shared
Active-False	Active false sharing	200,000 iterations, 8B, 20,000 writes	100% local
Passive-False	Passive false sharing	10,000,000 iterations, 8B, 8 write	50% shared
50% local			
Phold	Fixed-size blocks	Remote event percentage: 0 - 100%	100% shared
PCS	Fixed-size blocks	Events per cell: 20 - 200	100% shared
Airport	Fixed-size blocks	Planes per airport: 50 - 500	100% shared



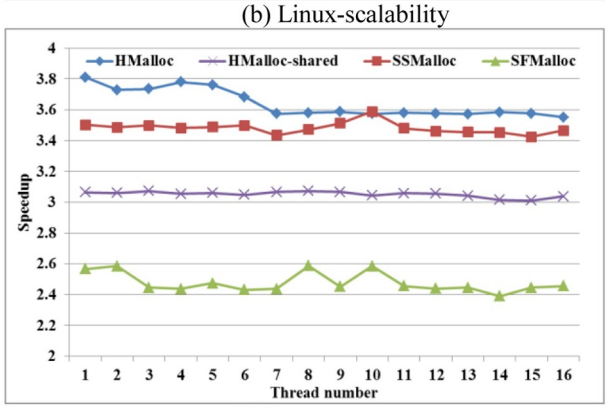
(a) Threadtest



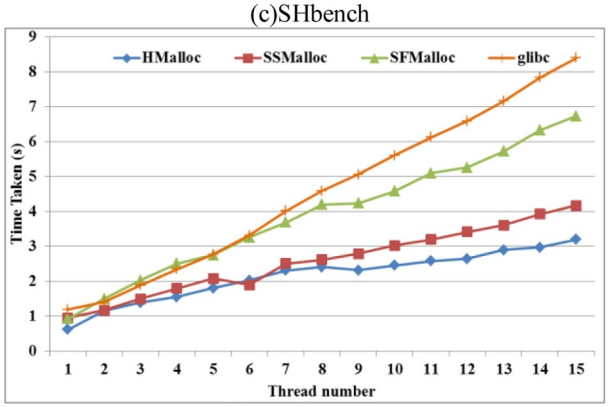
(b) Linux-scalability



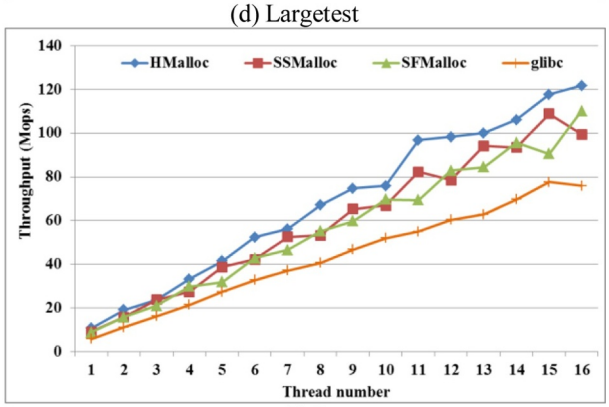
(c)SHbench



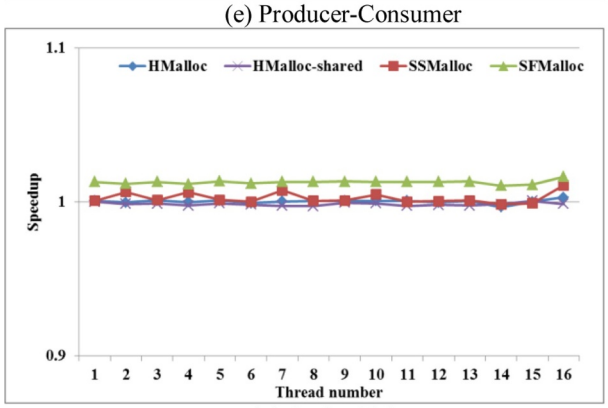
(d) Largetest



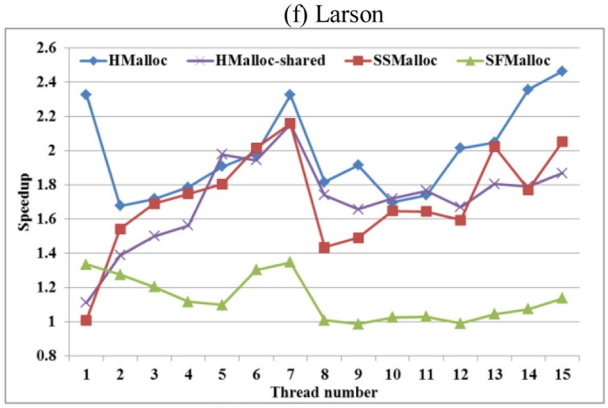
(e) Producer-Consumer



(f) Larson



(g) Active-False



(h) Passive-False

Fig. 11. Experimental Results.

In Threadtest benchmark, N threads allocate 100,000/N local memory blocks of 64B and then free all of them. This process is repeated 5000 times. Fig. 11(a) illustrates the experimental results of Threadtest benchmark. HMalloc achieves about 10% performance improvement when compared with SSMalloc, and about 21%–76% performance improvement when compared with SFMalloc. When compared with HMalloc-shared baseline, HMalloc can also achieve more than 15% performance improvement.

In Linux-scalability benchmark, each thread allocates local memory chunks of 512B for 1,000,000 times and then frees all of them. Fig. 11(b) presents the experimental results of Linux-scalability. HMalloc performs better than SSMalloc when thread number is less than 6, while performs worse than SSMalloc when thread number is larger than 6. Moreover, they both perform much better than HMalloc-shared, SFMalloc and the thread-safe allocator in glibc. The reason why HMalloc performs worse than SSMalloc is that Linux-scalability benchmark consumes large amount of memory. One thread will consume at least 500 MB memory space. Hence, HMalloc will produce a LSL list with about 125 local superblocks for each thread local heap. As described in the allocation process, HMalloc needs to traverse LSL list again to find an empty superblock when current superblock is exhausted. Traversing the list will consume a certain amount of time. Fortunately, this case rarely occurs in real multi-threaded applications.

