Parallelization of Bulk Operations for STL Dictionaries

Leonor Frias[1,⋆] and Johannes Singler[2]

[1] Dep. de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya
[2] Institut für Theoretische Informatik, Universität Karlsruhe
lfrias@lsi.upc.edu, singler@ira.uka.de

Abstract. STL dictionaries like `map` and `set` are commonly used in C++ programs. We consider parallelizing two of their bulk operations, namely the construction from many elements, and the insertion of many elements at a time. Practical algorithms are proposed for these tasks. The implementation is completely generic and engineered to provide best performance for the variety of possible input characteristics. It features transparent integration into the STL. This can make programs profit in an easy way from multi-core processing power. The performance measurements show the practical usefulness on real-world multi-core machines with up to eight cores.

1 Introduction

Multi-core processors bring parallel computing power to the customer at virtually no cost. Where automatic parallelization fails and OpenMP loop parallelization primitives are not strong enough, parallelized algorithms from a library are a sensible choice. Our group pursues this goal with the Multi-Core Standard Template Library [7], a parallel implementation of the C++ STL. To allow best benefit from the parallel computing power, as many operations as possible need to be parallelized. Sequential parts could otherwise severely limit the achievable speedup, according to Amdahl's law. Thus, it may be profitable to parallelize an operation even if the speedup is considerably less than the number of threads.

The STL contains four kinds of generic dictionary types, namely `set`, `map`, `multiset`, and `multimap`. The first two are unique, i. e. no two equal elements may be included. Parallelization of the bulk operations of these containers has been one of the major missing parts of the MCSTL, so far. Now, we have implemented all variants, but limit our description to `set` in this paper, as it is one of the more general and more complicated ones, at the same time. The fine-grained basic operations, i. e. insertion, query and deletion of a single element, are not worth parallelizing because parallelization overhead would dominate their logarithmic running time. But often, many elements are inserted at a time, or a

⋆ Supported by ALINEX project (TIN2005-05446) and by grants number 2005FI 00856 and 2006BE-2 0016 of the *Agència de Gestió d'Ajuts Universitaris i de Recerca* with funds of the European Social Fund. This work was partially done while visiting Universität Karlsruhe.

L. Bougé et al. (Eds.): Euro-Par 2007 Workshops, LNCS 4854, pp. 49–58, 2008.

dictionary is constructed newly and initialized by a large sets of elements. Of course, the latter is a special case of the former, namely inserting into an empty dictionary. Nevertheless, we treat both cases explicitly, for best performance.

Our parallelization augments the STL implementation coming with the GNU C++ compiler (libstdc++), which uses red-black trees [4]. Red-black trees are balanced binary trees[1] whose nodes can be either colored red or black, the root always being black. The following invariant is maintained at all times: Each path from the root to a leaf must contain the same number of black nodes. This limits the tree height to $2\lceil \log n \rceil$. By performing so-called *rotations*, the invariant can be held up efficiently for all operations.

2 Algorithms

For the PRAM model, parallel red-black tree algorithms have been proposed [5]. They are highly theoretical, use fine-grained pipelining etc., and are thus not suitable for real-world machines. To our knowledge, there is neither an implementation nor experimental results. Nonetheless, some of the high-level ideas can be transferred to practical algorithms.

The STAPL [1] library does provide a parallel tree implementation, with an interface similar to the STL but it is not publicly available. It also works for distributed-memory systems, but makes a completely new implementation necessary. Its quite rigid partitioning of the tree can lead to all the elements being inserted by one processor, in the worst case. In contrast, we have implemented our bulk insertion operations on the top of the sequential data structure core, which stays unaffected. Our partitioning of the data is temporary for one operation and more flexible, e. g. multiple threads can work on a relatively small part of the tree. The cost of splitting the tree is negligible compared to the cost of creating/inserting the elements into the tree, for the common case.

