Task-pushing: a Scalable Parallel GC Marking Algorithm without Synchronization Operations

Ming Wu[1] and Xiao-Feng Li[2]

[1]Institute of Computing Technology
Chinese Academy of Sciences
Haidian District, Beijing, China
wuming@ict.ac.cn

[2]Middleware Products Division
Software and Solutions Group, Intel Corp
Haidian District, Beijing, China
xiao.feng.li@intel.com

Abstract

This paper describes a scalable parallel marking technique for garbage collection that does not employ any synchronization operation. To achieve good scalability, two major design issues have to be resolved in parallel marking algorithm, i.e., the overhead of synchronization operations and load balance. This paper presents ***task-pushing****, a novel parallel marking algorithm where each thread proactively gives up its spare tasks to other threads. Enlightened by the idea of communicating sequential process (CSP),* ***task-pushing*** *arranges the computation into a process network, eliminating synchronization operations in the whole marking process. Load balance is achieved by dripping tasks from thread local mark-stack for other threads to execute. To the best of our knowledge, this is the first parallel marking algorithm that completely avoids the synchronization primitives. We evaluated* ***task-pushing*** *in aspects of queuing efficiency, load balancing strategy, synchronization overhead, and overall scalability. The results on a 16-way Intel Xeon machine showed that* ***task-pushing*** *has better scalability than* ***work-stealing*** *technique with pseudojbb and GCOld server-kind Java benchmarks.*

1. Introduction

Along with the increasing deployments of shared-memory multi-core or multi-processor computers, parallel and concurrent garbage collection (GC) is becoming more and more important in modern runtime system design. A well-developed parallel GC can effectively reduce the GC pause time, improve the runtime system scalability, and deliver better performance. Since Halstead introduced the first parallel GC [8], a variety of algorithms have been proposed to parallelize the GC process [10, 6, 7, 5, 4, 11, 14]. Today, most of the existing commercial JVMs have one or more parallel GC algorithms implemented.

Parallel GC needs to partition the collection work for the available GC threads. The collection process in a stop-the-world GC includes three main tasks:

1. Enumeration of root references,
2. Discovery of live objects, and
3. Reclamation of the garbage.

The parallelization of task 1 is well understood that each mutator can enumerate its own root set independently before the GC threads take control. In a copying GC, task 2 and task 3 are normally carried out together by scanning the objects when they are forwarded. Their parallelization is related with the GC copying order [10, 13]. In a non-copying GC like mark-sweep or compaction GC, task 2 and 3 are usually done in separate phases. Task 2 is the marking phase, and task 3 is the sweeping or compaction phase respectively. Task 3 is relatively easier to be parallelized once marking phase has identified all live objects with either an auxiliary data structure like mark-table or the object header meta-data. For example, the sweeping process can be parallelized by partitioning the heap regions among the GC threads. Since the free-list is local to the data-block it tracks, both load-balance and low synchronization overhead can be achieved straight-forwardly [4]. The compaction process can be parallelized once the target addresses of the to-be-moved objects are computed. Different from its sweeping counterpart, compaction GC usually partitions the heap regions according to the targeted compaction space. The difficult part in parallel compaction is how to preserve the object order with as few heap passes as possible [1, 14].

1-4244-0910-1/07/$20.00 ©2007 IEEE.

The marking phase is the major contributor to GC pause time in a non-copying stop-the-world GC when there are lots of live objects in large heap. Parallelization of the marking work on multiprocessors can alleviate the problem, but the parallelization efficiency is critical for the finally achieved server application performance. In sequential marking process, an auxiliary mark-stack is usually used. Every unmarked object reference that is found during object scanning is pushed onto the stack. The whole marking process can be viewed as iterations over the mark-stack elements. In each iteration, GC thread pops the top element off the stack, scans its referenced object, and pushes unmarked object references onto the stack. At first glance, this process is trivial to be parallelized because of its perfect iterative property. A straightforward parallelization is to share the mark-stack among GC threads for pushing and popping. But it requires intensive synchronized accesses to the mark-stack, which usually means high overhead and low scalability when the number of processors is big.

Some improvements were proposed to reduce the synchronization overhead [6, 7, 5, 11]. Endo et al. [6] has designed a load-balancing mechanism called *work-stealing*. When a GC thread runs out of its work, it can steal half of the tasks in another GC thread's stealable mark queue. Their work achieved impressive scalability when evaluated on servers with big number of processors. Since the stealable mark queue accesses are still critical sections, the synchronized access operations had to be carefully designed. For example, they improved the queue access algorithm from simple lock-then-steal sequence to try-lock-then-steal sequence during their development. Flood et al. [7] further improved their algorithm with a non-blocking implementation of a double-ended queue. Their implementation is known to have the best scalability for the marking phase up to now.

