International Journal of Networked and Distributed Computing (2023) 11:88–111
https://doi.org/10.1007/s44227-023-00013-w

RESEARCH ARTICLE

Performance Evaluation on Parallel Speculation-Based Construction of a Binary Search Tree

Hiroaki Hirata[1] · **Atsushi Nunome**[1]

Received: 22 June 2023 / Accepted: 6 October 2023 / Published online: 8 November 2023

Abstract
Binary search trees (BSTs) are one of the most important data structures in the field of computer science. We may easily write a parallel construction program of a BST by extending the sequential algorithm straightly. However, in such conventional approaches, the order of nodes inserted into a BST is determined dynamically, depending on the occasional state of the parallel thread execution. It results in a BST with a different structure (node position) generated on every execution of the parallel program. On the other hand, we have been developing parallel construction schemes of the BST with the same structure as a BST generated by the sequential algorithm. One is the speculatively parallel construction of a BST. And another is the purely (non-speculatively) parallel construction, but it was derived through the concept of thread-level speculation. This paper evaluates the performances of those construction schemes on several types of shared-memory multiprocessors. For the large enough size of BST, our new parallel programs can construct a BST with always the same structure on a little lower or sometimes higher performance than the program that makes a BST with a different structure on every execution. And in contrast with the general expectation that simply enlarging the size of parallel tasks increases misspeculation and damages the performance, we found that it sometimes enhances the performance of speculatively parallel execution.

Keywords Binary search tree · Thread-level speculation · Parallel algorithm · Speculative memory

1 Introduction

Binary search trees (BSTs) are one of the most important data structures in the field of computer science. They are often used for data sorting or implementation of priority queues, which many usual applications require. They are also used to implement abstract data structures, such as sets/multisets and associative arrays.

A parallel construction algorithm of a BST can be easily derived from the sequential algorithm. However, since the structure of a generated BST depends on the order of inserted nodes, such a parallel algorithm cannot create a BST with the same structure (node position) as a BST constructed by the sequential algorithm. That is, the structure of the generated BST will be different for every time of program execution. When we use BSTs only as an intermediate data representation for data sorting and do not require sorting stability, we may not need to care about the uniqueness of the tree structure. Here, sorting stability means that the order of data elements with the same key is preserved before and after sorting.

On the other hand, for example, if we would like to perform a read or modification for data in the nodes after generating the BST, the occasional difference in the BST structure makes the verification or the debugging of the program more complex. Therefore, we have been challenged to develop parallel construction schemes to create a BST having the same structure as a BST generated by the sequential algorithm. The first scheme is based on speculatively parallel execution, and we presented it in the previous literature [1]. It is not a simple speculative execution version, and we enhanced it by embedding the checkpoint restart mechanism to reduce the repair time from the misspeculation. And the second scheme we presented in [2] is purely parallel and no longer speculative, although we have developed it through the concept of speculative execution. It is an outcome of optimizing a speculatively parallel execution model. Such conceptual flow from the sequential execution

✉ Hiroaki Hirata
hrt@kit.ac.jp

Atsushi Nunome
nunome@kit.ac.jp

[1] Faculty of Information and Human Sciences, Kyoto Institute of Technology, Matsugasaki, Sakyo-ku, Kyoto-shi, Kyoto 606-8585, Japan

through the speculative execution to the purely parallel execution is a novel methodology in parallel processing.

Our mainstream research on speculative execution is the development of a general-purpose system based on thread-level parallel speculation [1, 3–6], and the construction of a BST is one of the applications we picked up often in our past papers. Those studies provide two contributions to computer engineering. One is the parallel construction scheme as a deliverable to create the BST with the same structure as the sequential construction. And the other is the expectation that our speculation-based approach may lead to a new methodology for developing parallel algorithms for other objectives too.

In this paper, we evaluate the performance of those parallel schemes on several types of shared-memory machines, investigate their relationship with machine memory models, and verify the effectiveness of the speculation-based BST construction.

The rest of the paper is organized as follows. Section 2 summarizes the conventional purely parallel construction of a BST. The structure of the constructed BST here is not unique. Section 3 summarizes the speculative construction of a BST. The described algorithm here is not an ordinary speculative and is enhanced to improve the performance. Section 4 summarizes the non-speculative but conceptually speculation-based construction of a BST. BSTs constructed in Sects. 3 and 4 have the same structure as those constructed sequentially. The ideas described in Sects. 3 and 4 are already published [1, 2], and Sect. 5 newly evaluates and analyzes the performance of them in detail. Section 6 concludes the evaluation results and future works.

2 Background

Program 1 is a simply parallelized version of constructing the BST. The data structure of nodes of the BST is defined as NODE at lines 1 and 7–10. To resolve conflicts among threads willing to add their nodes to the same location in the BST, each pointer to a child node (the member ptr in Link) is coupled with a lock variable (the member lock in Link). Each parallel thread executes the function insert() to insert a new node—which is assigned to the argument of insert()—to a BST. It walks down in the BST to search for the insertion point while dereferencing links to child nodes and then adds a new node to the insertion point found.

Program 1 Simply-parallelized insertion codes (pseudo C language).

```
1  typedef struct node  Node;
2  typedef struct link  Link;
3  struct link {
4     Node    *ptr;    /* pointer to the child node */
5     Lock_t   lock;   /* lock for the above ptr */
6  };
7  struct node {
8     ...
9     Link  left, right;
10 };
11 Link  tree = { NULL, LockInitialValue };
12
13 void init_node_link(Node *nd)
14 {
15    nd->left.ptr = NULL;
16    nd->left.lock = LockInitialValue;
17    nd->right.ptr = NULL;
18    nd->right.lock = LockInitialValue;
19 }
20
21 #define MR(A)  (*((Node * volatile *)(A)))
22
23 void insert(Node *nd)
24 {
25    Link  *cur = &tree;  /* link to current node */
26    Node  *t;            /* current node */
27
28    while(1) {
29       if( (t = cur->ptr) == NULL )  {
30          Lock(&(cur->lock));
31          if( (t = MR(cur->ptr)) == NULL )  {
32             init_node_link(nd);
33             memory_fence();
34             cur->ptr = nd;
35             Unlock(&(cur->lock));
36             return;
37          } else {
38             Unlock(&(cur->lock));
39          }
40       }
41       if( (key of *nd) < (key of *t) )
42          cur = &(t->left);
43       else
44          cur = &(t->right);
45    }
46 }
```

At line 29 of Program 1, the member ptr of the structure Node pointed to by the variable cur points to the current node in the BST. If the value of ptr is NULL, the thread acquires the lock for it (at line 30) and re-reads it (at line 31) using the macro MR(). This macro, defined in line 21, prevents the optimizing compiler from using the already-read value and enforces re-reading from memory. And if it is still NULL after re-reading, the thread writes the address of a new node there (at line 34) after initializing the new node (at line 32). When the value of ptr is not NULL at lines 29 or 31, the thread walks down to the next child node (at lines 41–44).

Thus, the thread acquires the lock only when adding a new node and does not while merely walking down through intermediate nodes of the BST. This strategy can significantly improve the performance compared to the case that the thread acquires locks to access every node link it passes by on its walk-down. On the other hand, however, this strategy is also hazardous in parallel programming. Feng et al. [7] presented almost the same strategy as Program 1, and Howley et al. [8] proposed a lock-free algorithm for a BST. However, they said nothing about a critical situation on parallel execution.

When we attempt to optimize programs by bypassing mutual exclusion controls, we should take care of the memory ordering models of the machines on which the programs run. In the case that Program 1 runs on processors implementing x86 architecture, it works well as we expected, even if the memory fence [9] at line 33 is omitted. x86 architecture uses the total store ordering model (TSO) [10], so writes performed by one processor are observed from other processors in the original order of the writes. Therefore, when another thread can observe the results of the assignment of line 34, it can also observe the result of line 32 (that is, lines 15–18).

On the other hand, when we execute Program 1 without the memory fence at line 33 on processors which use more relaxed memory ordering models, it may abnormally terminate. For example, IBM POWER/PowerPC uses the weak consistency model [11], and MIPS/RISC-V and ARMv8 use the release consistency model [12]. In these ordering models, the writes from one processor may be observed by other processors in a different order from the original. For example, suppose a thread has modified data X and then data Y. Another parallel thread cannot always read the modified value of data X even after reading the modified value of data Y. It may read the old value of data X before modification.

Here is a concrete scenario explaining a critical situation in constructing a BST. Assume a thread P added a node X as a child of a node Y. Thread P initialized the link pointers A and B of node X to NULL (at line 32) and then set the link pointer C of node Y to point to node X (at line 34). Even when another thread, Q here, can observe that C points to node X (at line 29), Q may not yet observe that A or B is NULL (at line 29, but in the next loop iteration). Therefore, to guarantee that thread Q can always read NULL from A/B if Q has read the address of node X stored in C, a programmer must put a memory fence (or memory barrier) operation at line 33. When using the GNU C compiler or others, we can perform the memory fence by calling the built-in function __sync_synchronize().

Program 1 supports an internal BST, in which every node stores an actual key (or data). There were also lock-free or wait-free algorithms [13–15] presented for an external (or leaf-oriented) BST. In external BSTs, only the leaf nodes store actual keys, and other intermediate nodes are used merely for routing purposes. Of course, those algorithms are all for concurrent manipulations of a BST, and the structure of a constructed BST is not unique. On the other hand, this paper discusses only parallel algorithms to construct an internal BST with the same structure as a sequentially constructed BST.

3 Speculatively Parallel Construction

3.1 Speculative Memory

Thread-Level Speculation (TLS) [16–34] is one easy way to parallelize a sequential program, even if there may be unknown dependencies among parallel tasks generated from the program. Of course, the practical performance gain depends on the parallelism inherent in the program. We have been developing a Software-based Speculative Memory (SSM) system [1, 4–6] as an aid oriented for both speculative and non-speculative parallelization.

Our SSM system implicitly provides each thread with a Speculation Buffer (SB), which is the private buffer to log the history of the speculative memory accesses. The SSM system employs the lazy versioning policy. When a thread writes speculatively to memory, the SSM system adds the pair of the memory location and the write value to the SB, and memory remains unchanged. When a thread reads speculatively from memory, it gets the value from the SB if the required data are stored in the SB. Otherwise, the thread reads from memory and then adds the memory location and the copy of the read value from memory to the SB. The contents of the SB are invisible from other threads.

At the end of the task, a thread waits until all of the threads preceding in the program order of the original sequential program finish their tasks. And then, the thread tries to detect the memory hazard by comparing the read values stored in the SB and the memory values. The mismatch between them shows the hazard, which means that after the

thread read the data, some preceding threads modified it. If no hazards are detected, the thread commits. That is, the thread writes back all of the write values buffered in its SB to memory.

If a hazard is detected, the thread aborts. Concretely, it flushes its SB and re-executes its task from the beginning. However, the SSM system supports the so-called checkpoint-repair mechanism. Therefore, the SSM system does not constantly enforce the thread to re-execute from the beginning of the task and can make it roll back to the point of reading the data that caused the hazard.

3.2 Speculative Node-Insertion Algorithm

Program 2 is a part of the speculatively parallel version using the SSM library. This paper roughly summarizes the program's behavior. More details, such as implementations of the SSM library functions and speculative execution mechanism, were described in [1]. Each thread executes the function insert() to add a new node to the BST.

Program 2 Speculative insertion codes (pseudo C language).

```
1  typedef struct node  Node;
2  typedef struct link  Link;
3  struct link {
4     Node    *ptr;   /* pointer to the child node */
5     Node    *prov;  /* provisional pointer */
6  };
7  struct node {
8     long  sn;   /* serial number */
9     ...
10    Link  left, right;
11 };
12 Link  tree = { NULL, NULL };
13
14 void init_node_link(Node *nd)
15 {
16    nd->left.ptr = NULL;
17    nd->left.prov = NULL;
18    nd->right.ptr = NULL;
19    nd->right.prov = NULL;
20 }
21
22 void insert(Node *nd)
23 {
24    Link  *cur = &tree;  /* link to current node */
25    Node  *t;            /* current node */
26
27    while(1) {
28       if( (t= cur->ptr) == NULL ) {
29          if( ((t = cur->prov) == NULL) || (nd->sn < t->sn) ) {
30             sm_readed_chkpt(&(cur->ptr),NULL);
31             init_node_link(nd);
32             memory_fence();
33             sm_write(&(cur->ptr),nd);
34             cur->prov = nd;
35             return;
36          }
37       }
38       sm_readed_chkpt(&(cur->ptr),t);
39       if( (key of *nd) < (key of *t) )
40          cur = &(t->left);
41       else
42          cur = &(t->right);
43    }
44 }
```

A tree node contains a serial number (at line 8), representing the logical order if the BST is sequentially constructed. Before a thread calls the function insert(), the serial number of the node assigned for its argument must have already been set. A node-link has two pointers to nodes (at lines 4–5). One is the "fixed" pointer ptr, pointing to the correct child node. Another is a provisional pointer prov, pointing to the node that may be a child. A pointer written speculatively to ptr by a thread is merely buffered to its SB and remains invisible from other threads until the thread commits. Therefore, a succeeding thread may attempt to add its node here without knowing that the preceding thread has already modified ptr. This failure to find the correct insertion point leads to misspeculation. Therefore, we make a thread write a pointer into prov before it commits, which helps the succeeding threads walk down to deeper levels in the BST.

A thread (P here) checks whether the current pointer is NULL (at line 28). Even if it is NULL, another thread (Q here) that has not yet committed might have modified it speculatively. So next, P checks the provisional pointer (at line 29). If it is also NULL, P speculatively adds its node here (at lines 30–34). Otherwise, P checks whether Q is P's preceding or succeeding thread by comparing their node's serial number (at line 29). If Q is P's preceding thread, P behaves as if Q's node has already been added here (and goes to line 38). However, if Q is P's succeeding thread, P speculatively adds its node here (at lines 30–34).