As stated before, bulk construction can be seen as a special case of insertion. Indeed, we treat it the other way around here. Bulk construction (of a subtree) is the base case for bulk insertion. This way, we get good performance also for the case where many elements are inserted between two already contained elements (or conceptually, $-\infty$ and ∞ for bulk construction).

Before describing the parallel tree construction and the bulk insertion, we present the common setup for both, i. e. preprocessing the data to enable parallelization, and allocating node objects. Let p be the number of threads used, n the size of existing tree, and k the number of elements to insert and to construct the tree from, respectively.

Common Setup. The first step is to make sure the input sequence S is sorted, which also is a precondition in [5]. If two wrongly ordered elements are found, checking the order is aborted, and the input sequence is sorted stably, using the existing MCSTL implementations. Because the actual input sequence must not

[1] Note that hash data structures are not feasible here since STL dictionaries support sorted sequence functionality as well.

be modified, we need linear temporary storage here. Finally, S is divided into p subsequences of (almost) equal size.

Allocation and Initialization. Each thread t allocates and constructs the nodes for its subsequence S_t, which are still unlinked. The tree nodes are stored in an array of pointers, shared among the threads. This allows the algorithm to easily locate the nodes for linking, later on. If we wanted to avoid the array of pointers, we would need a purely recursive algorithm where allocation and initialization of the pointers are intertwined.

We also deal with equal elements in this step. In parallel, the surplus elements are removed from the sequence. A gap might emerge at the end of each S_t. Since we use stable sorting, we can guarantee that the first equal element becomes the one inserted, as demanded by the specification.

2.1 Parallel Tree Construction

Once the common setup has been performed, the tree is constructed as a whole by setting the pointers and the color of each node, in parallel. For each node, its pointers can be calculated independently, using index arithmetic. This takes into account the gaps stemming from removed equal elements, as well as the incompleteness of the tree. Each node is only written to by one thread, there is no need for synchronization. Thus, this is perfectly parallelized.

The algorithm constructs a tree that is complete except for the last level, which is filled partially from left to right. The last level is colored red, all other nodes are colored black.[2] Another option would be to also color every ℓ^{th} level red. The smaller ℓ, the more favored are deletions in the subsequent usage of the dictionary. The larger ℓ, the more favored are insertions.

2.2 Parallel Bulk Insertion

The problem of inserting elements into an existing tree is much more involved than construction. The major question is how to achieve a good load balance between the threads. There are essentially two ways to divide the work into input subsequences S_t and corresponding subtrees T_t. *1.* We divide the tree according to the input sequence, i. e. we take elements from the sequence and split the tree into corresponding subtrees. *2.* We divide the input sequence according to the tree, i. e. we split S_t by the current root element, recursively. Both approaches fail if very many elements are inserted at the end of the path to the same leaf. The first approach gives guarantees only on $|S_t|$, but not on $|T_t|$. The second approach does not guarantee anything about $|S_t|$, and bounds $|T_t|$ only very weakly: One of the subtrees might have twice the height as the other, so only $|T_1| < |T_2|^2$ holds. We use the first option in the initial step, and the second approach to compensate for greatly different $|T_t|$ dynamically.

[2] We could omit this if the tree is complete, but this is a rare special case. For a tree containing only one element, the root node is colored black anyway.

The split and concatenate operations on red-black trees [8,9] are the major tools in our parallelization of the bulk insertion operation.

Split into Multiple Subtrees. A red-black tree is split into p red-black trees, such that the elements will be distributed according to $p-1$ pivots.

Concatenate. Two range-disjoint red-black trees and a node "in-between" are concatenated to form a single red-black tree. Rebalancing might be necessary.

The whole algorithm consists of four phases. For coordinating the work, *insertion tasks* and *concatenation tasks* are generated, to be processed in the future.

Phase 0. Common setup, as described above.