We believe the marking phase can be improved further when the synchronization operations can be removed completely. In this paper, we present a scalable parallel marking technique called *task-pushing* that achieves this goal. *Task-pushing* is different from all prior arts in parallel GC research in how the tasks are allocated and balanced among the GC threads. It applies the idea of communicating sequential process (CSP) [9] for thread coordination, hence achieving the merits of CSP in both synchronization and load balance. In *task-pushing*, each GC thread proactively discovers other GC threads that may want more work, and then pushes new tasks to them through a communication channel. Previously, tasks are assigned by either accessing a globally-shared task pool or stealing from other threads' task lists. These traditional ways of thread coordination cannot avoid thread synchronization primitives.

The contributions of this paper include:

1. We designed a CSP-based parallel marking algorithm for non-copying GC, which does not use any thread synchronization primitives;
2. In order to enable the CSP-based design, we developed a high-performance queue data structure that can be accessed by two threads simultaneously without atomic operation;
3. We studied the load-balancing strategies for both queuing and task selecting mechanisms;
4. We evaluated our parallel marking design on real hardware platform. Moreover, the marking phase was evaluated separately from other GC phases. This is important for us to understand the behavior characteristics of the specific phase.

We implemented *task-pushing* algorithm in Harmony, an Apache open source implementation of Java SE [2], and evaluated it on a 16-way Intel Xeon multiprocessor platform. We also implemented Flood's optimized *work-stealing* algorithm for comparison. The results demonstrate that *task-pushing* can achieve 8.3 times speedup with 16 processors with pseudojbb and GCOld. Moreover, *task-pushing* showed increasingly better performance than *work-stealing* when the processor number increases.

1.1 Organization

The rest of the paper is organized as follows. Next in section 2 we will discuss the related work. Then we introduce the algorithm of *task-pushing* and the queue data structure used for CSP process network in section 3. Section 4 is detailed discussion on the load-balancing techniques we developed. We evaluated *task-pushing* with server benchmarks in section 5. Finally in section 6 is the conclusion and future work.

2. Related Work

Parallel GC design was started from Halstead [8]. His copying GC was developed for Multilisp on shared memory multiprocessors. Each processor has its local heap organized as semi-space and moves objects from any from-space to its to-space. The algorithm uses lock bits to manipulate forwarding pointers and has no support for load-balancing, which resulted in limited scalability.

Imai and Tick [10] extended Halstead's parallel copying GC with dynamic load balancing. The idea is to put blocks with gray objects into a shared work pool, so that any thread that has finished its block scanning can grab a new block from the pool. This algorithm requires synchronized operations on both the from-space objects and work pool accesses.

Endo et al. [6] constructed a parallel stop-the-world mark-sweep GC and used *work-stealing* for load balance. The GC threads periodically check the auxiliary queues and if empty then move some tasks to them. Other starved threads can steal tasks from the queues. The accesses to the stealable mark queues are synchronized. Lockings are also needed for mark bits manipulation. They noted that substantial processor cycles are consumed by the locking operations, and improved the algorithm by using atomic CAS (compare-and-swap) instruction for mark bits manipulation.

Flood et al. [7] extended Endo's work in their mark-compaction and copying GCs. They improved the load-balancing mechanism with a cheaper work stealing mechanism, which was based on a double-ended queue proposed by Arora et al. [3].

Cheng and Blelloch [5] designed a real-time GC that supports both parallelism and concurrency. Their collector balances the work by employing a single shared stack among all threads. They used room synchronization [16] for the shared stack accesses and copy-copy synchronization for forwarding pointer accesses. Both synchronization mechanisms use atomic instructions.

Attanassio et al. [4] developed a couple of parallel GCs in different algorithms, including copying, mark-sweep, generational or non-generational. They used a shared list of work buffers for load balancing. Each processor repeatedly grabs a work buffer from the shared list. Any new references found are entered into the local work buffer. They avoided using atomic operation for object marking in mark-sweep collectors, but had to use atomic operation for the shared list accesses.

Ossia et al. [11] proposed a "server-oriented" GC that is parallel, incremental and mostly concurrent. They developed load balancing mechanism called work packet management, which is similar to the work pool of Imai's; but their GC partitions the global pool into sub-pools to reduce the atomic operations.

Abuaiadh et al. [1] extended Flood's parallel order-preserving compaction by reducing the number of heap passes from three to two, and balanced the work by splitting the heap into lots of small areas. More recently, Kermany and Petrank's Compressor [14] requires only one heap pass for parallel compaction. The parallelization of the compaction in these GCs are orthogonal to our parallelized marking phase, they can be combined to construct a highly scalable parallel mark-compaction GC.

At the same time as Compressor was presented, Siegwart and Hirzel presented a parallel hierarchical copying GC [13]. They borrowed Imai's algorithm to parallel the young generation collection in IBM's J9 JVM while achieving hierarchical copying order.

3. *Task-Pushing* Algorithm

In this section, we describe the design of the *task-pushing* algorithm. We will give a brief description about the major design points at high-level, then we discuss the queuing mechanism in details.