Function sm_readed_chkpt() is an SSM library function that adds a speculative read history to the SB and puts a checkpoint. When called at line 30, it records that the value of cur | ->|ptr was NULL. When called at line 38, it records that the value of cur | ->|ptr was the value stored in variable t. Here, we do not care what the actual content of cur | ->|ptr is. We merely pretend that the content of cur | ->|ptr was so. Before the thread tries to commit, it checks the content at the memory location specified by cur | ->|ptr. If the actual content on memory is not the same as the value recorded by function sm_readed_chkpt(), the thread re-reads from that memory location and re-executes the posterior part of the task.

A thread adds its node to a BST by calling an SSM library function sm_write(). At line 33, it speculatively writes the address of its node to cur | ->|ptr. The result of this speculative write is invisible from other threads until the thread commits. Therefore, to make its node addition visible immediately from other threads, the thread also writes directly to cur | ->|prov (at line 34). When we execute this program on the machine using relaxed memory ordering models, the memory fence is necessary between the call of init_node_link() (at line 31) and the write to cur | ->|prov (at line 34).

After a thread finishes the execution of insert(), it tries to commit. It scans its SB to verify the read histories, and if no hazard is detected, it commits. For Program 2, only the read histories recorded by function sm_readed_chkpt() should be verified. If the thread detects a hazard, it calls a repair function. It must find the correct insertion point by walking down in the BST from not the root node but the faulted node link. Of course, besides insert(), a programmer must provide a repair function that starts the walk-down from a given (faulted) node. As described above, the SSM system passes the memory location to be re-read as the faulted node-link information to that repair function.

4 TLS-Inspired Parallel Construction

4.1 Development Insight for a New Parallel Algorithm

TLS requires commitment, which causes the serialization among parallel threads. This serialization essentially comes with two overheads. First, every thread trying to commit must wait for all its preceding threads to commit. Second, the hazard detection, the re-execution on abort (or the repair from misspeculation), and the write-back on commitment should be performed exclusively from other threads. Therefore, the time it took for those cannot be hidden.

Our further step in TLS system development was to reduce (or remove, if possible) the overhead for the commitment in TLS. To enable each thread to add a node without waiting for its preceding threads, first, we must employ the eager versioning policy instead of the lazy versioning one, which SSM uses. However, in this case, the following two issues arise. One is managing the multiple versions of data as visible from all threads. In general, this creates a fatal overhead. However, when inserting a new node into a BST, a link pointer is modified only once from NULL to the new node's address and not modified multiple times by multiple threads. We can make use of this characteristic to reduce the version management overhead.

The other and more significant issue is who and how to resolve a detected hazard. With lazy versioning, it is natural for each thread to be responsible for detecting and resolving a hazard on the nodes it inserted by itself. With eager versioning, however, it may be natural for a thread to detect another thread's misspeculation result. Consequently, as presented in [2], we finally gave a thread that found a hazard during its walk-down (not the thread itself that caused a hazard) the role of repairing the misspeculated node insertion.

4.2 TLS-Inspired Parallel Node-Insertion Algorithm

Program 3 implements our new parallel algorithm. Each thread rearranges nodes when it detects another thread's misspeculation. While the program involves such analogous features with TLS, such as logging of speculative reads, hazard detection, and checkpoint repair, it is written as a purely parallel program directly using the standard lock primitives. Each node has a serial number (at line 8), and the locking granularity is the link pointer (lines 3–6 and 10). Each thread executes function insert() to insert a new node to a BST or reinsert a node once inserted but removed because of a hazard detected. When function insert() is called, structure member left.ptr/right.ptr of the node assigned for the argument of function insert() must be NULL or point to another node.

Program 3 TLS-inspired parallel insertion codes (pseudo C language).

```
1  typedef struct node  Node;
2  typedef struct link  Link;
3  struct link {
4     Node    *ptr;    /* pointer to the child node */
5     Lock_t   lock;   /* lock for the above ptr */
6  };
7  struct node {
8     long  sn;    /* serial number */
9     ...
10    Link  left, right;
11 };
12 Link  tree = { NULL, LockInitVal };
13
14 void insert(Node *nd)
15 {
16    Link  *cur = &tree;  /* link to current node */
17    Node  *t;            /* current node */
18    Link  *retry;
19    Node  *replaced;
20
21    reset_history();
22    while(1) {
23       if( ((t = cur->ptr) == NULL) || (nd->sn < t->sn) ) {
24          Lock(&(cur->lock));
25          if( (t = MR(cur->ptr)) == NULL )  {
26             if( (retry = verify()) != NULL ) {
27                Unlock(&(cur->lock));
28                cur = retry;
29                continue;
30             } else {
31                reset_node_link(&(nd->left));
32                reset_node_link(&(nd->right));
33                cur->ptr = nd;
34                Unlock(&(cur->lock));
35                return;
36             }
37          } else if( nd->sn < t->sn )  {
38             if( (retry = verify()) != NULL ) {
39                Unlock(&(cur->lock));
40                cur = retry;
41                continue;
42             } else {
43                replaced = substitute(cur,t,nd);
44                Unlock(&(cur->lock));
45                if( replaced != NULL )
46                   push_removed(replaced);
47                return;
48             }
49          } else Unlock(&(cur->lock));
50       }
51       add_history(cur,t);
52       cur = ( (key of *nd) < (key of *t) ) ? &(t->left) : &(t->right);
53    }
54 }
```

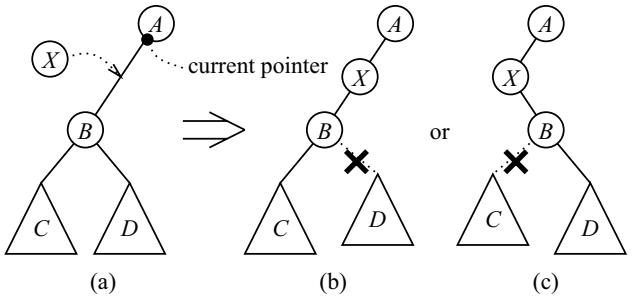


Fig. 1 Example of restructuring the BST on the node insertion

The outline of the node insertion is as the followings. A thread walks down in the BST to find the insertion point. When it encounters the NULL pointer or a pointer to the node that any succeeding thread has already added, it recognizes here is the insertion point. If another node has been added by a succeeding thread, the thread removes it and its subtrees from the BST and adds a new node here. Note that other threads may be walking down in the subtree being removed. Therefore, at this time, we had better remove nodes with a unit of a subtree and never decompose the subtree to nodes. On the other hand, when a thread tries to add a new node at an insertion point, it should verify that its insertion point is not included in a removed subtree. For this verification, a thread walks down with logging the nodes it passed by. This log is also helpful for restarting the walk-down not from the root node of the BST but from the cut-off point after the subtree is removed.

Now, we trace the behavior of a thread using Program 3. A thread starts the walk-down of the BST after clearing the logs of nodes on the walk-down path by calling the function reset_history() at line 21. This log is the private data structure for each thread and is implemented with a linear list containing the pairs of the pointer to the link field (left or right) in a node and its content (left.ptr or right.ptr). The function add_history() at line 51 adds a new pair of cur and t to the log.

If the current pointer (cur | ->|ptr) is NULL or points to a node with a later serial number (at line 23), this place may be the insertion point. Therefore, the thread acquires the lock for this link (at line 24) and then re-reads the current pointer from memory (at line 25) to confirm this place is undoubtedly the insertion point. When the current pointer (cur | ->|ptr) is still NULL, and this insertion point is not included in a removed subtree, the thread executes the codes in lines 31–35. The function reset_node_link() removes the node pointed to by the link assigned as its argument and inserts it into the list of removed subtrees (LRST). Here, note that the new node to be inserted may be a node removed from the BST before. And then, the function reset_node_link() initializes the link to NULL. After that, the thread adds the new node to this insertion point (at line 33).

When another node has already been added to the insertion point, and this insertion point is not included in a removed subtree, the thread executes the codes in lines 43–47. The function substitute() removes the already added node and adds a new node. Before, we said we remove nodes with the unit of a subtree, but it is better to retain the result of the already performed insertion as possible. Figure 1 shows an example of restructuring the BST. Figure 1a illustrates the BST before a thread adds a new node X as the left child of node A. A succeeding thread already added node B here, and node B has its child subtrees C and D. The function substitute() inserts node X between nodes A and B instead of replacing a subtree whose root is node B with node X. If node B's key is less than node X's, substitute() removes subtree D and restructures the BST to one shown in Fig. 1b. Otherwise, it removes subtree C and restructures the BST to one shown in Fig. 1c. The function substitute() returns the pointer to the root node of the removed subtree (C or D in the case of Fig. 1) (at line 43). Therefore, the thread inserts the removed subtree into the LRST by calling push_removed() (at line 46). Multiple locks are required for the manipulation in substitute(), and we must consider the case that other threads pass through the restructuring region in the BST. Therefore, the manipulation in substitute() must be implemented carefully. Details of substitute() [and other functions called from the inside of insert()] and the critical cases to be considered to guarantee the correctness of the program are described in [2].

When the insertion point is included in a removed subtree, the thread executes the codes in lines 27–29 or 39–41. The thread executes the function verify() (at lines 26 or 38) to check whether the found insertion point is included in a removed subtree. The function verify() scans the walk-down log and confirms the path from the root node to the insertion point is still connected. If it detects the path is cut off, it discards information from the cut-off point to the insertion point in the log and returns the pointer to the terminal node of the cut-off path. Thus, the thread can restart the walk-down from the cut-off point, reducing the repairing cost compared to restarting from the root node of the BST. Between the time when a thread has finished the execution of the function verify() and the time when it adds a new node to the insertion point, another thread may cut off the walk-down path. Therefore, even if verify() did not detect that the walk-down path is cut off, it is not always guaranteed that the found insertion point is correct. This rare case is not preferable in performance. Still, it does not pollute the correctness of the BST construction, because nodes inserted in a removed subtree are reinserted into the BST later.

5 Performance Evaluation

5.1 Evaluation Setting

5.1.1 Evaluated Programs

We measured the time each program described above took to construct a BST. As the search key, we used a randomly generated 29-character string. We started the measurement from the state where input data were already loaded to the main memory. The resulting time excludes the time spent on actual I/O, memory allocation of input data, key generation, and so on.

In our discussion below about the performance, we use labels consisting of two characters; the first character represents the lock implementation, and the second character represents the program (algorithm). For the lock implementation, we used either of the following two mechanisms for all inter-thread synchronization in a program;

- M(Mutex): the suspend lock mechanism, which puts a thread that could not acquire a lock to the sleeping state at the OS level, and
- S(Spin): the spin-lock mechanism, which lets a thread trying to acquire a lock check repeatedly whether it is available.

The second character of the labels represents one of the following three types of programs:

- P (Par: parallel) represents the purely parallel program shown as Program 1. It acquires a lock only for a leaf node when adding a new node. The structure of the generated BST is usually different every time of program execution.
- S (Spec: speculative) represents our speculatively parallel program shown as Program 2. It is implemented using our SSM library based on the lazy versioning policy. It creates a BST having the same structure as a BST generated by the sequential algorithm.
- R (Rear: rearranging) represents our purely parallel program developed with inspiration from TLS. It is shown as Program 3 and rearranges nodes in the BST on detecting a node inserted differently from the sequential insertion order.

Par is not a primary target for our discussion in this paper, but we evaluated it for comparison.

5.1.2 Problem Size and Task Granularity

For the data size of a BST, we evaluated the three cases: 10,000, 1,000,000, and 100,000,000 nodes.

About the task size, we took the two cases of $U = 8$ and $U = 64$, where U is the number of nodes included in a task. By increasing U, we can reduce the overhead of the task allocation. On the other hand, a large U may unbalance loads of threads (in Par and Rear) and increase the time spent for hazard detection, commitment, and repair from misspeculation (in Spec).

Each data node is initially included as an element of a linearly linked list, named the source node list, shared among parallel threads. A thread in parallel versions of the program (Par and Rear) acquires the lock and then takes out U nodes at once from the list. On the other hand, since threads in the speculative version (Spec) are logically ordered, each thread traverses the linked list independently from other threads and gets U nodes from the list while skipping the nodes allocated to other threads. So the pressure on the total memory access bandwidth is higher than that in the parallel versions, but instead, the threads don't suffer from waiting to acquire the lock to get their task.

The task allocation unit is U nodes, but in the case of Spec, the repair from misspeculation is performed for each node insertion, not each task. Assume the case of $U = 3$ for simplicity, and a thread inserts nodes A, B, and C in this order. After inserting them speculatively, the thread checks for hazards. When a hazard is detected only on the insertion of node B, the thread commits the result of the speculative insertion of node A, repairs from misspeculation on node B, and commits on node C. For node B, the thread restarts the walk-down not from the root node of the BST but from the faulted point. Note that misspeculation on node B does not involve reinserting node C.

5.1.3 List of Removed Subtrees

In Rear, each thread acquires the lock to get a task of U new nodes, as described before. Besides that, nodes included in the LRST must be reinserted. Therefore, a thread checks whether any subtree remains in the LRST every time after inserting a node into the BST. If any subtree does, the thread inserts them until the LRST becomes empty, then inserts the next node of a task. Here, an implementation issue of the LRST arises. In this paper, as the principle, all threads share one global LRST, which is implemented as a stack with a linear linked list structure. Therefore, mutual exclusion control is needed when a subtree is pushed to or popped from the LRST. Therefore, we enable each thread to keep up to L removed subtrees locally. A thread can add a removed subtree to the global LRST only if it keeps L subtrees locally. And a thread takes out a subtree from the global LRST only if it keeps no subtrees locally. Thus, we can alleviate the frequency of access racing to the global LRST.