Phase 1. Split the tree into p subtrees, the leftmost elements of the subsequences (except the first one) acting as splitters. This is performed in a top-down fashion, traversing at most $p-1$ paths, and generating $p-1$ concatenation tasks. With each concatenation task, we associate a node which is greater than all elements in the right subtree, but smaller than all elements in the right subtree. The algorithm chooses either the greatest element from the left subtree, or the least element from the subsequence to insert. This node will be used as tentative new root when concatenating the subtrees again, but might end up in another place, due to rebalancing rotations. The resulting subtrees and the corresponding subsequences form p independent insertion tasks, one for each thread.

Phase 2. Process the insertion tasks. Each thread inserts the elements of its subsequence into its subtree, using an advanced sequential bulk insertion algorithm (see Section 2.4).

Phase 3. Process the concatenation tasks to rebuild and rebalance the tree. This phase is not actually temporally separated from Phase 2, but only conceptually. As soon as one thread has finished processing an insertion task, it tries to process its parent concatenation task, recursively. However, this can only happen if the sibling subtask is also already done. Otherwise, the thread quits from this task and continues with another. It takes that one from its local queue or steals from another thread, if the own queue is empty. The root concatenation task does not have any parent, so the algorithm terminates after having processed it.

2.3 Analysis

We analyse the parallel time complexity of our algorithms. Assume for simplicity that the input sequence has already been preprocessed. To get rid of lower-order terms, we assume the number of elements in the tree being relatively large with respect to the number of threads, $n > p^2$.

Our construction algorithm takes $O(k/p)$ parallel time.

Calculating the worst-case cost for the bulk insertion algorithm is more involved. Splitting the tree sequentially into p parts takes $O(p \log n)$ time. As tests have shown, performing this step in parallel is not helpful assuming the number of cores currently available. In the worst case, one of the partitions contains a tree of size (almost) n, and the others contain (almost) empty trees.

Inserting a subsequence of size k/p sequentially into a tree of constant size takes $O(k/p)$ time. Inserting a sequence of size k/p sequentially into a tree of size n is upper bounded by the cost of inserting the elements one by one, i. e. $O(k/p \log n)$. Therefore, doing k insertions in p disjoint trees takes $O(k/p \log n)$ parallel time.

Finally, the p trees must be concatenated. Note that the concatenation operation itself is done sequentially but concatenations of disjoint trees will happen in parallel. A concatenation takes $O(\log n_1/n_2)$ sequential time, where n_1 is the size of the larger tree and n_2 is the size of the smaller subtree. Besides, the trees to be concatenated can differ in size by at most n elements. Therefore, one concatenation takes at most $O(\log n)$ time. Given that there are $O(\log p)$ levels of concatenations in total, the cost of concatenating all the trees is $O(\log p \log n)$.

As a result, the total cost of the operation is dominated by the insertion itself and therefore, it takes $O(k/p \log n)$ parallel time.

2.4 Dynamic Load-Balancing

The sequence of the elements to insert is perfectly divided among the threads. However, the corresponding subtrees might have very different sizes, which of course affects (wall-clock) running time negatively. Even worse, it is impossible to predict how elaborate the insertion will be, since this depends on the structure of the tree and the sequence. To counter this problem, we add dynamic load-balancing to Phase 2, using the randomized work-stealing [3] paradigm.

Each thread breaks down its principal insertion task into smaller ones. It goes down the subtree and divides the subsequence recursively, with respect to the current root. For each division, it creates the appropriate concatenation task, in order to reestablish the tree in Phase 3. The insertion task for the right subtree is pushed to the thread's work-stealing queue, while the recursion continues on the left subtree. However, the queue is only used if S_t is still longer than a threshold s.Otherwise, the algorithm works on the problem in isolation, to avoid the overhead of maintaining the queue. When there is nothing left to split, the bulk construction method is called for the remaining elements, and the new tree is concatenated to the leaf. If the subsequence has only a few elements left, the single-element insertion algorithm is called. As a side-effect, this gives us an more efficient sequential bulk-insertion algorithm, for s conceptually set to ∞.

To implement work-stealing, we use the lock-free double-ended queue provided by the MCSTL. It allows efficient synchronization of the work, using hardware-supported atomic operations. On the downside, its size must be limited at construction time. Fortunately, the size of all queues is bounded by the height of the tree, namely $2\lceil \log n \rceil$. In total, the approach is similar to the parallelized quicksort in the MCSTL [7, p. 6].