In SHbench benchmark, each thread allocates 1000 random-size (from 1B to 1KB) blocks and frees them serially. Each thread iterates this work 2000,000/N times. Fig. 11(c) illustrates the results of SHbench. HMalloc performs much better than other allocators. HMalloc can achieve 30% performance improvement when compared with SSMalloc and 100% performance improvement when compared with SFMalloc. Moreover, HMalloc performs obviously better than HMalloc-shared.

These experimental results demonstrate that the local memory allocation and deallocation mechanism in HMalloc are more efficient than those in current multi-threaded allocators. Moreover, HMalloc performs much better than HMalloc-shared in local memory allocation, verifying the benefit of separating local memory from shared memory. The memory allocated in Threadtest, Linux-scalability and SHbench benchmarks all belongs to local memory. Hence, the local memory management part in HMalloc is more suitable for local memory allocation. Coalescence free method can efficiently reduce the overhead of local memory allocation and deallocation. If local memory chunks are not separated from shared memory, coalescence free method cannot be directly applied to memory deallocation, because multiple threads may free memory blocks in the same superblock. Besides, lock contentions are eliminated in local memory allocation and deallocation process. SSMalloc and other allocators do not make full use of the difference between local and shared memory. Take SSMalloc as an example, a thread has to free local memory blocks one by one. Besides, empty superblocks should be removed from local heap to global heap. When all the superblocks in local heap are exhausted, a thread first tries to obtain a superblock from global heap, leading to lock contentions when multiple threads access the global heap simultaneously. Through these three benchmarks, the excellent performance and scalability of HMalloc can be verified.