We ran the program of Rear with several values of L and examined the optimal value of L. As a result, when $L = 2$, we could, in many cases, but not always, get the shortest execution time of the program. Therefore, we use $L = 2$ in this paper.

5.1.4 Execution Environment

We ran the BST construction programs on the following four types of server machines:

- Machine A has one Intel Xeon Gold 6126, which consists of 12 processor cores with 2-way SMT [34, 35]. The total number of logical processor cores is 24.
- Machine B has two Intel Xeon Silver 4216s, each consisting of 16 processor cores with 2-way SMT. The total number of logical processor cores is 64. The connection between two MPU chips and multiple memory banks causes non-uniform memory access time. Therefore, execution time may vary depending on which thread ran on which processor core.
- Machine C is an IBM Power S812L which has one POWER8 MPU. This MPU comprises two chip dies, each including five physical processor cores with 8-way SMT. The total number of logical processor cores is 80. POWER8 uses the weak consistency model, so explicit memory synchronization, such as memory fence operation, may be required when we write a non-conservative program on the usage of lock mechanisms.
- Machine D is a Mac Studio, which has one Apple M1 Ultra. This MPU comprises two chip dies directly connected without the usual bonding wires. The processor core implements ARMv8 architecture and does not support SMT. The total number of cores is 20, and 16 of them are high-performance cores, while the other 4 are high-efficiency cores. We enforced the threads of the BST construction programs to run only on high-performance cores. ARMv8 architecture employs release consistency as the memory ordering model, so explicit memory synchronization should sometimes be considered in multithread programming, similar to Machine C.

These machines' main memory had enough capacity to store all the data in the source node list and the BST.

All machines were managed by the Linux operating system, which dynamically controls the allocation of parallel threads to logical cores. In Machine D, we left the thread allocation among high-performance cores to the OS. We ran the BST construction program without any other user programs running simultaneously. Nevertheless, since we cannot eliminate the influence of daemon processes and the operating system, we measured the execution time more than thirty times. In this paper, we considered the actual execution time to be the shortest time measured.

We measured not only the execution time but also information about the misspeculation. We inserted codes to count the number of occurred misspeculations to programs for measuring the execution time. For Spec, we counted the number of nodes to be reinserted after the abort—although the repair function of SSM can reduce the misspeculation penalty. For Rear, we counted the dynamic number of reinserted nodes after being removed once from the BST. One (same) node might be reinserted multiple times. This count does not include the number of retried insertions immediately after the cut-off of the walk-down path is detected. We measured the number of reinsertions more than 30 times and considered the average number of the measured as the actual reinsertion number. The difference in the number of reinsertions in the same program may give us helpful information. However, note that we cannot directly compare them between different programs, because the reinsertion cost is not the same.

We used GCC (version 8.3.1 for Xeon, 4.8.3 for POWER8, and 12.1.0 for ARM) with the -O2 option to compile the BST construction programs and our SSM library. We used the Pthread library for multithreading.

5.2 Results

5.2.1 Basic Performance Modeling

Before discussing the experimental results, we here roughly model the potential for fundamental performance improvement. Let T_w be the time it takes a thread to execute a task, and T_e be the time it takes to complete a work that should be performed exclusively after or before executing a task. In the cases of Par and Rear, such exclusive work includes the take-out of a task from the source node list. Note that T_e does not include the waiting time to acquire a lock during the race with other threads. In the case of Spec, exclusive work includes detecting hazards (i.e., misspeculation) and committing speculative execution results while repairing misspeculation. T_e does not include the time to wait for preceding threads to finish their tasks.

While one thread performs the exclusive work above, other threads can execute their tasks. However, two or more threads must not perform the exclusive work simultaneously. Therefore, the total summation of T_es is the lower bound of the construction time of a BST. For simplicity, assume that T_w and T_e are constant, and T_w is a multiple of T_e. Under such

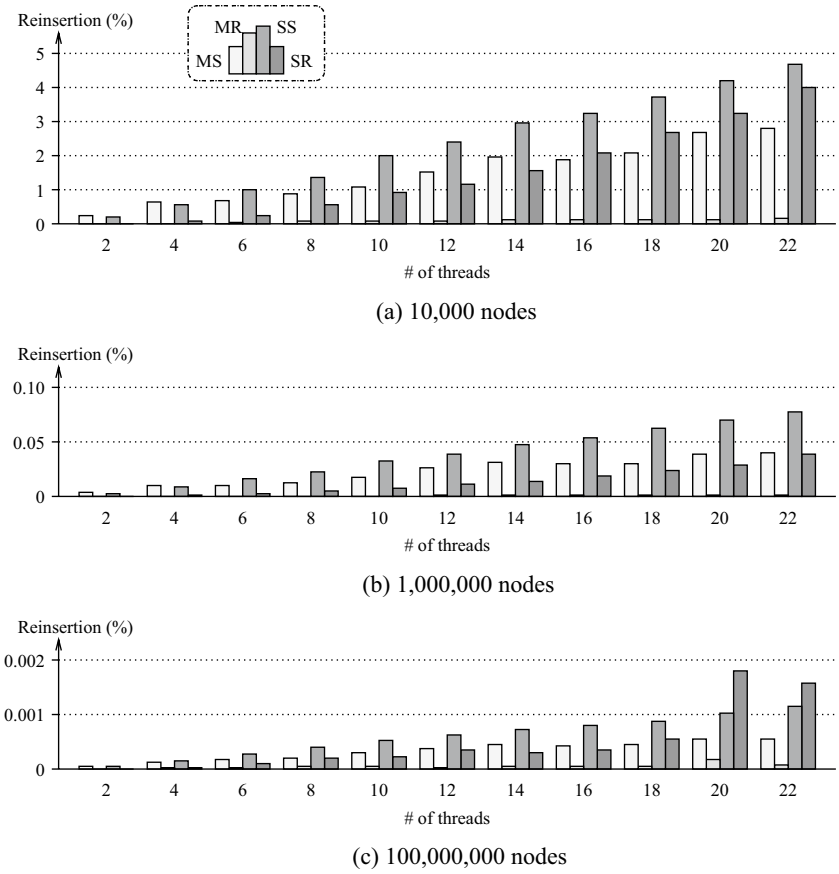


Fig. 2 Reinsertion ratios in the case of one 24-thread Xeon, 8-node task

an assumption, at most T_w/T_e threads can run practically in parallel, whereas other threads are waiting in the race to execute the exclusive work. Therefore, increasing the number N of parallel threads will improve performance when $N < T_w/T_e$. However, when $N > T_w/T_e$, we can not expect more performance improvement.

5.2.2 Results of One 24-Thread Xeon and 8-Node Task

Figure 2 shows the node reinsertion ratios to the total number of nodes when we ran them on machine A. The task size was 8.

It is natural that, as increasing the number of threads, misspeculation tends to occur more frequently. Figure 2 supports it. The number of reinsertions in SS and SR is more extensive than in MS and MR, respectively. The reason is threads using the spin lock mechanism are more active than those using the suspend lock, and the speculation tends to be deeper. In the case of using the suspend lock mechanism, the long suspending time limits the activity of threads, and the degree of speculation is reduced. As a result, it leads to less misspeculation. In Fig. 2, the reinsertion in the case of 10,000 nodes looks remarkable compared with the case of 1,000,000 or 100,000,000 nodes, but it is less than 5%.

Figure 3 shows the relative speedup ratios of parallel versions to the sequential version of the BST construction program.

In the case of 10,000 and 1,000,000 nodes shown in Fig. 3a, b, the performance of SP and SR degrades if the number of threads increases beyond some threshold. Such degradation suggests the number of parallel threads doing practical work decreased. The first reason can be explained based on the performance model described in Sect. 5.2.1. When the size of the BST is small, the depth of the BST is also not so deep, and so T_w is short. Since the number of practical parallel threads is limited by T_w/T_e, the performance is saturated even with a comparatively small number of threads. What is worse, on the small size of the BST, multiple threads are prone to race frequently with one another to add their nodes to the same node. Another reason is load unbalancing. During some periods before the end of program execution, some threads finish their tasks, and the number of left threads still working decreases. This time appears relatively long, because the total execution time is short.

Figure 2a shows that the node reinsertion occurs relatively many times in SS and SR, but we cannot read it affects dominantly the performance from Fig. 3a. In Fig. 3a, the

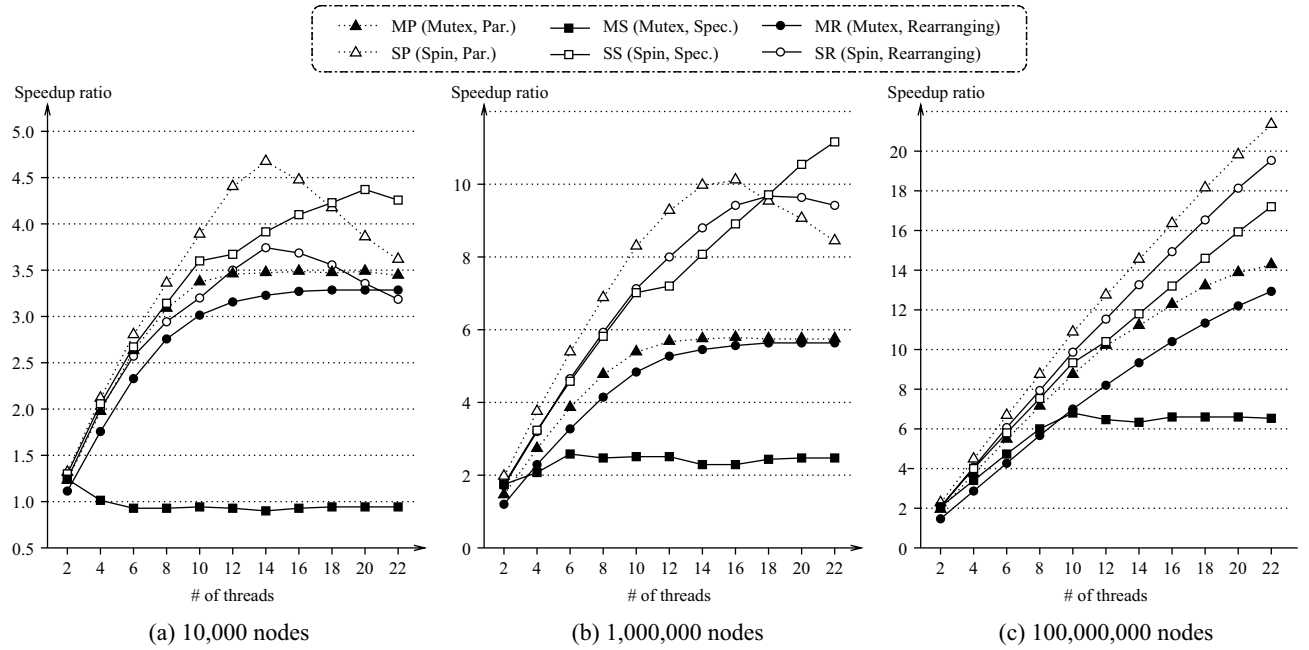


Fig. 3 Speedups in the case of one 24-thread Xeon, 8-node task

performance of SR degrades with more than 14 threads. The reinsertion of nodes may somewhat affect the performance, but we should understand the above reasons are dominant, because the performance of SP also degrades.

On the other hand, in the case of 100,000,000 nodes, Fig. 3c says the performances of SP and SR increase almost linearly in proportion to the number of threads. Enlarging the size of the BST makes its depth more profound, making T_w longer. Therefore, more threads can run practically in parallel. Moreover, since the number of leaf nodes increases approximately exponentially, the frequency of the races among threads to add their nodes to the same node is also reduced.

The computational complexity of SR is essentially larger than that of SP, because SR walks down twice in the BST to insert a node (although the second walk-down is merely for checking the cut-off of the path). What is more, the node reinsertion may occur in SR. The number of times SR tries to (re)insert nodes is more extensive than SP, and it also increases the occurrence of races among threads that add their nodes to the same node. Such performance differences are outstanding in the small size of BST, but as the size of the BST is larger, the frequency of the reinsertion decreases, because the insertion points may tend to be distributed (Fig. 3). In the case of 100,000,000 nodes, SR can generate the BST having the same structure as the sequential program with the sacrifice of less than 9.3% of the performance of SP.

When 10,000 nodes, SS is superior in performance to SR. On the other hand, when 1,000,000 and 100,000,000 nodes, SS is generally inferior to SR. Therefore, we can see the effectiveness of inserting nodes out-of-order compared to in-order node insertion. However, we can also see an unexpected result in Fig. 3b. With 18–22 threads, the performances of SP and SR are degrading, whereas that of SS is still increasing, eventually beyond the peak performances of SP and SR.

Naturally and generally, the performances of MP, MS, and MR are lower than those of SP, SS, and SR, respectively. When using the suspend lock, the waiting time to acquire a lock, including the suspending time, is significantly larger, in contrast to the case of spin lock which may disturb other working threads by pressuring the memory bandwidth caused by continuously repeated memory accesses. In the case of 10,000 and 1,000,000 nodes, we can see from Fig. 3a, b that the performances of MP and MR are almost saturated with more than 10 or 12 threads. This suggests that only 10 or 12 threads were doing practical work simultaneously and that other threads were suspended. We can interpret this saturation as the result that T_e—which includes the time, since a suspended thread is signaled to be awoken until it is practically resumed—dramatically increased in the model described in Sect. 5.2.1. On the other hand, in the case of 100,000,000 nodes, Fig. 3c tells the performances of MP and MR increase almost linearly in proportion to the

number of threads. This is because enlarging the BST size made T_w long. The performance of MR is less than that of MP by 22.4% at maximum in the case of 2 threads. However, the difference is more minor as the number of threads increases, and it is 9.4% in the case of 22 threads.