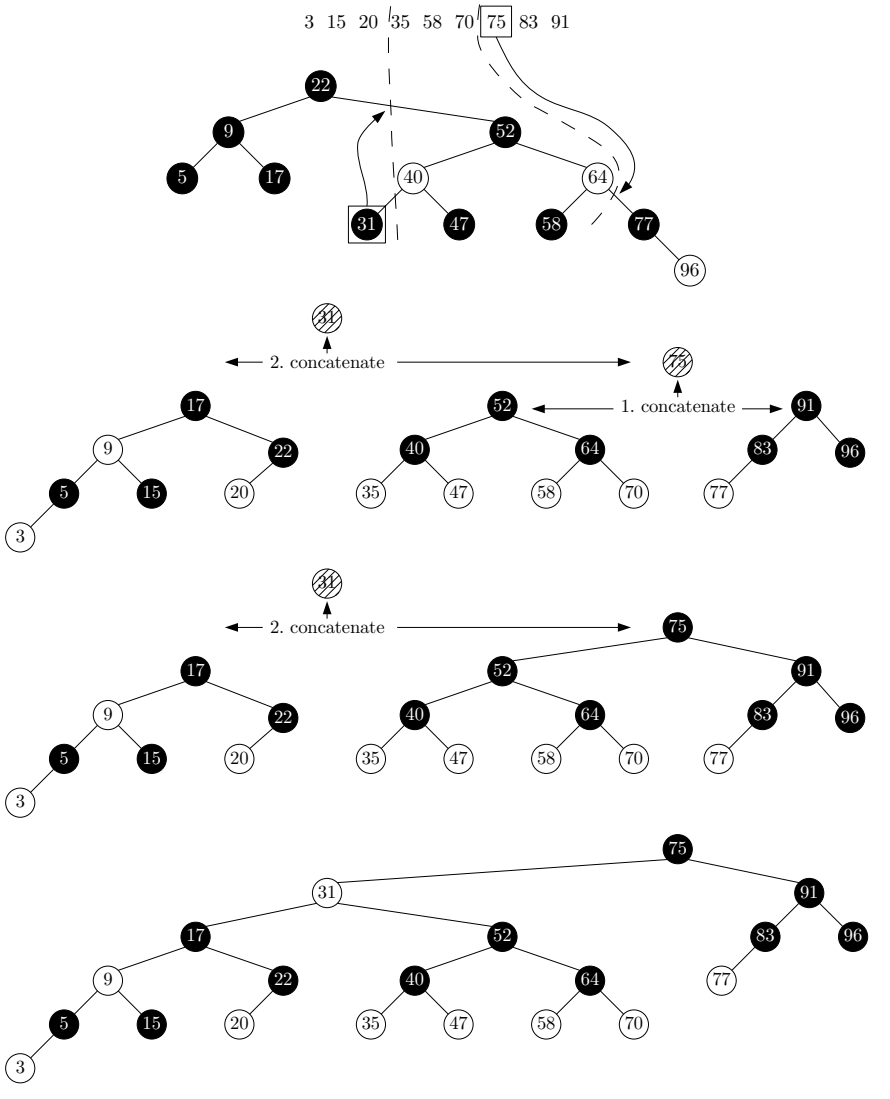

Fig. 1. Inserting into a tree, using three threads. "Red" nodes are shown transparent. *1.* Phase 0/1. The input sequence is divided equally, the tree is split accordingly. *2.* Phase 2. The two elements that will serve as the tentative roots in the future concatenation tasks are found to be 31 and 75. They are excluded from the subtree and not inserted to it, respectively. 58 is not inserted either, since it is a duplicate. *3./4.* Phase 3. Eventually, the concatenation task are processed. First, the right two subtrees are concatenated, using 75 as the new root. Then, the resulting tree is concatenated with the left one using 31 as the new root. In this case, a rotation is necessary for rebalancing purposes, so 75 ends up at the top position.