3.1 CSP-style task sharing

As we described in Section 1 "Introduction", the marking phase during collection is an iterative process over the mark-stack. Our parallelization algorithm firstly eliminates the shared mark-stack by maintaining separate local mark-stacks for different GC threads. Each GC thread acquires their initial tasks by evenly partitioning the root set references among the GC threads[1]. A new task is generated when an unmarked object is met during object scanning. By default, newly generated tasks are always pushed onto the mark-stack of the scanning thread. Since we implemented mark bits in object header meta-data and the mark process is idempotent, the synchronization on mark-bit manipulation is not required for correctness. In this way, we have a trivial parallel marking algorithm without any synchronization operations, although it might be badly load-balanced.

Our next step for parallelization is to allow the GC threads to share tasks, i.e., a thread can put its spare tasks into one or more queues so that other threads can grab them off the queues. Designing of the queue is critical for load-balance hence the scalability. We found the concept of CSP-based dataflow computation can be very well applied here. Each GC thread can be viewed as a sequential process, and the queues for task exchange are communication channels. Consequently, the overall marking process represents a process network. The good scalability merit of CSP computation can be achieved naturally. Based on the experiences in Shangri-La project by Chen et al. [18], we designed the process network for parallel marking as illustrated in Figure 1.

Every GC thread maintains an array of queues, one for each peer thread. The queues are the communication channels between GC threads. A queue is identified with a tuple <*i,j*>, meaning thread *i* pushes into and thread *j* pops off it. Queue <*i,j*> is an output queue of thread *i*; at the same time, it is an input queue of thread *j*. All the output queues of thread *i* are <*i*, *>, and all the input queues of thread *j* are <*, *j*>, where the asterisk symbol * refers to any legal number as a thread index.

GC thread *i* operates over its local mark-stack as usual, while occasionally pushing selected tasks into the queues <*i*, *> that have vacancies, and popping tasks off the queues <*, *i*> that have items. Figure 1 gives the pseudo code of *task-pushing* algorithm. Boolean variable *Exit_Marking* indicates whether the marking phase is ended. *mark_stack[i]* is the local mark-stack of Thread *i*.

[1] Wilson et al. [17] believed the roots should be scanned in declaration order for good locality. Our algorithm can balance the work loads well in spite of the root reference partitioning strategy.

***Thread i*:**

```
while( !Exit_Marking ) {
  while ((task = pop (mark_stack[i])) != NULL) {
    new_tasks = execute( task );
    foreach( task in new_tasks)   {
      if(j needs task-pushing and queue <i,j> has vacancy){
        enqueue( task, <i,j>);
        continue;
      }
      push(task, mark_stack[i]);
    }
  }
  if( queue <k,i> for any k has item ){
    task = dequeue( <k,i> );
    push(task, mark_stack[i] );
  }
}
```

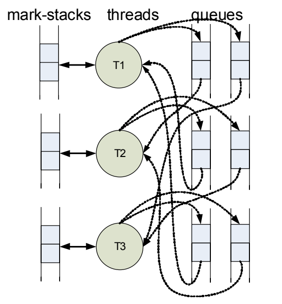

Figure 1. Pseudo code of *task-pushing*

When the queue <*i,j*> is full, Thread *i* skips the task pushing step. In this situation, Thread *j* can not be idle waiting for new tasks because its input queues are holding tasks at the moment.

Except for the queue operation, there is no other thread synchronization which might be needed to coordinate the threads activities. Once started, all threads will keep busy. We can expect good scalability from it as long as the process can be correctly terminated. Next we describe the termination mechanism.

Marking phase termination. The termination issue exists because a thread cannot locally determine if it should exit the marking phase. Empty mark-stack and empty input queues do not necessarily mean it has no more tasks, since other threads may pass new tasks to it soon.

We developed a termination detection mechanism enlightened by Peterson's mutual exclusion algorithm [19]. No CAS operation is required and the pseudo code is shown in Figure 2 with the same symbol denotations as in Figure 1.

A GC thread is arbitrarily designated to be the termination detecting thread (Thread *0* in the code). It executes different code sequence than all the rest threads (Thread *i*). A global flag *terminating* is introduced to indicate if the termination detection should be carried on. It is set TRUE by Thread *0* when it has no task, and set FALSE by other threads if any of them has tasks. Flag *no_work[i]* is maintained by Thread *i* to indicate whether it has tasks. When Thread *0* has no task, it sets *terminating*, and then checks if all other threads have no task either. If this is the case, Thread *0* will check again the *terminating* flag. If it is still TRUE, Thread *0* notifies other threads the marking phase is finished and exits. Thread *i* simply loops over if all its input queues are empty. Inside the loop, it might be notified by Thread *0* to exit the marking phase. Otherwise, if it can get new tasks from the input queues, Thread *i* simply clears the flags and returns to normal execution, whose entry is labeled as NORMAL. The proof of the correctness of the termination detection algorithm is presented in **Appendix A**.