The performance of MS in the case of 10,000 and 1,000,000 nodes is disastrous. In those cases, T_w is short, whereas T_e is long. Since the number of working threads is limited to T_w/T_e, we can not expect performance improvement. Especially, in the case of MS with 10,000 nodes, T_e cannot be entirely hidden with T_w, so the performance is less than single-thread performance. In the case of 100,000,000 nodes, an increase in task size improves the performance of MS, but it is not enough to keep more than ten threads awake.

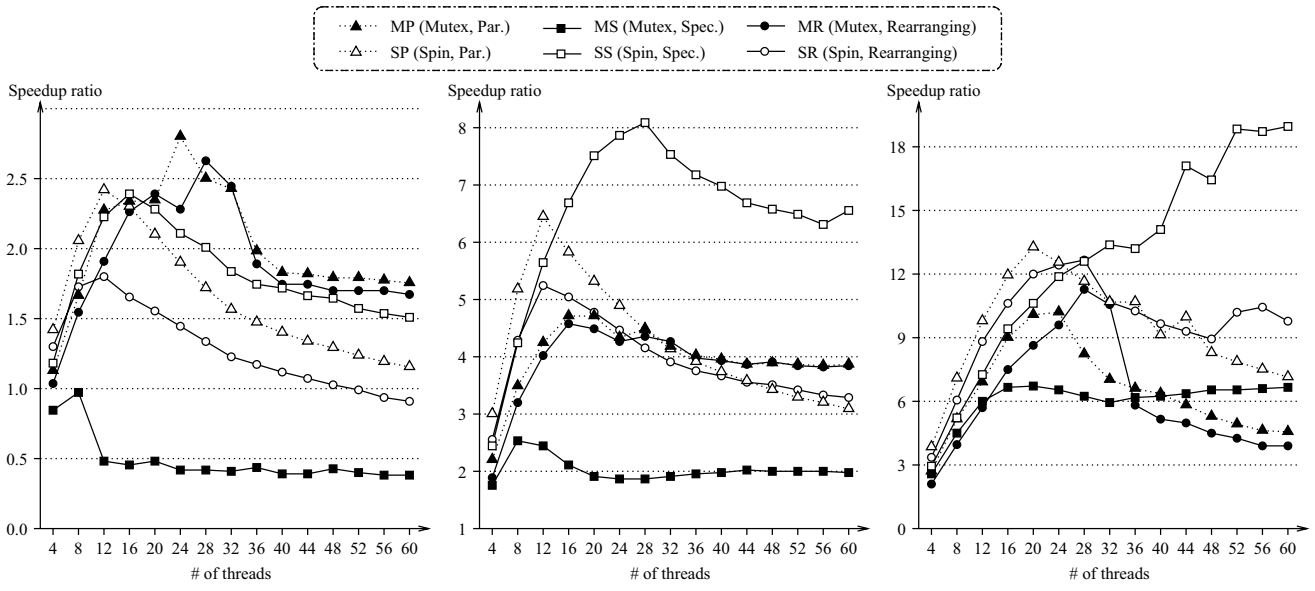
At last, Fig. 3c shows that parallel execution can sometimes achieve superlinear performance. We have no quantitative data to explain this over-speedup. Still, we guess it is brought by the scheduling performed by the OS through our experience monitoring processor cores' workloads (although we did not monitor when measuring the execution time, of course). The OS migrates a running thread from one processor core to another. Since the execution time of the single thread is relatively long, the thread is migrated many times. On the other hand, in the environment of parallel thread execution, the execution time is relatively short. And migration frequency seems small, perhaps because load unbalances among processor cores may also be improved. Therefore, we think that the migration overhead may enlarge the single-thread execution time, increasing the relative speedup of parallel execution.

5.2.3 Results of Two 32-Thread Xeon's and 8-Node Task

Figure 4 shows the speedup ratio when we ran the programs on machine B under a task size of 8. Machine B has two MPUs, and memory access is non-uniform. Therefore, thread allocation affects the performance more significantly than the case of machine A, tending to create various measurement results on every measurement.

Where the number of threads is less than 20, speedup ratios of SP, SS, and SR have similar tendencies to those shown in Fig. 3. However, they cannot achieve as significant speedup ratios as those on machine A. Linux usually allocates a new thread to as far processor core as possible, which may increase the distance between a thread and the memory bank storing necessary data for the thread. This may enlarge the data access delay or synchronization overhead. Thus, a machine with many MPUs cannot always greatly enhance the performance of practical parallel programs.

Figure 4 shows too many threads degrade the performance of SP and SR, including the case of 100,000,000 nodes. In Fig. 3c, such degradation was not observed, but on machine A, we should understand that 22-thread execution is still insufficient to degrade the performance. Increasing the number of threads may increase the frequency of the race among threads that try to add their nodes to the same node. It may also make the race to take a task from the source node list hot. For either of the races, repeated memory accesses to acquire locks pressure the memory bandwidth, damaging the overall performance.



(a) 10,000 nodes (b) 1,000,000 nodes (c) 100,000,000 nodes

Fig. 4 Speedups in the case of two 32-thread Xeon's, 8-node task

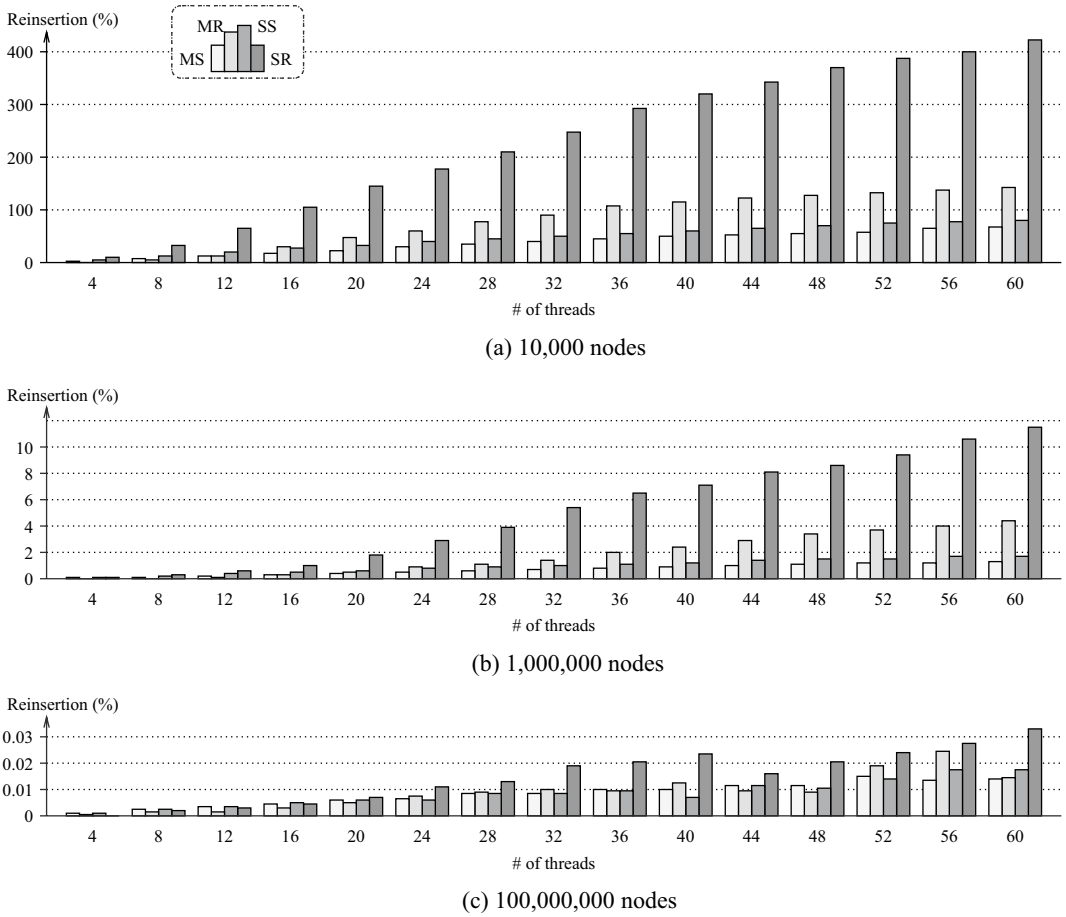


Fig. 5 Reinsertion ratios in the case of two 32-thread Xeon's, 8-node task

On that evidence, the performances of SP and SR are not always better or often worse than those of MP and MR.

Here, it is characteristic in Fig. 4b, c that SS is superior to SP and SR with more than 24 threads. This could also be interpreted that the performance degradations of SP and SR make the superiority of SS remarkable. However, in SS, a thread synchronizes one-to-one without using a lock and does not race against multiple other threads for synchronization. That is, it only tells the immediately succeeding thread that it has completed the commitment. Such lightweight synchronization helps to suppress the memory pressure brought by synchronization and saves the performance from degrading.

For the programs using the suspend lock, there are few new comments on the results. In Fig. 3c, the performances of MP and MR increase with the number of threads, which is almost the same in Fig. 4c. However, with more than 24 threads, Fig. 4c tells that the performances degrade. The reasons are races in synchronization, similar to the case of SP and SR. The suspend lock mechanism does not pressure the memory bandwidth, unlike the spin lock mechanism, but instead, the race among much more threads enlarges the overhead of lock/unlock operations themselves.

For reference, the reinsertion ratios are shown in Fig. 5. Compared with Machine A, they are generally higher. But even in the highest case (10,000 nodes, SR, 60 threads), it is less than 17%.

5.2.4 Results of POWER8 and 8-Node Task

Figure 6 shows the speedup ratio when we ran the programs on machine C under the task size of 8.

IIn many cases in Fig. 6, Par is superior in performance to Rear, which is superior to Spec. In Fig. 6b, SS is superior to SP or SR when more than 60 threads run. Similar superiority of SS is also shown in Figs. 3b and 4b, c. But here, in Fig. 6b, the peak performance of SS is not higher than that of SP. We should understand that the reason lies in the degradation of SPs performance rather than that SS achieved a higher performance over SP.

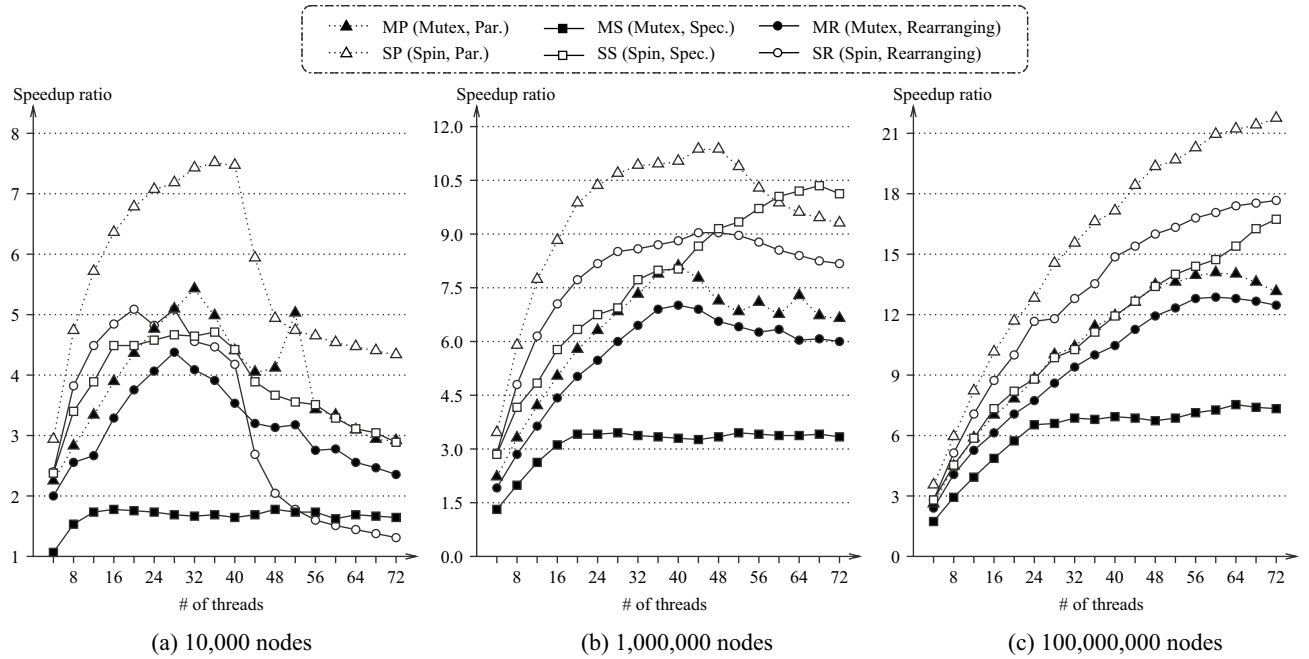


Fig. 6 Speedups in the case of POWER8, 8-node task

From Figs. 3, 4, and 6, we can model the performances outline as the following. The performance of the programs using the spin lock mechanism increases with increasing the number of threads and then degrades. SP achieves peak performance with a relatively small number of threads. SR does so with almost the same number or a slightly larger number of threads than SP. SS does so with a relatively large number of threads. The performances of MP and MR increase by increasing the number of threads and then become saturated or degrade. The performance of MS is saturated at a smaller number of threads than MP or MR. This means that T_e in MS is generally longer than T_e in MP/MR, although it is an agreeable result.