4.2. Large memory blocks

Largetest benchmark is used to test the performance of allocators with respect to large memory blocks. In this benchmark, each thread will allocate a large memory block of 64 KB after it allocates 100 memory blocks of 64B, and then frees all of them. This work will be repeated 1,000,000 times. All memory blocks belong to local memory blocks. As shown in Fig. 11(d), HMalloc can achieve about 10% performance improvement when compared with SSMalloc, and about 50% performance improvement when compared with SFMalloc. When compared with HMalloc-shared, HMalloc still can achieve about 20% performance improvement. The experimental results also validate that HMalloc has good stability and better performance when performs large amount of memory block requests with large size. Non-fixed-size superblocks can reduce the frequency of allocating memory from OS directly. When faced with memory allocation with large size, HMalloc will allocate superblocks of 4MB size. This will significantly reduce the frequency of allocating memory from OS.

4.3. Producer-consumer patterns

Producer-consumer pattern is often used to test whether an allocator will lead to unbounded memory blowup. Two benchmarks, Producer-consumer and Larson, are used to construct producer-consumer patterns to evaluate the robustness of these allocators. These two benchmarks also verify the performance of multi-threaded allocators when they perform frequent memory allocation and deallocation of shared memory.

In Producer-Consumer benchmark, a producer thread is responsible for allocating shared memory blocks of 8B for other consumer threads, and other threads free these memory blocks. This process is repeated 1,000,000 times. Fig. 11(e) shows the execution time of the four allocators when the number of consumer thread changes from 1 to 15. We can find that HMalloc perform better than SSMalloc in most cases, and the advantage becomes greater especially when the thread number increases. When the number of threads increases, CAS atomic instructions will be executed frequently to perform shared memory allocation and deallocation requests, which degrades the performance of SSMalloc obviously. HMalloc can achieve about 8% - 30% performance improvement when compared with SSMalloc, and about 28% - 112% performance improvement when compared with SFMalloc. The results validate that our proposed flag-based shared memory management method is high efficient and can avoid memory blowup.

Larson simulates the workload of a multi-threaded server. A thread first produces many blocks of a random size (from 8B to 512B) and inserts them into an array. Then, it frees a randomly-chosen block in the array, and allocates a new block to fill the array. The thread repeats executing this pair of free/malloc several times and then terminates. Before termination, it creates a new thread and

hands over the array to the new thread. The subsequent thread repeatedly works with the array. Larson continues for 10 s and reports the throughput of its memory operations. Fig. 11 (f) illustrates the results of Larson. Due to the block cache list, HMalloc has the largest throughput in most cases. Block cache list can cache the newly freed shared memory chunks. When a thread allocates a shared memory chunk, it will first traverse the block cache list. Hence, a thread can directly use the freed chunk when it allocates a chunk of the same size again. This can greatly reduce the overload of frequent free/malloc operations.

Through Producer-Consumer and Larson, the proposed flag-based shared memory management method has been proved efficient. HMalloc performs better than current well-known allocators when dealing with many and irregular shared memory allocation and deallocation requests. This method not only implements lock-free and synchronization-free allocation and deallocation process, but also greatly reduces the critical path. Besides, HMalloc will not lead to unbound memory blowup.

4.4. False sharing

False sharing is a notorious cause of poor performance in multi-threaded applications, it occurs when multiple processors share words in the same cache line without actually sharing data [7]. There are two types of false sharing, active false sharing and passive false sharing. The former occurs when an allocator give many threads parts of the same cache line, while the latter occurs when free allows a future malloc to produce false sharing. If a program introduces false sharing by spreading the pieces of a cache line across processors, the allocator may then passively induce false sharing after a free by letting each processor reuse pieces it freed, which can then lead to false sharing.

Two benchmarks, Active-False and Passive-False, are implemented to test the performance of HMalloc with respect to false sharing. Each thread in Active-False performs 200,000 malloc/free pairs of 8B blocks and each time it writes to every byte of the allocated block 20,000 times. In Passive-False, one thread allocates N 8B blocks and hands them to other N threads that immediately free them. Then, each of other threads allocates a 8B block and writes to every byte of the block. This work will be repeated 10,000,000 times. Since 8B is smaller than the cache line size (64B), these two benchmarks may cause severe false sharing.