Up to this point, we described briefly the main idea of *task-pushing*. Next we will give a detailed discussion on the queue data structure that enables the synchronization-free computation.

3.2 High Performance Queuing Design

Performance of CSP-style computation on real platform largely depends on the queue implementation. Since the commodity platforms do not have special hardware support for queuing, we developed a high performance queue data structure that does not have atomic CAS operations.

The idea of the queue is simple. We abstracted the basic queuing mechanism into single writer (enqueuing thread) and reader (dequeuing thread), i.e., SWSR (single-writer-single-reader) queue, and guaranteed the access correctness with cache coherence protocol. The queues with multiple writers or readers can be composed of SWSR sub-queues. We have used the queue design in our CSP-style network application development. In *task-pushing*, the SWSR queue is the only kind of queue which is needed.

All the entries in SWSR queue are required to be word-aligned, thus their loads and stores are guaranteed to be atomic. SWSR queue utilizes the inherent atomicity property of word-aligned memory access, which is available in all known modern processors. Since object reference is word-sized, this requirement is not a real constraint. SWSR queue uses value NULL (or any value that is invalid as a task identifier, i.e., object reference) to indicate a vacant entry. Any non-NULL entry holds a task identifier. Once an entry is dequeued, the reader stores a NULL into the entry; and before the writer enqueues, it checks whether the current entry value is NULL. The queue has a head and a tail pointer that always point to the

***Thread 0*:**

```
terminating = FALSE;
if(mark_stack[0] and <*,0> are empty)
  terminating = TRUE;
else
  goto NORMAL;

for( all thread i)
  if( ! no_work[i] || <i, *> not empty )
    goto NORMAL;

if( terminating ){
  Exit_Marking = TRUE;
  exit_mark_phase;
}

goto NORMAL;
```

***Thread i   (i≠0)*:**

```
no_work[i] = FALSE;
if(mark_stack[i] and <*,i> are empty)
  no_work[i] = TRUE;
else
  goto STOP_CHECK;

while(<*, i> are empty){
  if( Exit_Marking ){
    exit_mark_phase;
  }
}

STOP_CHECK:
no_work[i] = FALSE;
terminating = FALSE;
goto NORMAL;
```

Figure 2. Pseudo code of marking phase termination detection

```
struct queue {
      int head;
      int tail;
      void* entry[queue_size];
};
```

(a) Data structure of SWSR queue. All entry[queue_size] elements are word-aligned.

```
void* dequeue(que){
d1: head = que->head;
d2: data = que->entry[head];
d3: if( data == NULL )
d4:     return NULL;
d5: que->entry[head] = NULL;
d6: que->head = (head+1)%queue_size;
d7: return data;
}
```

(b) Dequeue implementation

```
bool enqueue(que, data){
e1: tail = que->tail;
e2: old = que->entry[tail];
e3: if( old != NULL )
e4:       return FALSE;
e5: que->entry[tail] = data;
e6: que->tail = (tail+1) % queue_size;
e7: return TRUE;
}
```

(c) Enqueue implementation

Figure 3. SWSR queue implementation

first filled and first unfilled entry respectively.

The pseudo code of the algorithm is given in Figure 3, with a label for each statement. A proof of the design's correctness is presented in **Appendix B**.

4. Load Balance in *Task-Pushing*

There are some design subtleties in *task-pushing* in order to achieve good scalability. In this section, we describe two issues with load balancing and our solutions.

4.1 Task selection for balanced sharing

As we described, any unmarked object references found during object scanning are regarded as newly generated tasks. In our first design of *task-pushing*, the new tasks of Thread *i* are fairly assigned to all threads by 1) enqueuing them into <*i*, *> that have vacant entries and, 2) pushing them onto Thread *i*'s local mark-stack. Each GC thread maintains a counter for task assignment to peer threads in round-robin fashion.

This simple load-balancing strategy can keep all threads busy; however, there is still room for improvement. For example, in some situation, a busy thread may not be able to immediately generate new tasks for other idle threads when lots of its scanned objects have no non-null reference.

We improved the algorithm by allowing task dripping from the bottom of the mark-stack. When a GC thread is scanning an object, not able to generate any new task and a peer thread has vacancies in its input queue, the thread will drip a task from the bottom of its mark-stack for the peer thread. The implementation of this double-ended mark-stack borrows some idea from Arora's *work-stealing* queue [3] but needs no atomic instruction since the mark-stack is local to the GC thread. The evaluation of task selection strategy will be presented in Section 5.

4.2 Queue length selection

The SWSR queues can also be a source of load balance. In the same situation as described above, if the queued tasks' execution can not immediately produce new tasks, load imbalance can be resulted in. The other situation is that some big tasks (tasks with some big tree hanging on them) are deposited in a queue waiting for being processed.