Figure 7 shows the node reinsertion ratio. The reinsertion ratio of SR with 10,000 nodes looks remarkable. In Fig. 6a, as the number of threads increases from 20 to 40, the performance of SR degrades, while the performance of SP still increases. That is, the difference in the performance between SR and SP becomes larger. We can infer that the node reinsertion affects the performance here, but the reinsertion ratio of SR is 8.0–19.9%. On the other hand, we can see that more than 40 threads are too many for the problem size of 10,000 nodes. However, when comparing with Fig. 14a, which is shown later, we can see that the performances of SP and SR degrade dominantly because of the reason concerning the task retrieval rather than the node (re)insertion. Therefore, we cannot discuss here the influence of the reinsertion ratio of more than 20%.

5.2.5 Results of M1 Ultla and 8-Node Task

Figure 8 shows the speedup ratio when we ran the programs on machine D under the task size of 8. Figure 9 shows the node reinsertion ratio.

Except for SP/SS/SR in Fig. 8a, Par is superior in performance to Rear, which is superior to Spec, similar to cases on many other machines. Figure 8c shows good scalability, which may be usual because of the execution with a few threads. When 100,000,000 nodes, SR achieves almost the same performance as SP. SR can build a BST with the same structure as the sequential construction with a 2.7–4.4% lower performance than SP.

5.2.6 Results of One 24-Thread Xeon and 64-Node Task

Figure 10 shows the relative speedup ratios of parallel versions to the sequential version of the BST construction program when we ran them on machine A. The task size was 64.

From the comparison of Fig. 10 with Fig. 3, we can see that the performances of SP and MP under $U = 68$ are generally improved from that under $U = 8$. It is remarkable, especially in the case of 10,000 and 1,000,000 nodes. As

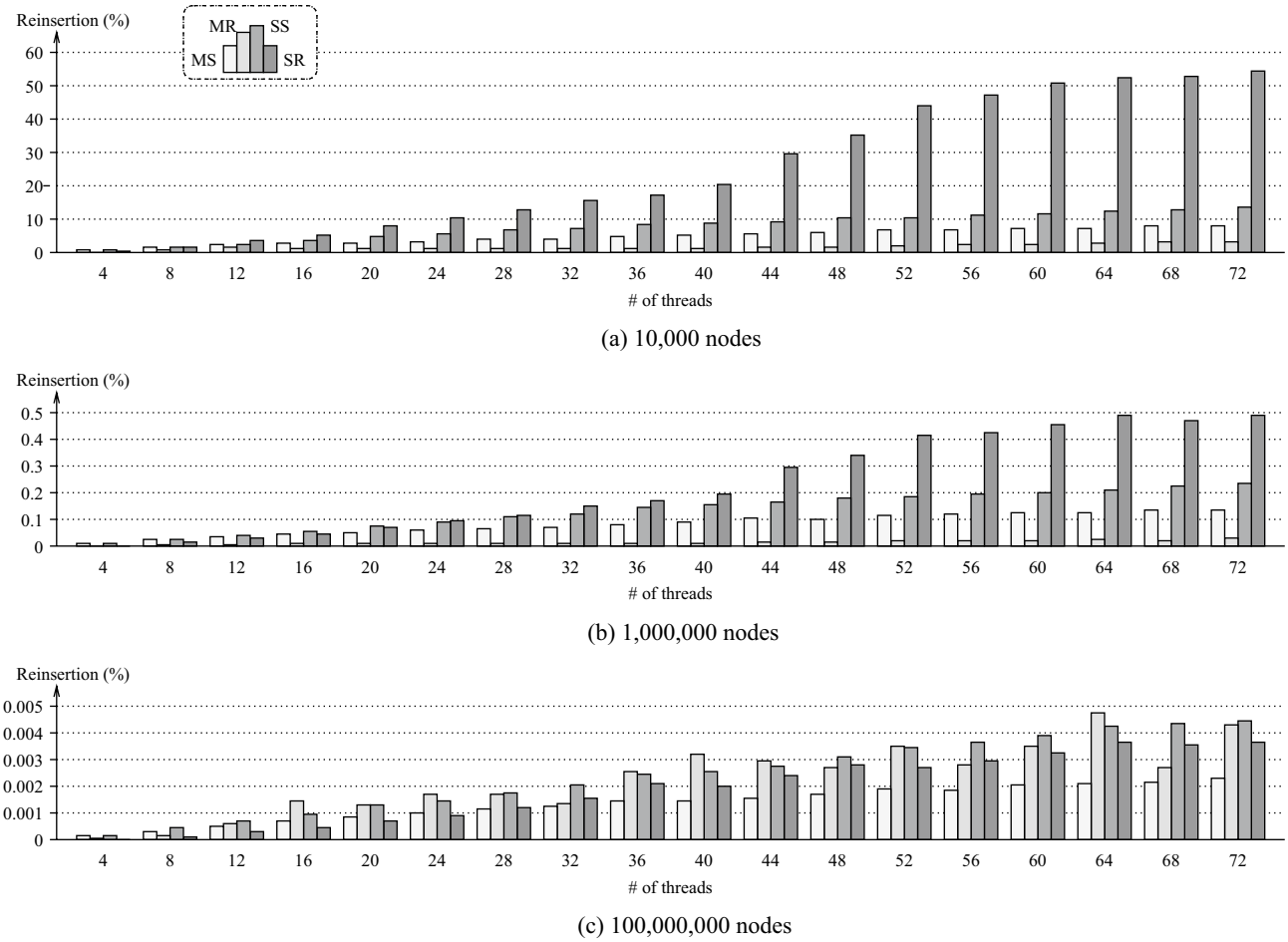


Fig. 7 Reinsertion ratios in the case of POWER8, 8-node task

increasing U, T_e increases, and T_w also does but in a larger proportion than T_e. Therefore, T_w/T_e increases, and the performance can be improved.

Figure 11 shows the node reinsertion ratio. Compared with Fig. 2, the reinsertion ratio dramatically increases in all cases. Especially, the reinsertion ratio of SR is outstanding. In the case of 10,000 nodes and 22 threads, the total execution number of the function insert() is over 2.5 times.

The performance of SR is improved in Fig. 10b, c, whereas in Fig. 10a, it is crucially spoiled. By increasing U, the much later nodes in the sequential insertion order are inserted earlier, causing more reinsertion, as shown in Fig. 11a. Figure 10 tells the performance damage due to such a penalty of reinsertion may be recovered for a large BST, but it is impossible for a small BST. For SS, similar results to SR are shown in Fig. 10, too. Therefore, increasing U is not profitable for speculation in the case of small BST.

In Fig. 10b, c, SP is superior in performance to SR, which is superior to SS, as is the usual cases under $U = 8$. However, for the programs using the suspend lock mechanism, we got unexpected results. Increasing U generally increases the frequency of misspeculation in SS (and SR, too). Therefore, it must make T_e longer, even when our repair mechanism of SSM alleviates the misspeculation penalty. Therefore, we thought that enlarging the task size was not preferable for speculative execution. However, Fig. 10b shows that MS is superior to MP in the range of up to 12 threads, and Fig. 10c shows that the performance of MS is almost the same as MP. Because the performance of MS is also almost the same as SS in Fig. 10c, we can understand that there were practically few threads suspended in MS. Thus, in situations with no suspended threads, MS can achieve excellent performance. That is, MS can build a BST with the same structure as the sequentially generated one without performance loss compared to the purely parallel construction.

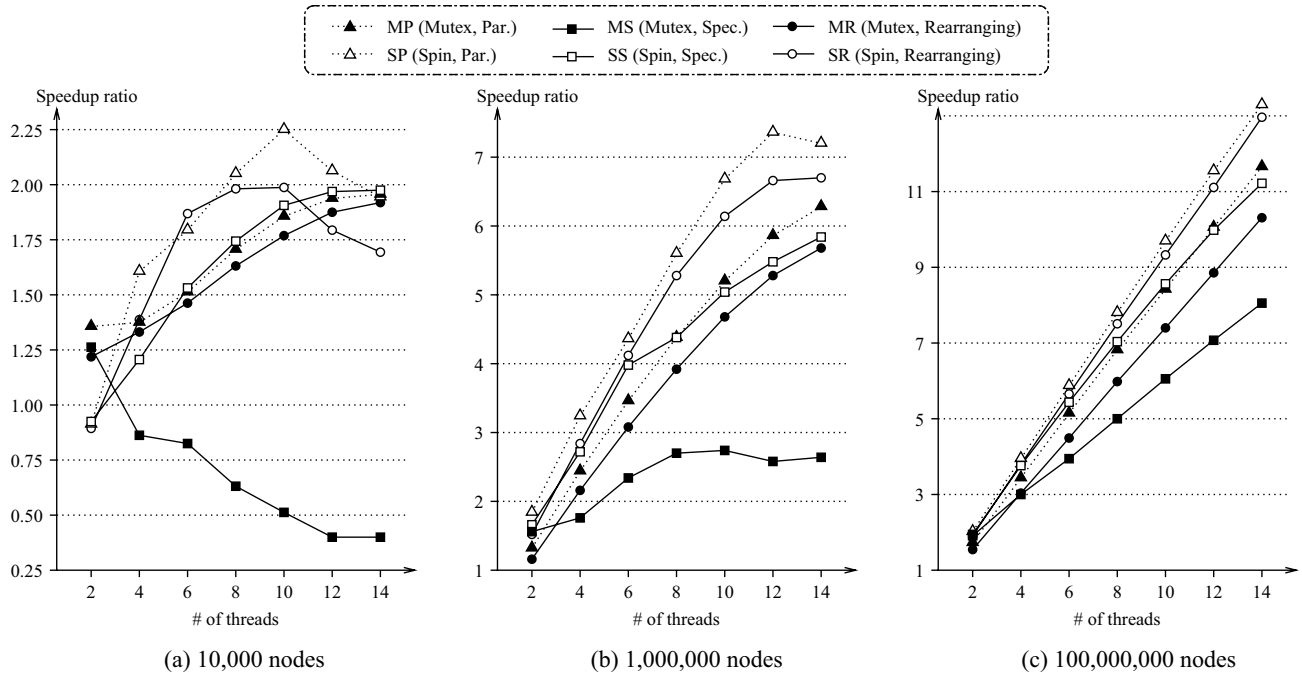


Fig. 8 Speedups in the case of M1 Ultra, 8-node task

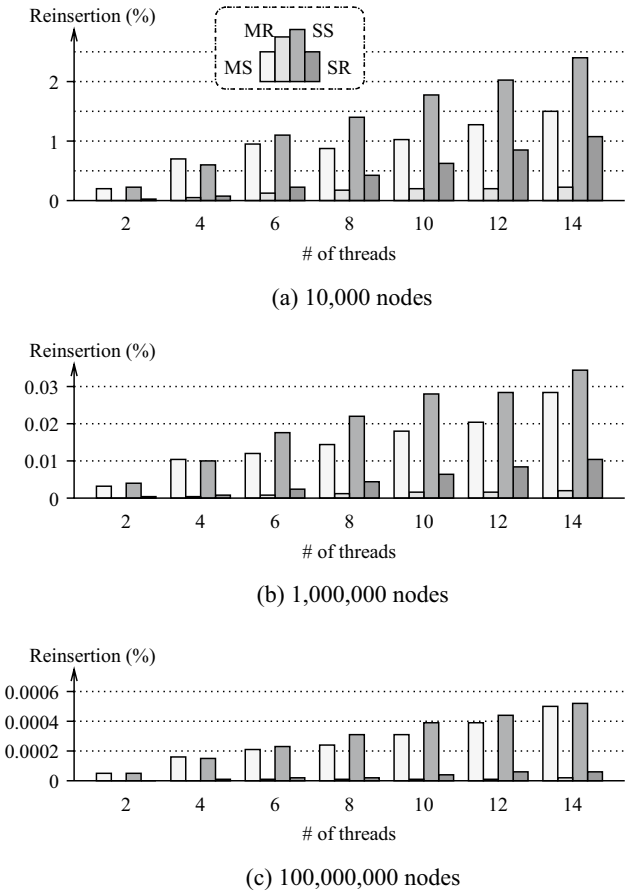


Fig. 9 Reinsertion ratios in the case of M1 Ultra, 8-node task

5.2.7 Results of Two 32-Thread Xeon's and 64-Node Task

Figure 12 shows the relative speedup ratios of parallel versions to the sequential version of the BST construction program when we ran them on machine B. The task size was 64. Where the number of threads is less than 20, Fig. 12 shows a similar tendency to Fig. 10, as in the relation between Figs. 4 and 3.

Performance improvement of SP is remarkable compared with the case of $U = 8$. It is due to an improvement of T_w/T_e first. Besides, by increasing U, the times a thread takes out its tasks from the source node list is reduced, and it helps to alleviate the race. Therefore, memory bandwidth pressure brought by the spin lock is lessened. Consequently, the performance degradation of SP, which was remarkable in Fig. 4, is alleviated in Fig. 12.

Figure 13 shows the node reinsertion ratio. The reinsertion ratios in Fig. 13 are still higher than those of Fig. 2. The reinsertion ratios of SR are outstanding not only in Fig. 13a but also in Fig. 13b. In the case of 1,000,000 nodes, the workload of SR is over 2 times with more than 16 threads and about 5.2 times with 60 threads. The workload of MR is also over 2 times with more than 36 threads.