Fig. 11 (g) and 11 (h) represents the results of Active-False and Passive-False, respectively. We can see that the five allocators perform nearly same when tested by Active-False. That is to say, current allocators are all able to avoid active false sharing by aligning memory requested size to multiples of the cache line size. While in the results of Passive-False benchmark, HMalloc performs better than the other three allocators, validating that HMalloc can avoid passive false sharing as much as possible, because HMalloc separates local memory from shared memory. When a thread frees a shared memory block, the block cannot be used as a local memory block by the thread or other threads. Hence, passive false sharing will not happen in HMalloc memory management. While in SSMalloc or SFMalloc, a memory block can be allocated as a local memory block or as a shared memory block. Furthermore, HMalloc still performs better than HMalloc-shared, because 50% memory allocations belong to local memory allocations and the others are shared memory allocations in Passive-False benchmarks. This can also validate that the local memory allocation part in HMalloc is more suitable for local memory allocation and the shared memory allocation part is more suitable for shared memory allocation.

These two benchmarks demonstrate that HMalloc can avoid false sharing as much as possible. This is due to that thread local memory and shared memory are managed separately.

4.5. Memory space efficiency

Table 2 reports the peak size of memory space consumed by the 8 benchmarks with 16 threads. VmPeak represents the peak size of the virtual memory allocated from the OS and VmHWM represents the peak size of physical memory consumed. The results are obtained from VmPeak (peak virtual memory size) and VmHWM (peak resident set size) in /proc/PID/status [25].

Table 2
Maximum memory footprint (MB).

	Linux-scalability		Threadtest		SHbench		LargeTest	
	VmPeak	VmHWM	VmPeak	VmHWM	VmPeak	VmHWM	VmPeak	VmHWM
HMalloc	1592.4	1460	317.9	114.1	379.3	177.5	313.7	14.1
SSMalloc	1738.9	1521.2	1292	16.1	1226.5	17.3	1292.0	7.6
SFMalloc	2335.4	2269.9	1226.3	13.9	1284.2	14.5	1218.6	5.0
Glic	3389.2	2929	1226.5	13.9	1226.5	13.0	1423.1	7.6
	Larson		Producer-consumer		Active-False		Passive-False	
	VmPeak	VmHWM	VmPeak	VmHWM	VmPeak	VmHWM	VmPeak	VmHWM
HMalloc	1080.6	602.7	188.6	15.7	252.3	44.4	237.8	44.4
SSMalloc	643.7	123.3	233.2	3.2	1292	7.4	202.4	5.2
SFMalloc	521.3	108.4	1349.7	6.8	1349.7	6.8	184.1	3.3
Glic	1569.8	96.4	233.2	3.2	1140.5	5.3	1216.2	5.4

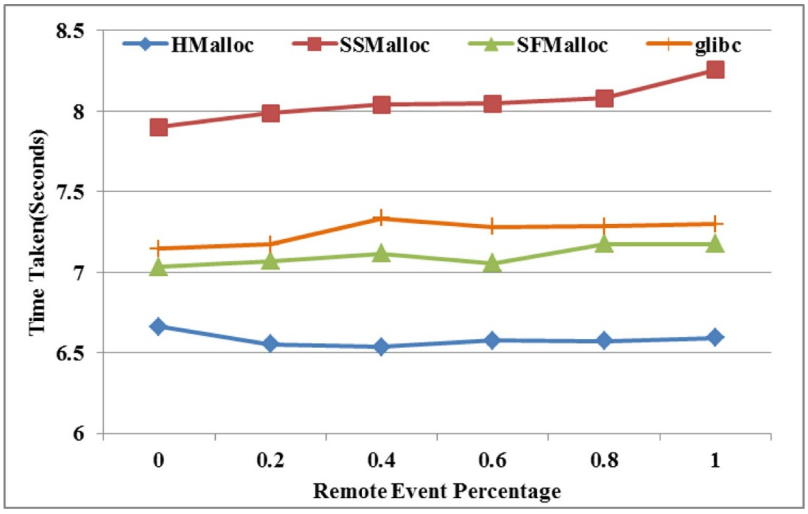


Fig. 12. Phold benchmark.