To reduce the potential negative impact of the queue on load-balance, an obvious choice is to reduce the length of the queue to mitigate the task deposition in the queue, so that the shared tasks can be more evenly distributed among the threads. In the extreme case where the queue has only one entry, it actually degenerates into a shared variable between two threads. However, when the length of the queues becomes too short, the weakened buffering effect may cause some loss of scalability. We expect the single entry queue brings the most scalable result when the GC threads are mostly busy-working locally and task passing happens infrequently; but longer queue would win if the threads exchange tasks frequently. Since the SWSR queue was designed for network streaming applications, it can support very high throughput of data passing. The average data throughput requirement of *task-pushing* is far lower than the capability of a long queue. We believe a very small number of queue entries are enough to sustain the best scalability of most sever applications. The evaluation presented in next section confirmed our speculation.

A short queue has an important implication to the system memory requirement. Since *task-pushing* maintains *N*(N-1)* number of queues for *N* threads, a short queue means the total memory requirement for the queues are negligible compared to the heap size. For example, with a two-entry queue, all the queues for 16 threads consume less than 2KB memory (i.e., 1920 bytes).

5. Evaluations

5.1 Benchmarks and Experimental Platform

Task-pushing is mainly designed for large-scale parallel computer, so we evaluated the technique with GCOld and pseudojbb, two benchmarks that imitate server application behavior. We ran both benchmarks with heap size of 320MB.

GCOld [15] is a synthetic benchmark which models a range of server applications in their general object characteristics. The program maintains an array of pointers to roots of binary trees. A run of GCOld consists of an initialization phase and a steady state. The initialization phase allocates and initializes the data structures, and the steady state consists of a number of steps. Each step allocates certain amount of short-lived data, does some amount of mutator work. Each step also allocates certain number of bytes in a long-lived tree structure that replaces some existing tree and makes it unreachable, then does some pointer-mutations to the long-lived trees. The parameters that control the work amount of each step can be specified in command line. In our experiments, we ran GCOld with 300MB of live data, and allocated three bytes of short-lived data for every byte of long-lived data.

Pseudojbb is the same as SPECjbb2000 [12] except that it executes a certain number of transactions to measure the execution time, rather than running for a certain period of time to measure the processed transactions. In our evaluation, we configured pseudojbb to run 8 warehouses with each warehouse processing 500000 transactions.

The platform we used for the evaluation is a machine of Unisys Enterprise Server ES7000 500 series. It has 16 Intel Xeon processors, each with 3.0 GHz frequency and 4MB cache. The operating system installed is Fedora Core release 3 from Redhat.

In order to best characterize the behavior, we measured only the parallel marking phase so as to exclude any potential confusion caused by other phases. We always use equal number of GC threads to the used processors, and one thread is affined to one processor.

5.2 SWSR queue length selection

We measured *task-pushing* scalability with different SWSR queue lengths from one entry to 64 entries. The result shows in Figure 4, which plots the speedup curves with processor number ranging from one through sixteen. On the whole, the curves show that the scalability increases with shorter queue when the length is bigger than two. Best scalability is achieved at length one for pseudojbb, while for GCOld, the best scalability with large processor number is achieved at queue length two. The result is consistent with our analysis, indicating that the buffering effect of the queue for high-throughput is not critical for *task-pushing*; but we were still surprised a little bit by the obvious gaps between the curves.

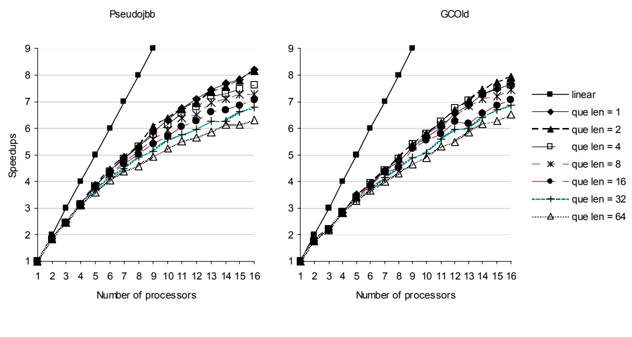

Figure 4. *Task-pushing* scalability of different queue length (*with hybrid task sharing strategy as explained in Section 5.3*)

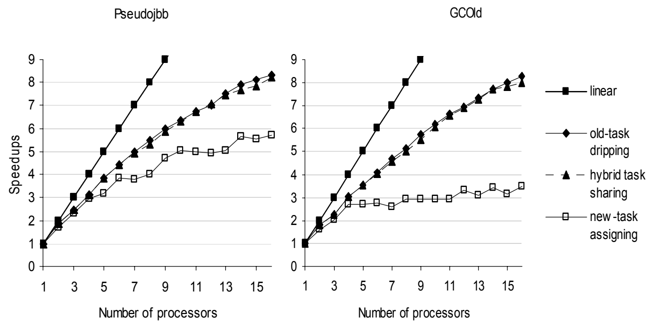

Figure 5. Scalability with different task selection strategy (with single-entry queue)

5.3 Task selection strategy

To better understand the implications of task selection in *task-pushing*, we implemented three different strategies: new task assigning, old task dripping, and the hybrid of the two.