The performance of SR is sensitive to U, because a large U tends to cause reinsertions. In the case of 10,000 nodes, the performance of SR is disastrous, and in the case of 1,000,000 nodes, it is improved with a small number of threads but degrades with many threads. The reason for these performance degradations is supported by high

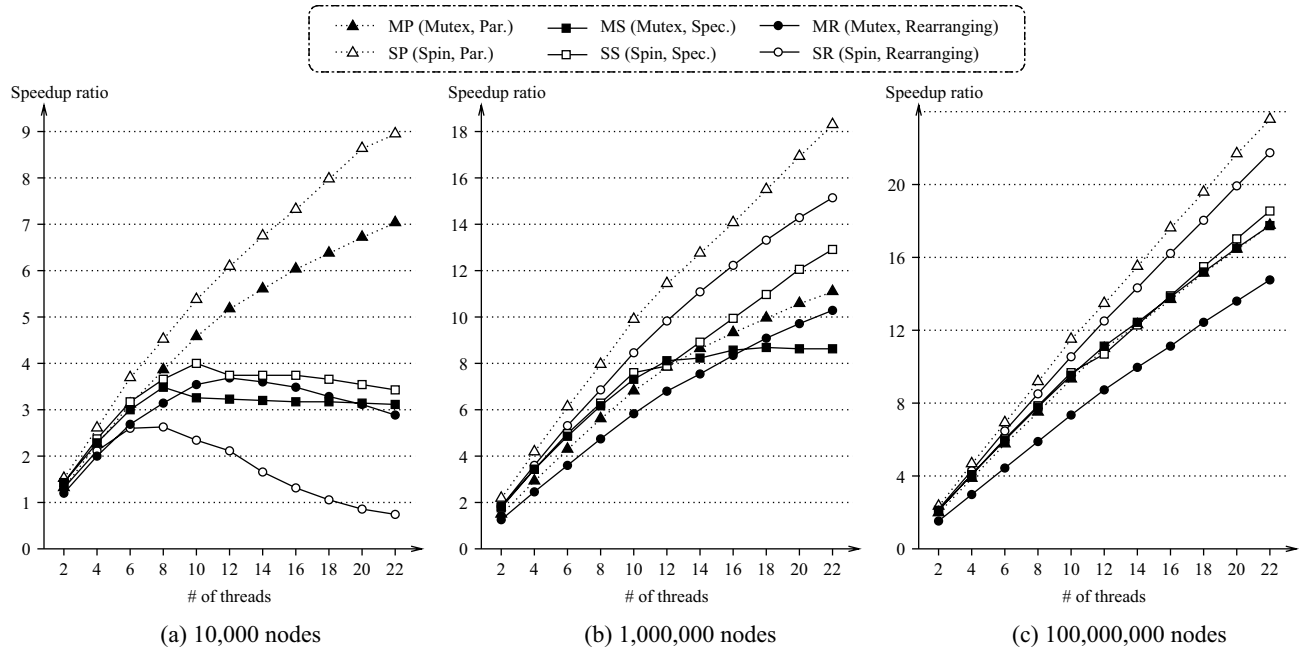


Fig. 10 Speedups in the case of one 24-thread Xeon, 64-node task

reinsertion ratios shown in Fig. 13a, b. However, in the case of 100,000,000 nodes, with 60 threads, it is almost the same as SP.

In contrast to SP, the performance of MP is not so good. The superiority between MS and MR is not also deterministic. Especially the curves in Fig. 12 are not smooth, and it is not easy to read the tendency of performances. It may be because differences in the distances between the processor core on which the thread is running and the memory bank storing node data affect the performance.

5.2.8 Results of POWER8 and and 64-Node Task

Figure 14 shows the speedup ratio when we ran the programs on machine C under task size $U = 64$. And Fig. 15 shows the node reinsertion ratio.

From comparing Fig. 14a with Fig. 6a, we can see a large U spoils the performance of Spec and Rear, especially for a small size of BST. It was also true in the cases of the Xeon (machines A and B). The reason is supported by too high reinsertion ratios shown in Fig. 15a. In Fig. 15a, b, the reinsertion ratios not only of SR but also of MR are outstandingly high. On the other hand, from the comparison of Fig. 14c with Fig. 6c, we can see the performances of SP/SS/SR in the case of 100,000,000 nodes which are merely a little improved. There are no such dramatic changes as seen on the Xeon Silver (machine B).

In the case of 10,000 nodes, Par performs better than Spec, which does better than Rear, independently from the difference of used lock mechanisms. By enlarging the BST size, the performance of Rear is improved to be better than Spec, and in the case of 100,000,000 nodes, Par performs better than Rear, which does better than Spec. This performance ranking is different from the cases of Xeon, while the performance of MS is almost the same as MR. Since the performance of MS is also almost the same as SS, we can guess threads in MS were rarely suspended. Thus, MS can build a BST with the same structure as the sequentially generated one with 3.6–23.9% lower performance than MP.

5.2.9 Results of M1 Ultla and 64-Node Task

Figure 16 shows the speedup ratio when we ran the programs on machine D under task size $U = 64$. And Fig. 17 shows the node reinsertion ratio.

From the comparison between Figs. 8a and 16a, in the case of 10,000 nodes, we can see that increasing U improves the performances of Par and Spec and worsens the performance of Rear (SR greatly but MR slightly). This is similar to the cases on other machines. On the other hand, the reinsertion ratios shown in Fig. 17 are all higher than those shown in Fig. 9 in the respective case. This is consistent with the fact that increasing U causes the performance to degrade, but it cannot explain why the performance improves. Consequently, we cannot discuss (or analyze) the performance only

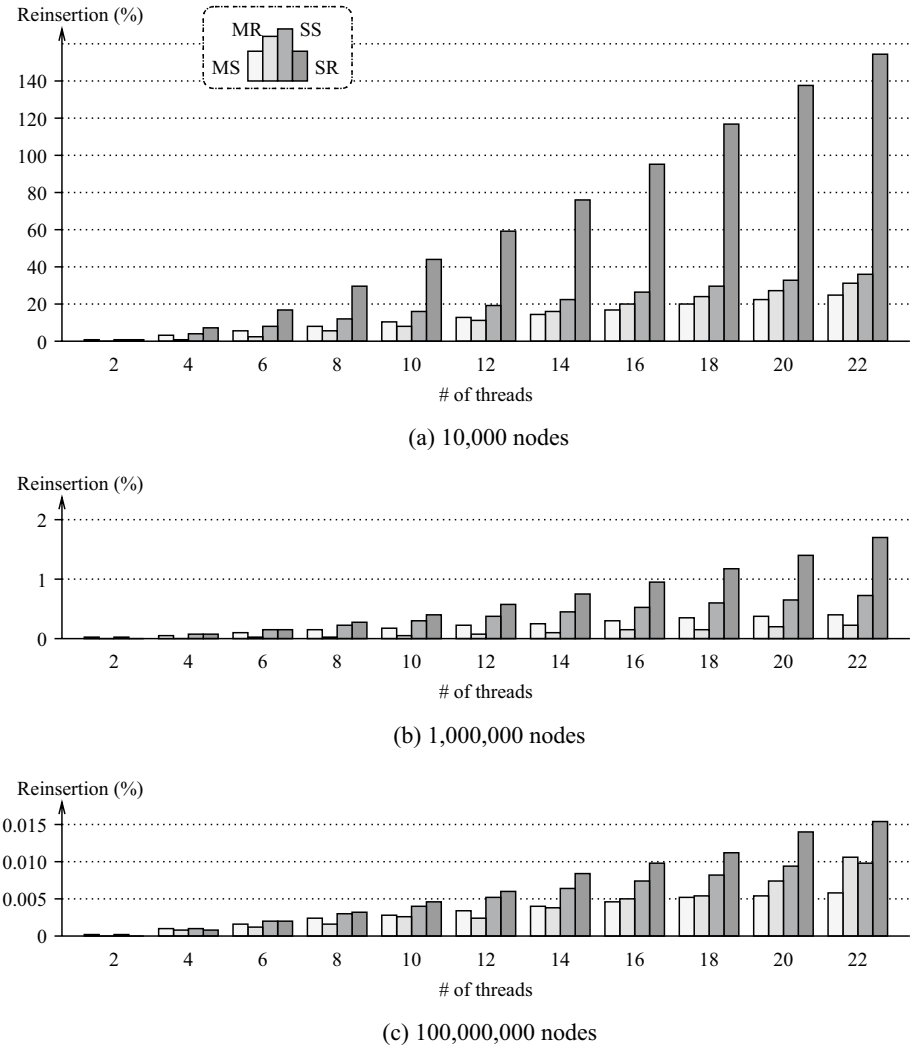


Fig. 11 Reinsertion ratios in the case of one 24-thread Xeon, 64-node task

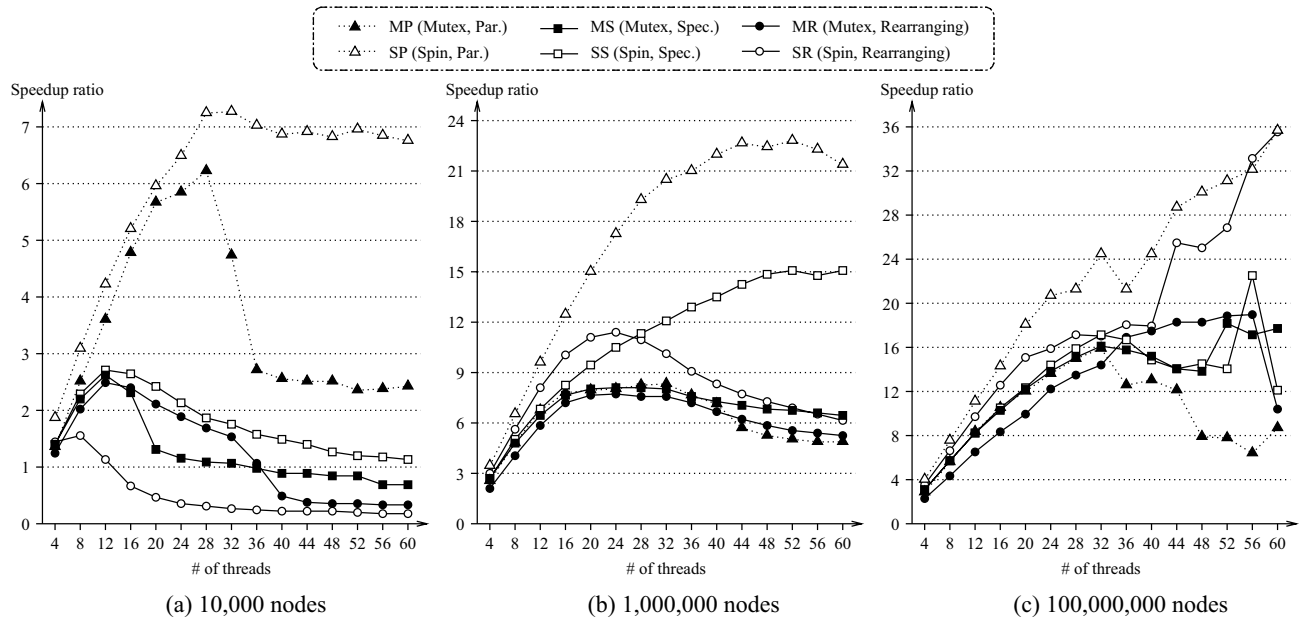


Fig. 12 Speedups in the case of two 32-thread Xeon's, 64-node task

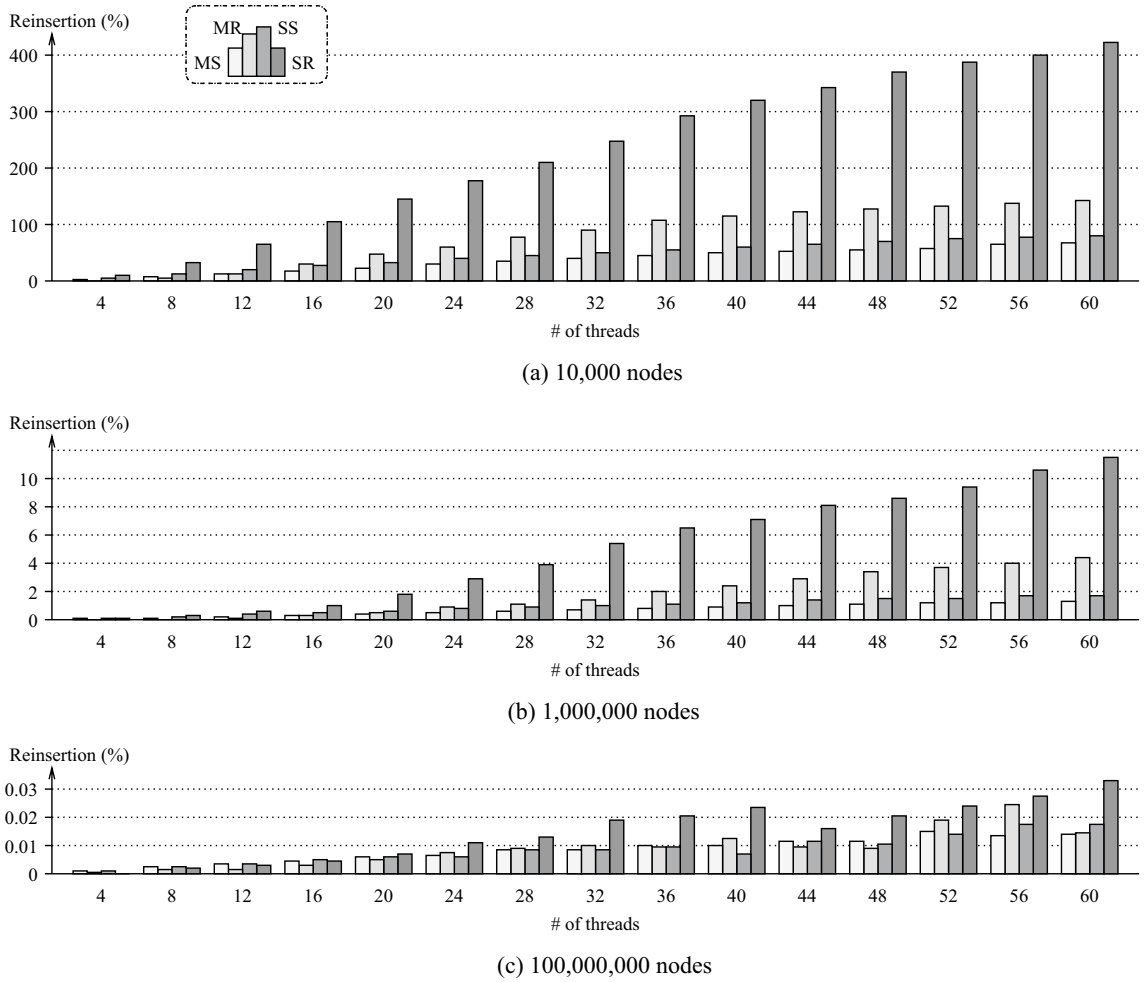


Fig. 13 Reinsertion ratios in the case of two 32-thread Xeon's, 64-node task

in the aspect of computational complexity. The efficiency of the synchronization, or removal of the waiting time, is also important, but it is not easy to measure the actual waiting time of the threads in practice.