In most benchmarks, HMalloc achieves the smallest virtual memory footprint. Especially, HMalloc has the smallest virtual and physical memory footprint in Linux-scalability benchmark. On one hand, this benchmark consumes huge memory space. One the other hand, non-fixed-size superblocks will not produce too much waste of memory space. While in other benchmarks, HMalloc usually achieves larger physical memory footprint. Larson is a special case because the amount of memory usage depends on the speed of the memory allocator [25]. One reason is that HMalloc adopts coalescence free method to free local memory blocks. Coalescence free method will be triggered only when the number of freed blocks and the number of allocated blocks in a superblock are same. The other reason is that a shared superblock is assumed to be exhausted when VacantObjects equals to 0 and the Index-th block is occupied. Hence, HMalloc sometimes consumes more physical memory space to achieve more efficient memory management. Fortunately, the memory footprint sizes are acceptable.

4.6. Phold benchmark

Phold [19] is a famous PDES simulation benchmark which is often used to test the performance of PDES simulator. Here, this benchmark is also used to test the performance when HMalloc is applied to multi-threaded PDES application. Fig. 12 shows the execution time of multi-threaded Phold benchmark when adopting the four allocators under different remote event percentages. The percentage of remote event can directly reflect the frequency of communications between threads.

According to the experimental results, we can conclude that HMalloc performs the best among the four allocators, followed by SFMalloc, thread-safe allocator in glibc and SSMalloc. HMalloc can achieve about 10% performance improvement when compared with SFMalloc or thread-safe allocator, and about 20% performance improvement when compared with SSMalloc. The results demonstrate that HMalloc is more suitable for PDES application. Moreover, the performance of HMalloc is not affected by remote event percentage. Moreover, SSMalloc performs the worst, and this is contrary to the experimental results of the traditional multi-threaded benchmarks. This phenomenon denotes that multi-threaded PDES applications are different from traditional multi-threaded applications. The inter-thread communications in PDES applications are more complex and unpredictable. A thread in Phold benchmark may communicate with any other threads. While in traditional multi-threaded benchmarks, the inter-thread communications are simpler. Take Producer-Consumer benchmark as an example, a consumer thread can only communicates with the producer thread. Frequent and complex inter-thread communications will bring challenges to SSMalloc, such as frequent CAS atomic operations and passive false sharing. Therefore, a multi-threaded memory allocator is suitable for common multi-threaded applications, but may not be suitable for PDES applications.

4.7. PCS benchmark

Personal communications services (PCS) [3] network model simulates personal communications. The service area of the network is populated with a set of geographically distributed transmitters and receivers called radio ports. A set of radio channels are assigned to each radio port, and the user in the coverage area sends and receives phone calls using the radio channels. When a user moves from one cell to another during a phone call a hand-off is said to occur. In this case, the PCS network attempts to allocate a radio channel in the new cell to allow the phone call connection to continue. If all channels in the new cell are busy, then the phone call is forced to terminate. Here, a hand-off is modelled as an event.