New-task assigning: During object scanning, only the newly generated tasks are passed to other threads.

Old-task dripping: GC thread always drips tasks from the bottom of its mark-stack and passes them to its peer threads.

Hybrid task sharing: If there is no new task generated during object scanning, the GC thread drips tasks from local mark-stack; otherwise, the new tasks are passed to other threads.

The measured speedups are shown in Figure 5. The strategy “new-task assigning” got staggered speedups when processor number increases. The staggering is obviously caused by load-unbalance, i.e., an additional GC thread may not improve the overall marking performance because some threads dominate the marking time. The other two strategies which have task dripping achieved rather smooth scalability curves, whereas the “old-task dripping” is always the winner. The results are compliant with the intuition: The object reference in the bottom of the mark-stack usually represents bigger task compared to the one on the stack top, because the heap tracing algorithm implies that earlier pushed object often has a deeper tree hanging on it. This demonstrates that task dripping is essential for *task-pushing* on large scale multiprocessor platforms.

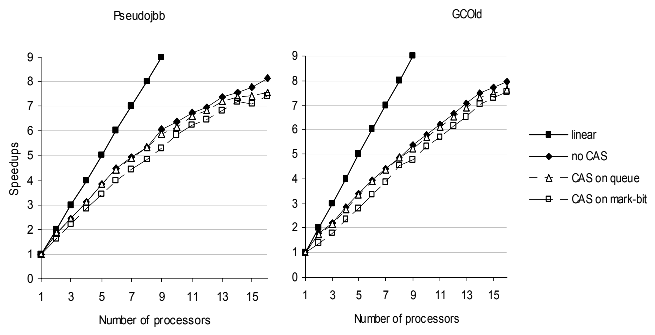

Figure 6. Synchronization impact on scalability (*with hybrid task sharing strategy and two-entry queue*)

5.4 Impact of atomic CAS operations

It is of our strong interests to understand how the elimination of synchronization operations impacts the scalability. There are two places where the atomic CAS operations were traditionally used in GC marking but are removed in *task-pushing*. One is for the mark bits manipulation in object header meta-data; the other is for the queue accesses.

Removal of the atomic CAS operations for mark-bit manipulation would potentially cause duplicate markings. Although the duplication does not lead to any incorrectness, we want to know how the benefit of CAS removal outweighs the duplicate marking overhead. In Figure 6, compared to the curve without atomic CAS operations, the scalability with atomic mark-bit manipulation is rather bad. The gap starts early from two processors and increases with more processors. The reason is that marking operation is required for every live object and small additional overhead may incur serious performance issue.

The purpose of the SWSR queues is to construct the CSP-style thread coordination and to balance workloads among GC threads. Besides the impact of queue length selection on scalability, it is also interesting to know the effect of CAS operation removal. Figure 6 shows that the scalability of a CAS-based queue is not as bad as the mark-bit CAS manipulation. With small number of processors, the gap to the "no CAS" curve is almost invisible. But the "CAS on queue" curve declines much faster than other curves when the processor number increases. At the point of 16 processors, it drops to be close to the speedup of "CAS on mark-bit". This is understandable. The total mark-bit manipulations are constant no matter how many processors are involved, while the count of queue accesses depends on the task pushing frequencies. With more processors used in parallel marking, more task pushings happen for load balancing. Besides, CAS overhead is not only the instruction cost itself, but also includes the overhead of failed trials of synchronized operations. More processors cause more contentions, which in turn lead to more failed trials.

5.5 Task Pushing vs. Work Stealing

To better understand how scalable *task-pushing* is, we compared it with *work-stealing* technique. We developed a *work-stealing* implementation of parallel marking that can replace the *task-pushing* one in the same GC module of Harmony. The algorithm is based on Flood et al. [7] for its well-known good scalability.

Figure 7 compares the execution time and the scalability of both parallel marking designs. The *task-pushing* is configured as follows: 1) SWSR queue length is one; 2) Task selection strategy is "old-task dripping"; 3) There is no atomic CAS operation on the queue access and mark-bit manipulation. The *work-stealing* implementation in the comparison has no atomic CAS on mark-bit manipulation either.