In Fig. 16b, c, the performances of all programs are improved by enlarging the BST size. SP performs better than SR, and SR does than SS. However, the performance ranking in programs using suspend lock mechanism is different from those using the spin lock mechanism. MP performs better than MS, and MS does than MR. Such a difference in the ranking order between programs using different lock mechanisms was also shown in the cases of machines A and B.

Figure 16c shows that the performance of MS is slightly lower than SS. It suggests that threads were sometimes suspended to wait for their preceding threads. Nevertheless, consequently, MS achieves a very slightly higher performance than MP in Fig. 16c.

5.3 Discussion

We measured on some shared-memory types of multiprocessors. Still, the results differed depending on the machines because of the differences in the balance between processor speed and memory speed, memory bank organizations on NUMA, synchronization mechanisms, and resource utilization under SMT. This suggests that it is difficult to understand the universal superiority of algorithms (or parallel execution strategy) from a result measured on one machine. However, we can roughly summarize the results we obtained as the following.

- In most cases other than the small size of BSTs, SP is superior in performance to SR, and SR is superior to SS.
- However, SS sometimes achieves better peak performance than SP and SR.
- When the task size is small, MP is superior in performance to MR, and MR is superior to MS.

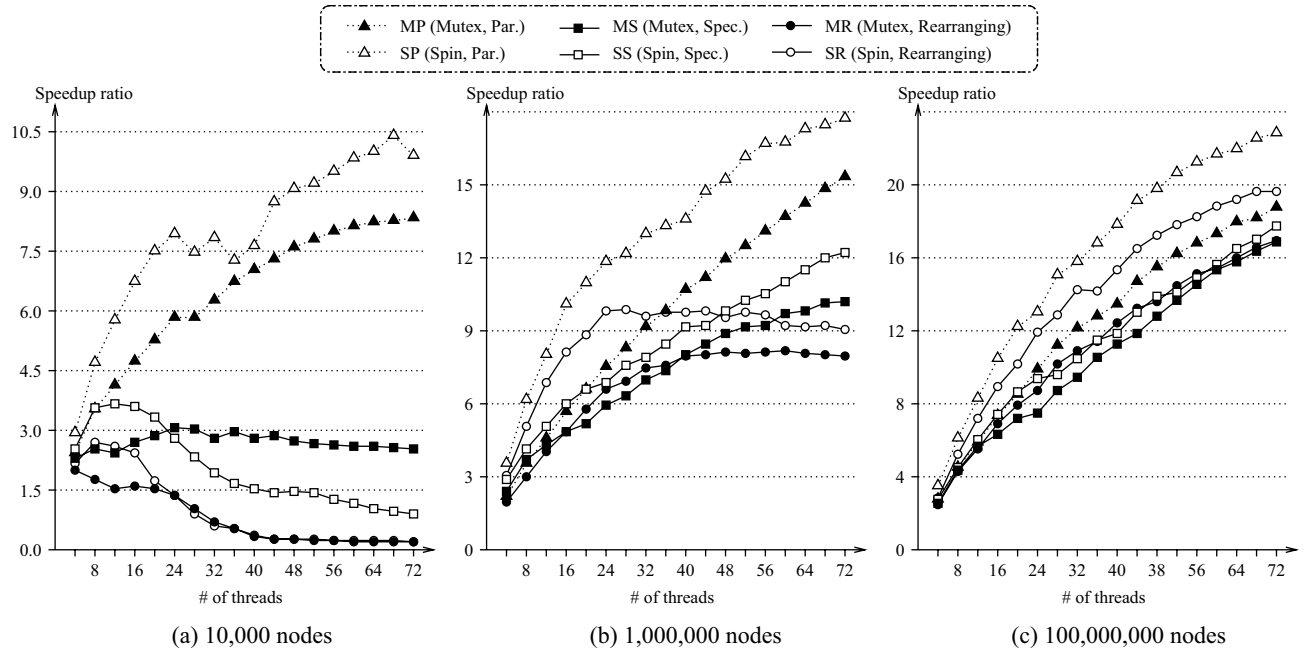


Fig. 14 Speedups in the case of POWER8, 64-node task

- Except for POWER8, however, when the task size is increased, the performance of MS is improved. For enough large-size of BST, MS is faster than MR and, sometimes, has equal speeds to MP.
- With a small size of BST, increasing the task size degrades the performances of SR and MR. On the other hand, with a large size of BST, increasing the task size improves the performances of SR and MR.
- Increasing the task size provokes misspeculation. In our experiment, the number of node (re)insertions grows nearly seven times at maximum with many threads and a small size of BST. An increase in the task size is most influential on SR among the speculation-based schemes. In contrast, with a small task size and a small size of BST, MR presents a smaller number of node (re)insertions than any other scheme.
- Even in the case of a large size of BST, misspeculation occurs, of course, and increasing the task size increases the number of reinsertions. However, they are often small (less than 1%) and seem ignorable.
- We measured the amount of node (re)insertions as one indicator concerning the computational complexity, but it does not always dominate the performance. What is more, it may be helpful to optimize a scheme, but it is not suitable to compare the performances between different schemes.

We evaluated both cases of using the spin and suspend lock mechanisms in this paper. The waiting overhead of lock operation primitives in the spin lock is smaller than in the suspend lock, and the program characteristics can be more easily reflectable in the performance measurement results. Since it helps us analyze the results by associating with the program behavior, it is significant to evaluate the case of using the spin lock mechanism experimentally. But generally, we should use the suspend lock in userland rather than the spin lock. Therefore, the result of programs using the suspend lock mechanism is more significant in practice than those using the spin lock mechanism, even if the performance superiority between them is reversed in the ideal environment.

We had believed a priori that Par is essentially superior in performance to Spec and Rear. It is indeed true in many cases of our evaluation results. But even, Par suffers from the overhead for mutual exclusion control. Therefore, in some cases, MS overcame MP in performance, being unexpected for us. Deep (excess) speculation increases the frequency of misspeculation. However, from our experimental results, we can see that deeper speculation does not always directly

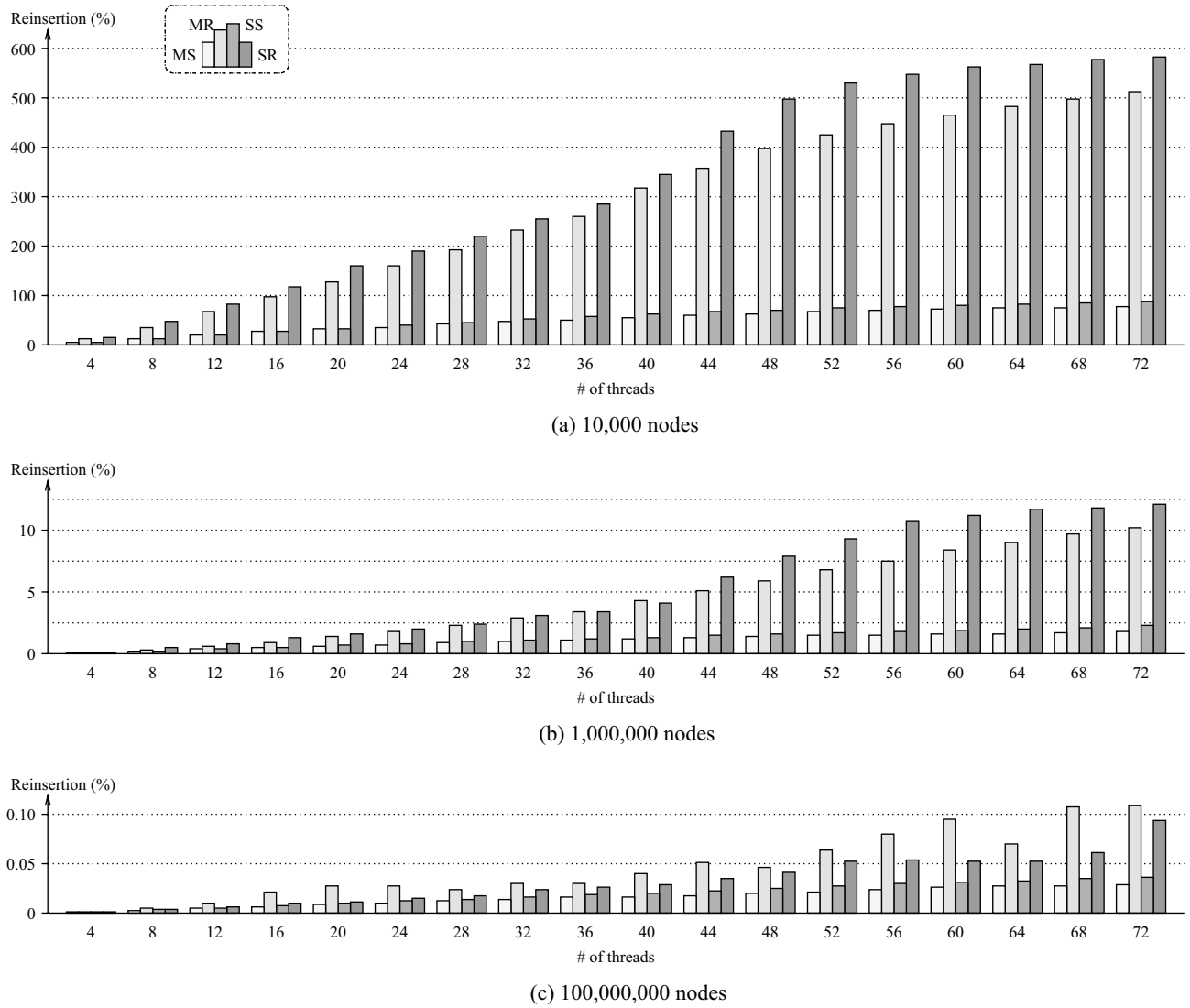


Fig. 15 Reinsertion ratios in the case of POWER8, 64-node task

bring performance degradation. Of course, the repair mechanism of our SSM library contributes to reducing the misspeculation penalty. Instead, the waiting time to commit is a bigger problem for MS. When we can enable threads to begin to commit without being suspended by increasing the task size, MS can achieve good performance. Consequently, MS can generate a BST with the same structure as sequential construction, taking an equal or shorter time to or than MP. But unfortunately, we do not yet have a technique to decide the optimum task size to maximize the performance of MS.

On the other hand, when the task size is small, it was shown that MR outperforms MS, because MR does not suffer from the waiting time to commit. However, MR is more sensitive to misspeculation than MS. Simply increasing the task sizes based on the loop unrolling fashion leads to deepening speculation and, as a result, increasing the frequency of misspeculation, so the performance of MR will either degrade or be enhanced slightly at best.

In conclusion, MR with a small task size is better for a small BST, and MS with a larger one is better for a large

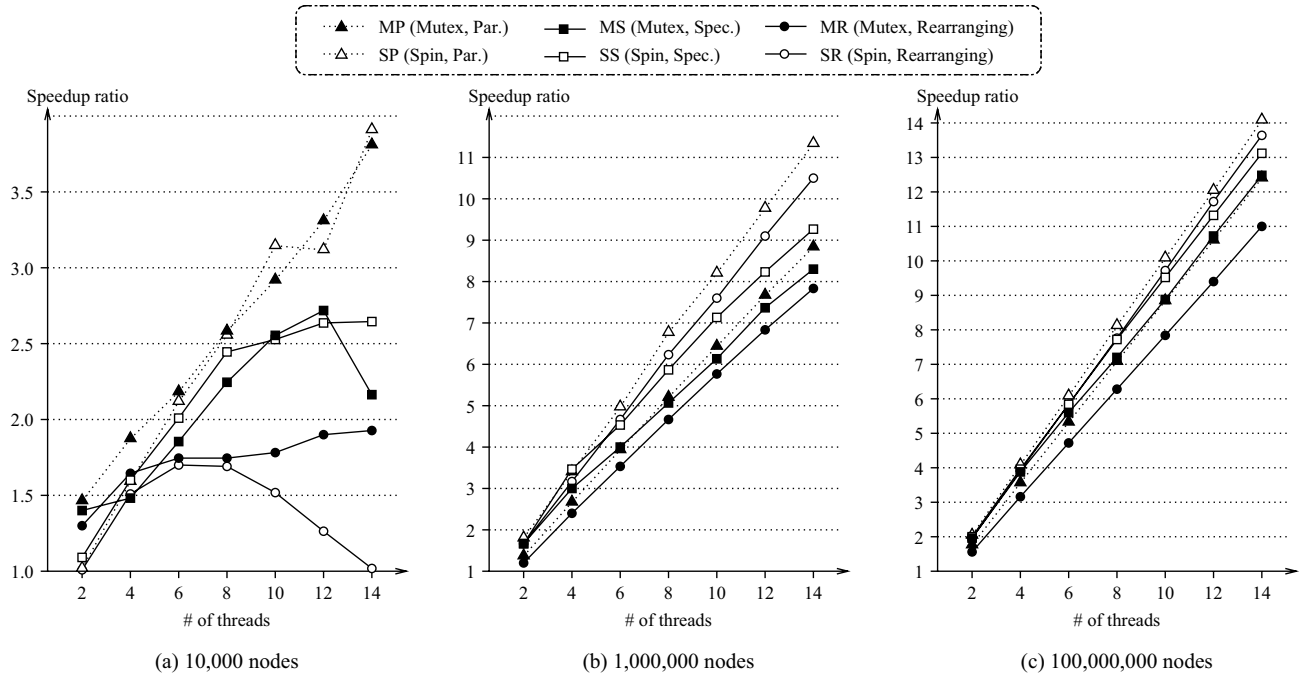


Fig. 16 Speedups in the case of M1 Ultra, 64-node task

BST. In other words, for the case that misspeculation is likely to occur in high frequency, we had better employ MR with a small task size rather than MS. On the other hand, when the frequency of misspeculation is relatively low, increasing the task size will improve the performance of MS beyond MR. In some cases, MS may achieve almost the same performance as MP.