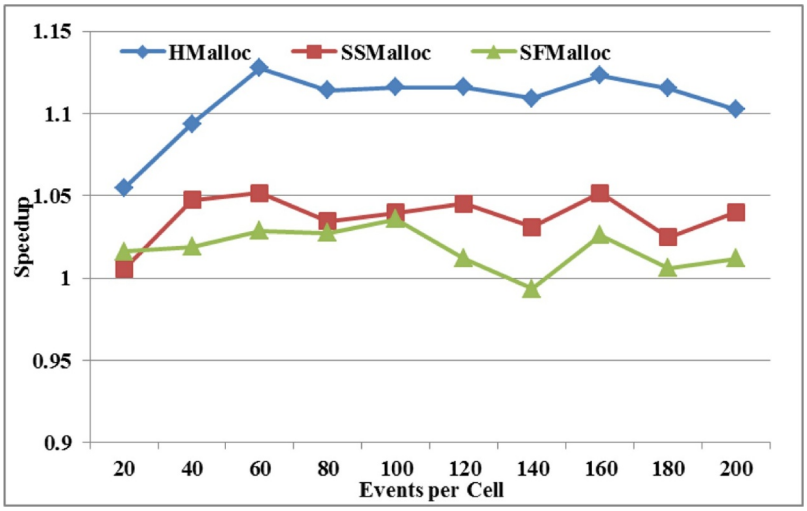


Fig. 13. PCS benchmark.

Fig. 13 illustrates the results of PCS when events in per cell changes from 20 to 200. HMalloc can achieve over 5% performance improvement when compared with SSMalloc and SFMalloc, and over 10% performance improvement when compared with thread-safe allocator in glibc. Since memory management operations do not take up the major overhead in PCS benchmark, the performance improvement is considerable. Besides, SSMalloc performs better than thread-safe allocator in glibc again. Different from Phold benchmark, the inter-thread communications between threads is much simpler.

4.8. Airport benchmark

Airport benchmark is also a typical PDES application, which is used to simulate the airport schedule. This benchmark was designed by David Bauer [3] for ROSS 4.0. There are three types of events, ARRIVAL, DEPARTURE and LAND. Each airport is modelled as a LP. When a plane plans to fly from airport A to airport B, A will first schedule a DEPARTURE event, representing that a plane is flying away. Next, the DEPARTURE event will produce a new ARRIVAL event to airport B. when B processes the ARRIVAL event, a new event LAND will be scheduled after a certain time interval. In this benchmark, the number of planes in each airport is used as the experimental variable.

Fig. 14 presents the execution time of Airport benchmark. According to the experimental results, we can find that HMalloc still outperform other three allocators. Besides, the performance of SSMalloc and SFMalloc is very close, and both of them perform better than the allocator in glibc. Compared with SFMalloc and SSMalloc, HMalloc can obtain about more than 5% performance improvement, and about 10% performance improvement when compared with thread-safe allocator in glibc. The experimental results of Airport benchmark can further validate the high efficiency of HMalloc when used for PDES applications.

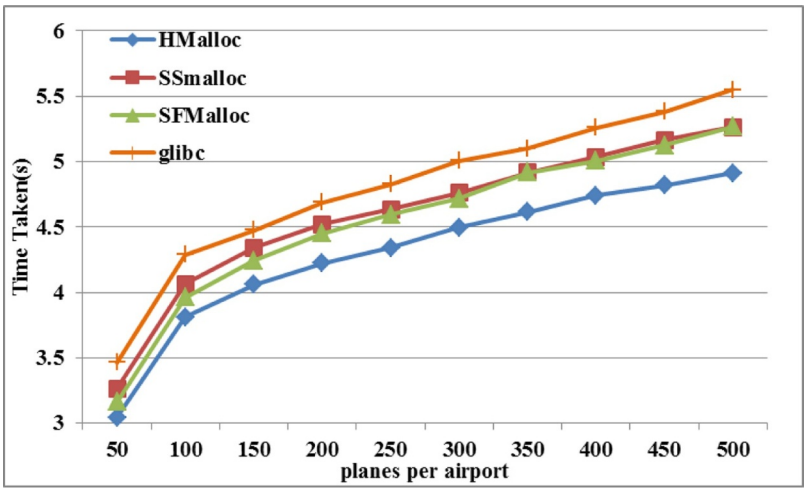


Fig. 14. Airport benchmark.