3 Interface and Implementation Aspects

Memory Management. For both bulk operations, allocating the node objects constitutes a considerable share of the work to be done. We cannot allocate several of them with one function call, since we need to be able to deallocate them one by one in case the user deletes elements from the data structure. The memory management concept of C++ does not allow such an asymmetric usage. Tests have shown that allocating memory concurrently scales quite well, but imposes some work overhead compared to the strictly sequential implementation. There also exist memory managers designed for this specific problem, e. g. Hoard [2]. We successfully used it on the Sun machine. However, for our 64-bit Intel platform, Hoard could not even match the performance of the built-in allocator, so we did not use it there.

Trade-Offs for Different Input Data and Comparator Types. STL dictionaries, apart from being parameterized by the data type, are also customized by a *comparator* type. This can lead to very different costs for comparing and copying an element. We also have to relate this to the cost of copying the node as a whole, which contains three pointers and the node color flag, in addition to the actual payload.

We tested this for sorting the input elements, experimentally. For large elements, it is better to sort pointers only, to avoid costly copying. For small elements, it is more cache-efficient to sort the elements directly.

4 Experimental Results

Experiments. We tested our program on two multi-processor machines with the following processors: *1.* Sun T1 (1 socket, 8 cores, 1.0 GHz, 32 threads, 3 MB shared L2 cache), *2.* Intel Xeon E5345 (2 sockets, 2×4 cores, 2.33 GHz, $2 \times 2 \times 4$ MB L2 cache, shared among two cores each). For the Intel machine, we used the Intel C++ compiler 10.0.25, on the Sun machine, we took GCC 4.2.0, always compiling with optimization (`-O3`) and OpenMP support. In both cases, we used the libstdc++ implementation coming with GCC 4.2.0. On the Xeon, compiling using the Intel compiler lead to more consistent results, due to the better OpenMP implementation, which is causing less overhead.

We have run each test at least 30 times and taken the average values for the plots, accompanied by the standard deviation range (too small to be seen in most cases). The following parameters concerning the input were considered:

Tree size/insertion sequence length ratio. Let n be the size of the existing dictionary/tree. Let $r = n/k$. Thus, $r = 0$ is equivalent to bulk construction.

Sortedness of the insertion sequence. The input sequence can be presorted or not. We focus on presorted insertion sequences here because parallel sorting is a separate issue.

Key data type. We show experiments on 32-bit signed integers.

Randomness of the input sequence. The input sequence by default consists of values in the range {RAND_MIN ... RAND_MAX}, but can optionally be limited to a smaller range, {RAND_MIN/100 ... RAND_MAX/100} here. This poses a challenge to the load-balancing mechanisms.

Both load-balancing mechanisms were switched on by default, but deactivated for some experiments, to show their influence on the running time.

For the insertion tests, we first build a tree of the appropriate size using the sequential algorithm, and then insert the elements using our parallel bulk insertion. This guarantees fair initial conditions for comparing the sequential and the parallel implementation. Actually, the shape of the existing tree does affect performance, though the sequential insertion is more affected than our algorithms.

Results for the Sequential Case. Our sequential algorithm is never substantially slower and in some cases substantially faster (e. g. Figures 2, 4 and 5). In particular, our specialized algorithms are very competitive when having random inputs limited to a certain range, being more than twice as fast (see Figures 8 and 9) as the standard implementation. This is because of saving many comparisons in the upper levels of the tree, taking advantage of the input characteristics.

Results for the Parallel Case. For the dictionary construction, our algorithms scale quite well, as shown in Figures 2 and 3. In particular, on the 8-core Sun T1, even the absolute speedup slightly exceeds the number of cores, culminating in about 11. This shows the usefulness of per-core multithreading in this case.

For insertion, our algorithm is most effective when the existing tree is smaller than the data to insert (see Figures 5 and 6). This is also due to the fast (sequential) insertion procedure, compared to the original algorithm. Splitting the input sequence is most effective in terms of number of comparisons because the subsequences left when reaching a leaf still consist of several elements.