6 Conclusion

This paper presented new parallel construction schemes of a BST with the same structure as the sequential construction and evaluated their performances on several types of shared-memory multiprocessors. For the large enough size of BST, our new parallel programs can construct a BST with always the same structure on a little lower or sometimes higher performance than the program that makes a BST with a different structure on every execution. And in contrast with the general expectation that enlarging the task size increases misspeculation and damages the performance, we found it sometimes enhances the performance of speculatively parallel execution.

We had intended to improve the performance of the first scheme (Spec) and developed the second (Rear). However, from our experiments in this paper, we saw that one does not always perform better than the other and that the superiority of one to the other changes depending on different conditions. Therefore, our future work is to integrate the two schemes to achieve the best performance always. Moreover, our further work is the development of

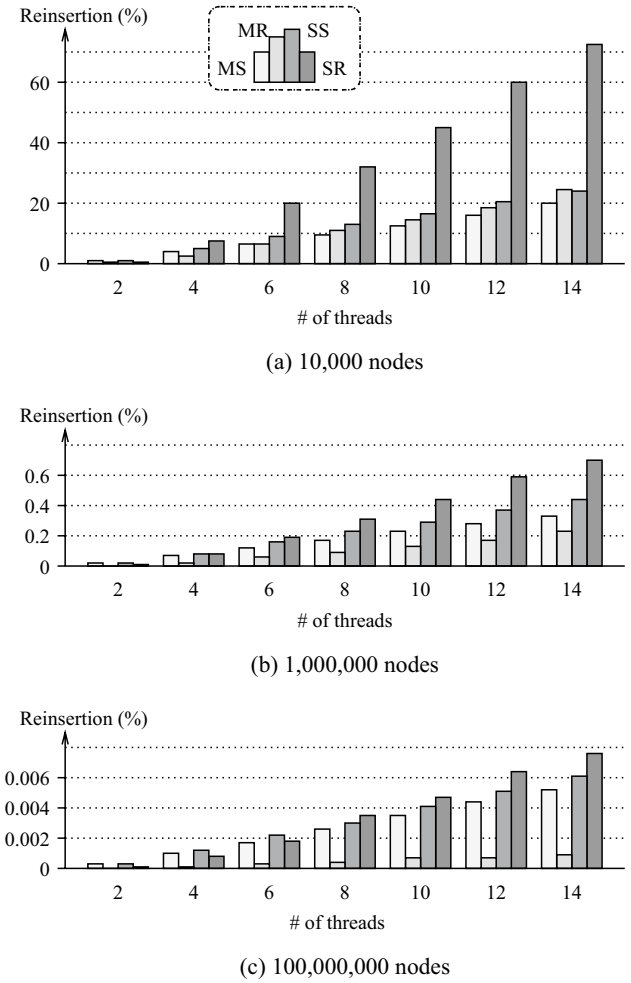


Fig. 17 Reinsertion ratios in the case of M1 Ultra, 64-node task

instruments to make our TLS-based technique applicable for applications other than constructing a BST.

Funding This article was funded by the Japan Society for the Promotion of Science (JSPS) KAKENHI (Grant No. JP21K11806), and the JSPS Core-to-Core program (Grant No. JPJSCCB2023005).

Data availability Raw data were generated at Kyoto Institute of Technology. Derived data supporting the findings of this study are available from the corresponding author on request.

Declarations

Conflict of interest The authors declare that they have no conflicts of interest.

References

1. Hirata H, Nunome A (2021) Reducing the repairing penalty on misspeculation in thread-level speculation. In: Proceedings of the 8th international virtual conference on applied computing & information technology (ACIT 2021). ACM, pp 39–45. https://doi.org/10.1145/3468081.3471120
2. Hirata H, Nunome A (2022) Parallel binary search tree construction inspired by thread-level speculation. In: Proceedings of the 23rd ACIS international summer virtual conference on software engineering, artificial intelligence, networking and parallel/distributed computing (SNPD 2022-Summer). IEEE, pp 74–81. https://doi.org/10.1109/SNPD-Summer57817.2022.00021
3. Hirata H, Nunome A, Shibayama K (2016) Speculative memory: an architectural support for explicit speculations in multithreaded programming. In: Proceedings of the 15th international conference on computer and information science (ICIS 2016). IEEE, pp 715–721. https://doi.org/10.1109/ICIS.2016.7550843
4. Fujisawa K, Nunome A, Shibayama K, Hirata H (2017) A software implementation of speculative memory. In: Proceedings of the 18th international conference on software engineering, artificial intelligence, networking and parallel/distributed computing (SNPD 2017). IEEE, pp 437–443. https://doi.org/10.1109/SNPD.2017.8022759
5. Matsunaga D, Nunome A, Hirata H (2019) Shelving a code block for thread-level speculation. In: Proceedings of the 20th international conference on software engineering, artificial intelligence, networking and parallel/distributed computing (SNPD 2019). IEEE, pp 427–434. https://doi.org/10.1109/SNPD.2019.8935751
6. Hirata H, Nunome A (2020) Decoupling computation and result write-back for thread-level parallelization. Int J Softw Innov (IJSI) 8(3):19–34. https://doi.org/10.4018/IJSI.2020070102
7. Feng J, Naiman DQ, Cooper B (2011) A parallelized binary search tree. J Inf Technol Softw Eng 1(1):1–5. https://doi.org/10.4172/2165-7866.1000103
8. Howley SV, Jones J (2012) A non-blocking internal binary search tree. In: Proceedings of the 24th symposium on parallelism in algorithms and architectures. ACM, pp 161–171. https://doi.org/10.1145/2312005.2312036
9. McKenney PE (2010) Memory barriers: a hardware view for software hackers. Preprint at https://www.researchgate.net/publication/228824849_Memory_Barriers_a_Hardware_View_for_Software_Hackers. Accessed 25 Mar 2023
10. Sewell P, Sarkar S, Owens S, Nardelli FZ, Myreen MO (2010) x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun ACM 53(7):89–97. https://doi.org/10.1145/1785414.1785443
11. Adve SV, Hill MD (1990) Weak ordering: a new definition. In: Proceedings of the 17th international symposium on computer architecture (ISCA '17). ACM, pp 2–14. https://doi.org/10.1145/325096.325100
12. Gharachorloo K, Lenoski D, Laudon J, Gibbons P, Gupta A, Hennessy J (1990) Memory consistency and event ordering in scalable shared-memory multiprocessors. In: Proceedings of the 17th international symposium on computer architecture (ISCA '17). ACM, pp 15–26. https://doi.org/10.1145/325096.325102

13. Ellen F, Fatourou P, Ruppert E, Breugel F (2010) Non-blocking binary search trees. In: Proceedings of the 29th symposium on principles of distributed computing (PODC '10). ACM, pp 131–140. https://doi.org/10.1145/1835698.1835736
14. Natarajan A, Mittal N (2014) Fast concurrent lock-free binary search trees. In: Proceedings of the 19th ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP '14). ACM, pp 317–328. https://doi.org/10.1145/2555243.2555256
15. Fatourou P, Papavasileiou E, Ruppert E (2019) Persistent non-blocking binary search trees supporting wait-free range queries. In: Proceedings of the 31st symposium on parallelism in algorithms and architectures (SPAA '19). ACM, pp 275–286. https://doi.org/10.1145/3323165.3323197
16. Akkary H, Driscoll MA (1998) A dynamic multithreading processor. In: Proceedings of the 31st annual international symposium on microarchitecture (MICRO-31). IEEE, pp 226–236. https://doi.org/10.1109/MICRO.1998.742784
17. Marcuello P, González A (1999) Clustered speculative multithreaded processors. In: Proceedings of the 13th international conference on supercomputing. ACM, pp 365–372. https://doi.org/10.1145/305138.305214
18. Hammond L, Hubbert BA, Siu M, Prabhu MK, Chen M, Olukotun K (2000) The Stanford hydra CMP. IEEE Micro 20(2):71–84. https://doi.org/10.1109/40.848474
19. Vijaykumar TN, Gopal S, Smith JE, Sohi G (2001) Speculative versioning cache. IEEE Trans Parallel Distrib Syst 12(12):1305–1317. https://doi.org/10.1109/71.970565
20. Cintra M, Torrellas J (2002) Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In: Proceedings of the 8th international symposium on high-performance computer architecture. IEEE, pp 43–54. https://doi.org/10.1109/HPCA.2002.995697
21. Steffan JG, Colohan CB, Zhai A, Mowry TC (2002) Improving value communication for thread-level speculation. In: Proceedings of the 8th international symposium on high-performance computer architecture. IEEE, pp 65–75. https://doi.org/10.1109/HPCA.2002.995699
22. Prabhu MK, Olukotun K (2003) Using thread-level speculation to simplify manual parallelization. In: Proceedings of the 9th ACM SIGPLAN symposium on principles and practice of parallel programming. ACM, pp 1–12. https://doi.org/10.1145/781498.781500
23. Garzaran MJ, Prvulovic M, Llaberia JM, Vinals V, Rauchwerger L, Torrellas J (2004) Software logging under speculative parallelization. High Perform Mem Syst. https://doi.org/10.1007/978-1-4419-8987-1_12
24. Steffan JG, Colohan C, Zhai A, Mowry TC (2005) The STAMPede approach to thread-level speculation. ACM Trans Comput Syst 23(3):253–300. https://doi.org/10.1145/1082469.1082471
25. Ohsawa T, Takagi M, Kawahara S, Matsushita S (2005) Pinot: speculative multi-threading processor architecture exploiting parallelism over a wide range of granularities. In: Proceedings of the 38th annual international symposium on microarchitecture (MICRO-38). IEEE, pp 81–92. https://doi.org/10.1109/MICRO.2005.26
26. Colohan CB, Ailamaki A, Steffan JG, Mowry TC (2006) Tolerating dependences between large speculative threads via sub-threads. In: Proceedings of the 33rd annual international symposium on computer architecture (ISCA '06). IEEE, pp 216–226. https://doi.org/10.1109/ISCA.2006.43
27. Praun C, Ceze L, Cascaval C (2007) Implicit parallelism with ordered transactions. In: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '7), pp. 79–89. https://doi.org/10.1145/1229428.1229443
28. Tian C, Feng M, Nagarajan V, Gupta R (2008) Copy or discard execution model for speculative parallelization on multicores. In: Proceedings of the 41st IEEE/ACM international symposium on microarchitecture (MICRO-41). IEEE, pp 330–341. https://doi.org/10.1109/MICRO.2008.4771802
29. Mehrara M, Hao J, Hsu P, Mahlke S (2009) Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In: Proceedings of the 30th ACM SIGPLAN conference on programming language design and implementation. pp 166–176. https://doi.org/10.1145/1543135.1542495
30. Hertzberg B, Olukotun K (2011) Runtime automatic speculative parallelization. In: Proceedings of the 9th Annual IEEE/ACM international symposium on code generation and optimization. IEEE, pp 64–73. https://doi.org/10.1109/CGO.2011.5764675
31. Odaira R, Nakaike T (2014) Thread-level speculation on off-the-shelf hardware transactional memory. In: Proceedings of the IEEE international symposium on workload characterization. pp 212–221. https://doi.org/10.1109/IISWC.2014.6983060
32. Shoji Y, Nunome A, Hirata H, Shibayama K (2015) A large-scale speculation for the thread-level parallelization. In: Proceedings of the 3rd international conference on applied computing and information technology (ACIT 2015). IEEE, pp 162–168. https://doi.org/10.1109/ACIT-CSI.2015.39
33. Salamanca J, Amaral JN, Araujo G (2016) Evaluating and improving thread-level speculation in hardware transactional memories. In: Proceedings of the IEEE international parallel and distributed processing symposium (IPDPS). IEEE, pp 586–595. https://doi.org/10.1109/IPDPS.2016.84
34. Hirata H, Kimura K, Nagamine S, Mochizuki Y, Nishimura A, Nakase Y, Nishizawa T (1992) An elementary processor architecture with simultaneous instruction issuing from multiple threads. In: Proceedings of the 19th annual international symposium on computer architecture (ISCA '92). ACM, pp 136–145. https://doi.org/10.1145/146628.139710
35. Tullsen DM, Eggers SJ, Levy HM (1995) Simultaneous multithreading: maximizing on-chip parallelism. In: Proceedings of the 22nd annual international symposium on computer architecture (ISCA '95). ACM, pp 369–380. https://doi.org/10.1145/225830.224449