Figure 7.(a)(b) are for pseudojbb, and (c)(d) are for GCOld. Figure 7.(a) and (c) are the execution time normalized to sequential marking execution time and only show the partial curves for processor number from 5 through 16. *Task-pushing* performs a little worse when the number of processors is less than 7, and then it delivers increasingly better performance than *work-stealing* when processor number becomes bigger. The more processors are used, the better performance *task-pushing* can deliver. The initial little worse performance of *task-pushing* with small processor number is caused by additional operations for each task execution: *Task-pushing* maintains a counter to guide the target thread selection for task pushing. However, this overhead will be amortized by the benefits

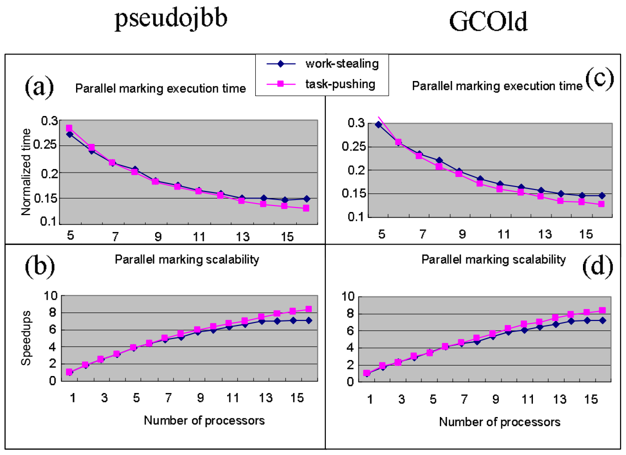

Figure 7. Performance and scalability of *task-pushing* compared with *work-stealing*

of effective scalability exhibited with a larger number of processors.

Figure 7.(b) and (d) are the speedups comparison. The data show that *task-pushing* has better scalability than *work-stealing*, and the gap is larger with bigger processor number. It is interesting to find that the speedups of *work-stealing* reach an inflection point at 15 processors, while *task-pushing* keeps growing.

One question people may have with *task-pushing* is if it is still able to keep good load balance when different threads generate new tasks at very different rates. This can happen when the object graph looks like Figure 8. If there are two threads, and the only root reference to *O1* is assigned to Thread *1*, it will push *O2* and *O3* at scanning *O1*. When it pops *O3* for scanning, *O4* and *O5* will be pushed, and *O2* is dripped and pushed to Thread *2*. If the marking goes on in this way, Thread *1* will be always busy generating new tasks, but Thread *2* will have no task generated at all. Obvious load imbalance will be incurred if the object graph is similar to the one in Figure 8, not mention the situation with more than two threads. But we believe *work-stealing* will behave no better in this situation. Because of the nature of GC marking that traces live object from root set references, the inherent data dependence in this kind of object graph dictates little marking parallelism. No matter whether it is *task-pushing* or *work-stealing*, load imbalance is ensured unless the data dependence can be broken by speculative computation. Fortunately, we have not encountered this situation in any of our experimented applications.

6. Conclusion and Future Work

Marking is one of the major components in non-copying GC design. It usually dominates the GC pause time when there are lots of live objects in a large heap. Parallel marking can effectively reduce the execution time. To achieve good scalability in large-scale multiprocessor platforms, synchronization operations and load balance must be carefully designed. We developed a parallel GC marking technique called *task-pushing* that addresses these two main issues. *Task-pushing* applied CSP-style computation to coordinate the thread activities, and a high-performance SWSR queue was designed to entirely remove the synchronization overhead. Different task sharing strategies were studied and we found task dripping can deliver best scalability.

We evaluated *task-pushing* and *work-stealing* with two server-kind benchmarks on a commodity 16-way machine. The results showed that *task-pushing*, though with a little bit more operations for single task execution, brings increasingly better performance when the processor number increases.

We are now looking at how to apply the *task-pushing* technique into sliding compaction phase, so that the whole collection process can be parallelized in CSP-style and scales well on large-scale multiprocessor environment. For next step we are also interested to eliminate the synchronization primitives in a copying collector.

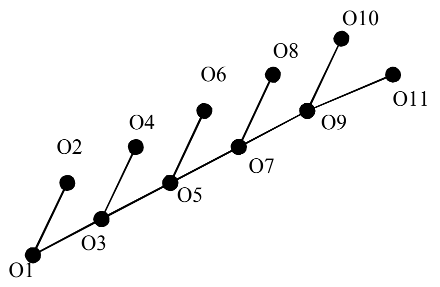

Figure 8. Object graph that may result in very different task generating rates

Acknowledgement

We thank Prof. Weiwu Hu for promoting the collaborations in this work. We thank Chen Yang and Chunrong Lai for kind helps in both experiments and discussions. We would also thank Intel for supporting this work.

References

[1] Diab Abuaiadh, Yoav Ossia, Erez Petrank, and Uri Silbershtein. An efficient parallel heap compaction algorithm. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, ACM SIGPLAN Notices, Vancouver, October 2004. ACM Press.

[2] Apache Harmony, http://incubator.apache.org/harmony/.

[3] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 119-129, 1998.

[4] Clement R. Attanasio, David F. Bacon, Anthony Cocchi, and Stephen Smith. A comparative evaluation of parallel garbage collector implementations. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, August 2001.