In this section, multiple multi-threaded benchmarks are implemented and executed to compare HMalloc with existing well-known multi-threaded memory allocators, aimed at validating the performance of HMalloc. Among them, Threadtest, Linux-scalability and SHbench have demonstrated that HMalloc has faster speed and better scalability. Largetest benchmark shows that HMalloc performs much better when processing large amounts of memory requests with large size. Through Producer-consumer and Larson, HMalloc is robust and can avoid unbounded memory blowup. Besides, the flag-based shard memory management method is proved to be efficient. Active-False and Passive-False shows that HMalloc can avoid false sharing as much as possible. To testify that HMalloc is more suitable for multi-threaded PDES applications, Phold, PCS and Airport benchmarks are accomplished. The experimental results fully demonstrate that HMalloc is more suitable for PDES applications when compared with other memory allocators. In conclusion, HMalloc allocator can efficiently enhance the performance of memory management in multi-threaded applications.

5. Conclusion

This paper proposes an efficient multi-threaded memory allocator named HMalloc. It innovatively separates thread shared memory from local memory. Local memory blocks and shared memory blocks are managed separately and allocated from different superblocks. Hence, false sharing and lock contentions can be avoided entirely in local memory allocation and deallocation. Besides, coalescence free method is adopted to optimize the process of local memory deallocation. Moreover, HMalloc utilizes a flag-based shared memory management method to implement lock-free and synchronization-free shared memory allocation and deallocation, without atomic instructions. Compared with traditional remote free methods, the flag-based method is simpler and minimizes the critical path. Furthermore, HMalloc adopts two sizes of superblocks, 64KB and 4MB, to accommodate various memory requests and reduce the lock contentions in OS level.

In experimental part, eight famous multi-threaded benchmarks are used to verify the performance of HMalloc in multiple aspects, scalability, speed, producer-consumer pattern and false sharing. Experimental results demonstrate that HMalloc can achieve significant performance improvement when compared with existing well-known allocators in all aspects. When tested by PDES benchmarks, HMalloc still performs better. In future work, we will further refine the superblock sizes to improve the memory space efficiency.

Acknowledgment

We appreciate the support from National Natural Science Foundation of China (Nos. 61702527, 61903368 and 61802422). We also appreciate the constructive suggestions from anonymous reviewers.

References

[1] A. Boukerche, S. Das, Dynamic load balancing strategies for conservative parallel simulation, Proceedings of the 11th Workshop on Parallel and Distributed Simulation (PADS), 1997, pp. 32–37.
[2] A. Gidenstam, M. Papatriantafilou, P. Tsigas, NBMALLOC: allocating memory in a lock-free manner, Algorithmica 58 (2010) 304–338.
[3] C. Carothers, D. Bauer, S. Pearce, ROSS: A high performance, low memory, modular time warp system, *Proceedings of the* 11th Workshop on Parallel and Distributed Simulation (PADS), 2000.
[4] C. Lever, D. Boreham, Malloc( ) performance in a multithreaded linux environment, Proceedings of the Annual Conference on USENIX Annual Technical Conference, June 2000, pp. 301–311.
[5] D. Dice, A. Garthwaite, Mostly lock-free Malloc, Proceedings of the ISMM, Berlin, Germany, 2002, pp. 163–174.
[6] D. Jagtap, N. Abu-Ghazaleh, D. Ponomarev, Optimization of parallel discrete event simulator for multi-core systems, Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium, 2012, pp. 520–531.
[7] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *Proceedings of the ASPLOS*, Cambridge, MA USA.
[8] Google, "Tcmalloc: thread-caching malloc," http://google-perftools.googlecode.com/svn/trunk/doc/tcmalloc.html.
[9] H. Park, P.A. Fishwick, A GPU-based application framework supporting fast discrete-event simulation, SIMULATION Trans. Soc. Model. Simul. 86 (10) (2010) 613–628.
[10] J. Wang, D. Jagtap, N. Abu-Ghazaleh, D. Ponomarev, Parallel discrete event simulation for multi-core systems: analysis and optimization, IEEE Trans. Parallel Distrib. Syst. 25 (6) (2014) 1574–1584.
[11] J. Wang, D. Ponomarev, N. Abu-Ghazaleh, Performance analysis of multithreaded PDES on a cluster of multicores, Proceedings of the ACM/IEEE/SCS international workshop on principles of advanced and distributed simulation (PADS), 2012.
[12] L. Chen, Y. Lu, Y. Yao, S. Peng, L. Wu, A well-balanced time warp system on multi-core environments, Proceedings of the Principles of Advanced and Distributed Simulation (PADS), 2011, pp. 1–9.
[13] M. Areias, R. Rocha, An efficient and scalable memory allocator for multithreaded tabled evaluation of logic programs, Proceedings of the IEEE 18th International Conference on Parallel and Distributed Systems, 2012, pp. 636–643.
[14] MicroQuill. Smartheap for smp benchtest. http://www.microquill.com.
[15] M.M. Michael, Scalable Lock-Free Dynamic Memory Allocation, Proceedings of the PLDI, Washington, DC, USA, 2004 June.
[16] M. Springer, H. Masuhara, DynaSOAr: a parallel memory allocator for object-oriented programming on GPUs with efficient memory access, Proceedings of the *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, 2019 June.
[17] P. Larson, M. Krishnan, Memory allocation for long-running server applications, Proceedings of the *ISMM*, Canada, Vancouver, B.C., 1998.
[18] R. Fujimoto, Parallel discrete event simulation, Commun. ACM 33 (10) (1990) 30–53.
[19] R. Fujimoto, Performance of time warp under synthetic workloads, Proceedings of the SCS Multiconference on Distributed Simulation, 22 1990, pp. 23–28.
[20] R. Leite, R. Rocha, LRMalloc: a modern and competitive lock-free dynamic memory allocator, Proceedings of the VECPAR, 2018, pp. 230–243.
[21] R. Liu, H. Chen, SSMalloc: a low-latency, locality-conscious memory allocator with stable performance scalability, Proceedings of the APSys, Seoul, S. Korea,