In all cases, speedups of at least 3 for 4 threads and 5 for 8 threads, are achieved. The break-even is already reached for as little as 1000 elements, the maximum speedup is usually hit for 100000 or more elements.

The tree splitting step, used to make an initial balancing in the case of insertion, is shown to be really effective. For instance, compare Figures 6 and 7, which only differ in whether Phase 1 of the algorithm is run. We can see that both the speedup and the scalability are far better when the initial splitting is activated.

On top of that, the parallelization scales nicely, specifically when using dynamic load-balancing. Switching load-balancing off hurts performance for large inputs, while for small inputs it does not create considerable overhead. Dynamic load-balancing makes the algorithm more robust, comparing Figures 8 and 9.

As a by-product, we show the effects of mapping 2 threads to cores in different ways (see Figure 4). For small input, the most profitable configuration is sharing the cache, i. e. both threads running on the same die. As the input data

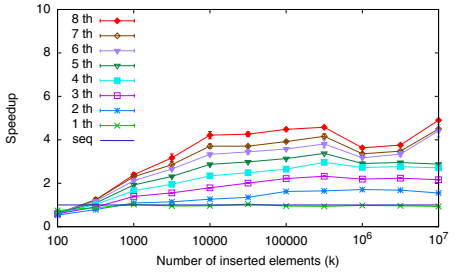

Fig. 2. Constructing a set of integers on the Xeon

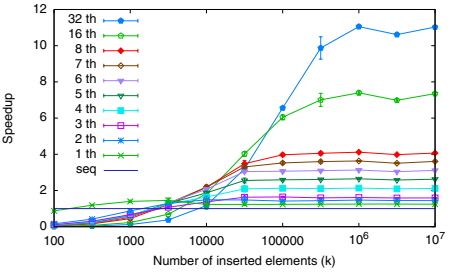

Fig. 3. Constructing a set of integers on the T1

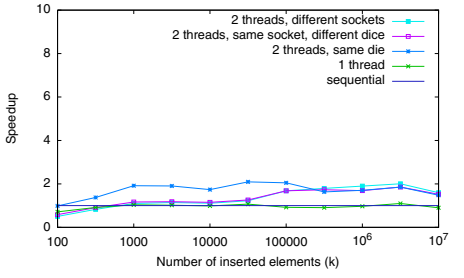

Fig. 4. Constructing a set of integers on the Xeon fixing two cores

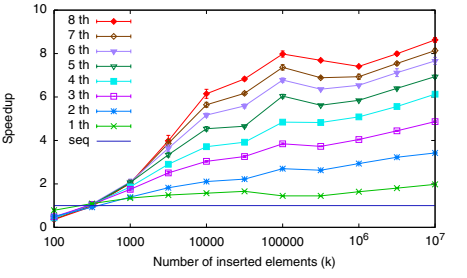

Fig. 5. Inserting integers into a set ($r = 0.1$) on the Xeon

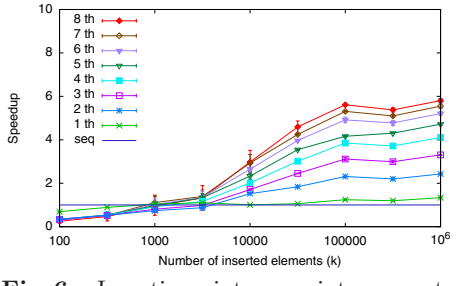

Fig. 6. Inserting integers into a set ($r = 10$) on the Xeon

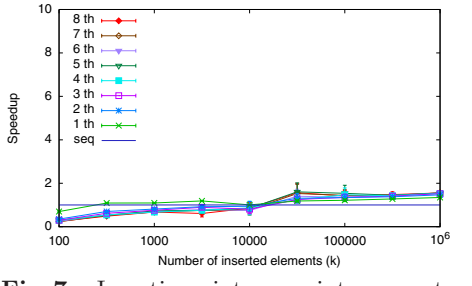

Fig. 7. Inserting integers into a set ($r = 10$) on the Xeon, without using initial splitting of the tree