[5] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2001*, volume 36(5) of ACM SIGPLAN Notices, pages 125-136, Snowbird, Utah, USA, May 2001. ACM Press.

[6] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of High Performance Networking and Computing (SC'97)*, 1997.

[7] Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM'01)*, Monterey, CA, April 2001.

[8] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501-538, October 1985.

[9] C. A. R. Hoare. Communicating Sequential Processes. http://www.usingcsp.com/. June 21, 2004.

[10] Akira Imai and Evan Tick. Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 4(9), 1993.

[11] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Berlin, June 2002. ACM Press.

[12] Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, release 1.01 edition, 2001

[13] David Siegwart, Martin Hirzel. Improving Locality with Parallel Hierarchical Copying GC. In *International Symposium on Memory management (ISMM)*, 2006.

[14] Haim Kermany, Erez Petrank. The Compressor: concurrent, incremental, and parallel compaction. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI 2006)*, Pages 354-363, 2006.

[15] Sun Microsystems Laboratories. GCOld Benchmark. http://www.experimentalstuff.com/Technologies/GCold/.

[16] Guy E. Blelloch, Perry Cheng, and Phil Gibbons. Room synchronizations. In *ACM Symposium on Parallel Algorithms and Architecture*. ACM Press, July 2001.

[17] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI),* 1991.

[18] Michael K. Chen, Xiao-Feng Li, Ruiqi Lian, Jason H. Lin, Lixia Liu, Tao Liu, Roy Ju: Shangri-La: achieving high performance from compiled network applications while enabling ease of programming. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI 2005),* Pages 224-236, 2005.

[19] J. Anderson, Y.-J. Kim, and T. Herman, " Shared-memory Mutual Exclusion: Major Research Trends Since 1986 ", *Distributed Computing*, Volume 16, pages 75-110, 2003.

Appendix A: Proof of correctness of marking process termination

The tasks in the process can only stay in either the thread mark-stacks or the queues. If some threads have tasks in their mark-stacks, the marking phase cannot finish because the *no_work[i]* value for some Thread *i* is FALSE. To prove the correctness of the termination mechanism, we only need to show that no thread can exit the marking phase when no task is in mark-stacks while some tasks are in the queues. There are two scenarios.

1. The trivial case. The tasks in the queues are seen by Thread *0* during its checking on *no_work[i]* and <*i*,*> for all Thread *i*. In this case, Thread *0* simply stops the checking and goes back to normal execution. No thread can exit the marking phase.
2. The tasks in the queues are not seen by Thread *0* during its checking on *no_work[i]* and <*i*,*> for all Thread *i*. The cause for this scenario is that, after Thread *0* checks Thread *i* and finds *no_work[i]* is TRUE and <*i*,*> are empty, another Thread *j* puts some tasks into Thread *i*'s input queue <*j*, *i*>. If this happens, Thread *0* must check Thread *j*'s status after its checking on Thread *i*'s status; otherwise, it should have found Thread *j* has tasks and stop the checking at early time. Then we need consider two sub-cases:
   a) Thread *i* dequeues the tasks from <*j*, *i*> before Thread *0* checks Thread *j*'s status. Since Thread *i* only dequeues after it sets *terminating* flag to be FALSE, Thread *0* will find *no_work[j]* is TRUE and <*j*,*> are empty, but then *terminating* is FALSE. No thread can exit marking phase.
   b) Thread *i* dequeues the tasks from <*j*, *i*> after Thread *0* checks Thread *j*'s status. In this situation, Thread *0* will find <*j*, *i*> is not empty when it checks Thread *j*'s status. It will stop the checking and goes back to normal execution.

End of Proof ■

Appendix B: Proof of correctness of SWSR queue design

We describe the proof with the illustration in Figure 3. To prove the correctness of the algorithm, it is enough if we can prove the operations of dequeue and enqueue are atomic with respect to each other, i.e., sequence d1~d7 and sequence e1~e7 act like they are not interleaved in any execution scenarios. We only need to consider the shared data related load (d2 or e2), check (d3 or e3), and store (d5 or e5) operations, since the non-shared data operations are not relevant to the atomicity.

If head and tail are pointing to different entries, then reader and writer are accessing different entries, they are atomic for each other obviously. We consider only the case when head and tail are pointing to the same entry. Then there are two cases:

1. The operations of enqueue and dequeue are not interleaved. This is a trivial case.
2. The operations of enqueue and dequeue are interleaved, i.e., either d2 happens between e2 and e5, or e2 happens between d2 and d5. In either case, the two threads will load the same entry value. Since their checkings in d3 and e3 are for opposite conditions (NULL or non-NULL), only one of them can find the condition is FALSE, and continues to perform the store operation (d5 or e5). Then the two threads' operation sequences look externally just like non-interleaved, since we can always think of the thread that returns without storing finishes before the storing thread starts.

End of Proof ■