2012.
[22] R. Vitali, A. Pellegrini, F. Quaglia, Towards symmetric multithreaded optimistic simulation kernels, Proceedings of the Advanced and Distributed Simulation (PADS), 2012, pp. 211–220.
[23] S. Das, R. Fujimoto, K. Panesar, D. Allison, M. Hybinette, GTW: a time warp system for shared memory multiprocessors, Proceedings of the Winter Simulation Conference, 1994, pp. 1332–1339.
[24] S. Schneider, C.D. Antonopoulos, D.S. Nikolopoulos, Scalable locality-conscious multithreaded memory allocation, Proceedings of the 5th International Symposium on Memory Management, 2006, pp. 84–94.
[25] S. Seo, J. Kim, J. Lee, SFMalloc: a lock-free and mostly synchronization-free dynamic memory allocator for manycores, Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2011, pp. 253–263.
[26] W. Tang, Y. Yao, A GPU-based discrete event simulation kernel, SIMULATION Trans. Soc. Model. Simul. 11 (2013) (2013) 1335–1354.
[27] W. Tang, Y. Yao, F. Zhu, A hierarchical parallel discrete event simulation kernel for multicore platform, Cluster Comput. 16 (3) (2013) 379–387.
[28] WarpIV Technologies(J. Steinman et al). The warpiv parallel simulation kernel version 1.5.2.2008, software available fromhttp://www.warpiv.com/.
[29] X. Huang, C.I. Rodrigues, S. Jones, I. Buck, W. Hwu, XMalloc: a scalable lock-free dynamic memory allocator for many-core machines, Proceedings of the 10th IEEE International Conference on Computer and Information Technology, 2010, pp. 1134–1139.
[30] Y. Cho, D. Lee, H.K. Jun, Y.I. Eom, Lock-free memory allocator without garbage collection on multicore embedded devices, Proceedings of the IEEE International Conference on Consumer Electronics (ICCE), 2014, pp. 428–429.
[31] Y. Yao, Y. Zhang, Solution for analytic simulation based on parallel processing, J. Syst. Simul. 20 (24) (2008) 6617–6621.