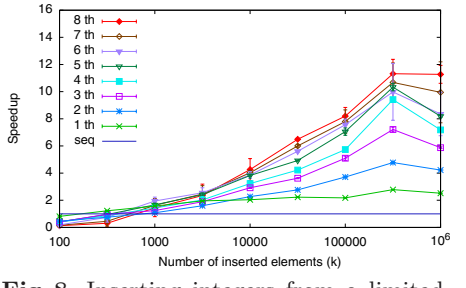

Fig. 8. Inserting integers from a limited range into a set ($r = 10$) on the Xeon

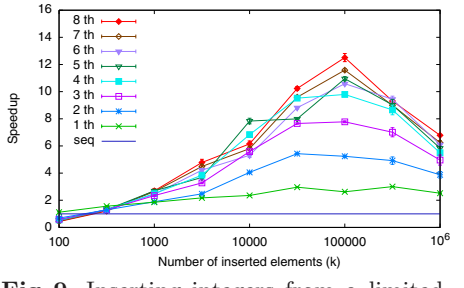

Fig. 9. Inserting integers from a limited range into a set ($r = 10$) on the Xeon, without using dynamic load-balancing

gets larger, memory bandwith becomes more important, so running the threads on different sockets wins. Running both threads on the same socket, but on different dice is in fact worst, because it combines both drawbacks. For the other tests, we have refrained to explicitly bind threads to specific cores, however, because the algorithms should be allowed to run on a non-exclusive machine in general.

5 Conclusion and Future Work

In this paper, we have shown that construction of and bulk insertion into a red-black tree can be effectively parallelized on multi-core machines, on top of a sequential implementation which remains unaffected. Our construction bulk operation shares some ideas with the theoretical algorithm in [5], giving a practical implementation. The code has been released in the MCSTL [6], version 0.8.0-beta.

To speed up programs that do not use bulk operations explicitly, we could use lazy updating. Each sequence of insertions, queries, and deletions could be split into consecutive subsequences of maximal length, consisting of only one of the operations. This approach could transparently be covered by the library methods. Then, for example, a loop inserting a single element in each iteration, would also benefit from the parallelized bulk insertion.

Another enhancement to the functionality is bulk *deletion* of elements. Since it is easy to remove a consecutive range from a tree, we will rather tackle the problem posed by a `remove_if` call, specializing it for the dictionary types.

References

1. An, P., Jula, A., Rus, S., Saunders, S., Smith, T., Tanase, G., Thomas, N., Amato, N.M., Rauchwerger, L.: STAPL: An Adaptive, Generic Parallel C++ Library. In: LCPC, pp. 193–208 (2001), `http://parasol.tamu.edu/groups/rwergergroup/research/stapl/`
2. Berger, E.D., McKinley, K.S., Blumofe, R.D., Wilson, P.R.: Hoard: A scalable memory allocator for multithreaded applications. In: ASPLOS-IX (2000)
3. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. JACM 46(5), 720–748 (1999)
4. Guibas, Sedgewick: A dichromatic framework for balanced trees. In: FOCS (1978)
5. Park, H., Park, K.: Parallel algorithms for red-black trees. Theoretical Computer Science 262, 415–435 (2001)
6. Singler, J.: The MCSTL website (June 2006), `http://algo2.iti.uni-karlsruhe.de/singler/mcstl/`
7. Singler, J., Sanders, P., Putze, F.: The Multi-Core Standard Template Library. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, Springer, Heidelberg (2007)
8. Tarjan, R.E.: Data structures and network algorithms. In: CBMS-NSF Regional Conference Series on Applied Mathematics, vol. 44, SIAM, Philadelphia (1983)
9. Wein, R.: Efficient implementation of red-black trees with split and catenate operations. Technical report, Tel-Aviv University (2005)