# MCSTL: The Multi-Core Standard Template Library

Felix Putze
Universität Karlsruhe
putze@ira.uka.de

Peter Sanders
Universität Karlsruhe
sanders@ira.uka.de

Johannes Singler
Universität Karlsruhe
singler@ira.uka.de

***Categories and Subject Descriptors*** D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.2.13 [*Software Engineering*]: Reusable Software—Reusable Libraries

***General Terms*** Algorithms, Performance, Design

***Keywords*** Multi-Core, Algorithm Library, Parallel Algorithms

## 1. Introduction

To benefit from the increased power of current and upcoming multi-core processors, programs have to exploit parallelism. This now becomes mandatory not just for a selected number of specialized programs, but for all nontrivial applications. This is a problem because automatic parallelization is still working only for simple programs and explicit parallelization is expensive and outside the qualification of most current programmers.

This poster addresses a third alternative — easy to use libraries of parallel algorithm implementations. We present initial work on the **M**ulti-**C**ore **S**tandard **T**emplate **L**ibrary. Parallelizing the C++ **S**tandard **T**emplate **L**ibrary is a good starting point since it is part of the C++ programming language and offers a widely-known, simple interface to many useful and efficient algorithms. Programs that use the STL can thus be partially parallelized by simple recompilation using the MCSTL.

We limit ourselves to shared memory systems with coherent caches and thus can offer features that would be difficult to implement efficiently on distributed memory systems. "Traditional" parallel computing works with many processors, specialized applications, and huge inputs. In contrast, the MCSTL should already yield noticeable speedup for as few as two cores for as many applications as possible. In particular, the amount of work submitted to each call of one of the simple algorithms in the STL may be fairly small. In other words, the tight coupling offered by shared memory machines in general and multi-core processors in particular, is not only an opportunity but also an *obligation* to scale *down* to small inputs rather than *up* to many processors.

For the reason of robustness, the MCSTL provides dynamic load balancing for many algorithms even when static load balancing would be enough on a dedicated machine. Our algorithms use heuristics to decide how much parallelism can be *efficiently* used, and reduces parallelism to this level, to avoid any performance degradation.



*PPoPP'07* March 14–17, 2007, San Jose, California, USA.


| Algorithm Class | Function Call(s) | Status | w/LB | w/oLB |
|---|---|---|---|---|
| Embarrassingly Parallel | `for_each`, `generate`, `generate_n`, `fill`, `fill_n`, `count`, `count_if`, `transform`, `replace`, `replace_if`, `min_element`, `max_element`, | impl | yes | yes |
| Find | `find`, `find_first_of`, `adjacent_find`, `find_if`, `mismatch`, `equal`, `lexicographical_compare` | impl | yes | nww |
| Search | `search`, `search_n` | impl | yes | nww |
| Numerical Algorithms | `accumulate`, `partial_sum`, `inner_product`, `adjacent_difference` | impl | planned | yes |
| Partition | `partition`, *`stable_partition`* | impl | yes | nww |
| Merge | `merge`, `multiway_merge`, *`inplace_merge`* | impl | planned | yes |
| Partial Sort | `nth_element`, `partial_sort` *`(_copy)`* | impl | yes | planned |
| Sort | `sort`, `stable_sort` | impl | yes | yes |
| Random Permutation | `random_shuffle` | impl | yes | nww |
| Containers | *`vector`, `(multi_)map/set`, `priority_queue` operations* | planned | | |
| Vector Arithmetic | *`valarray` operations* | planned | | |
| Complex Set Operations | *`set_union`, `set_intersection`, `set_difference`, `set_symmetric_difference`* | planned | | |
| Heap Construction | *`make_heap`, `sort_heap`* | planned | | |

**Table 1.** Legend: impl = already implemented, except those in *italics*, nww = not worthwhile, wLB / w/oLB = with / without dynamic load-balancing

***Related Work*** STAPL [1] provides parallel container classes that allow writing scalable parallel programs on distributed memory machines. However, judged from publications, only few of the STL algorithms have been implemented, and those that have been implemented sometimes deviate from the STL semantics.

We view the main contribution of this work as selecting good starting points and engineering efficient implementations for algorithms of the STL. Table 1 summarizes the current status of the implementation.

## 2. Algorithms

Several STL algorithms are *embarrasssingly parallel*, see Table 1. The MCSTL offers two major schemes to choose from: static equal distribution and efficient dynamic load-balancing using "work-stealing". In the latter, threads that run out of work steal jobs without the victim's interaction [2]. There is only little overhead due to using atomic operations and a user-tunable granularity. Dynamic

load balancing gives performance guarantees even in the case of highly heterogeneous jobs or inter-process interference.

`find` and related functions are hard to parallelize, since the running time is unpredictable. To avoid bad worst case execution times due to threading overhead, we start sequentially, and geometrically increase the assigned block size.

The implementation of `partition` is particularly useful as subprocedure for further functionality. We use a blocked strategy similar to [4] which is dynamically load-balanced. `nth_element` and `partial_sort` are based on it.

In addition to the standard (binary) `merge`, multiway merging in supported also, as an extension to the STL corpus. Both variants use *exact* multi-sequence partitioning [5] for perfect load-balance without any performance penalty, providing the first generic implementation of this algorithm we are aware of.

We implement two different parallel *sorting* algorithms that all have their merits:

*Multiway Mergesort* provides stability, performance guarantees and best performance. *Load-Balanced Quicksort* uses `partition` for the initial step and rebalances the recursion dynamically as described in [4], using a lock-free double-ended queue.

Our `random_shuffle` implementation provides both parallelism and cache-efficiency by using a hierarchy of bins [3], thus providing superlinear speedup.

## 3. Software Engineering

The MCSTL is based on OpenMP 2.5, which supports elegant programming, is very efficient due to thread pooling, and is highly portable. In addition, atomic processor operations are used through a thin platform-specific layer.

*Using* the MCSTL in a program is extremely simple: Create one symbolic link to the original STL, and append two include file search paths and the OpenMP switch to the compiler command. The user can rely on the library chosing the appropriate level of parallelism, or set this and other tuning parameters. All execution may be forced sequential at compile time, as well.

The MCSTL is available freely on our website[1], and can be used by everyone free of charge.

## 4. Experimental Results

Unless stated otherwise, our parallel algorithm implementations provide asymptotic linear speedup in the number of cores. Lower-order terms in time complexity for communication, have small constants, implying that often an implementation must be preferred which is not theoretically optimal in this sense.

As shown in Figure 2, sorting not only scales to the 8 cores provided by the Sun T1 processor, but also benefits from 4-fold software multi-threading, achieving speedups up to 22. Actually, such a behavior is typical for most algorithms on that machine. Speedup is already achieved for as little as about 3000 elements, when choosing an appropriate number of threads.

Computing the Mandelbrot fractal benefits a lot from using dynamic load-balancing (see Figure 1).

The exemplary results shown here also extend to other platforms we tested on.

## 5. Conclusions

We demonstrate that most algorithms of the STL can be efficiently parallelized on multi-core processors. Even nontrivial tasks like sorting with load-balancing achieve excellent speedup. Simultaneous multithreading has also been shown to have a great potential

[1] http://algo2.iti.uni-karlsruhe.de/singler/mcstl (Version 0.7.0 used here)

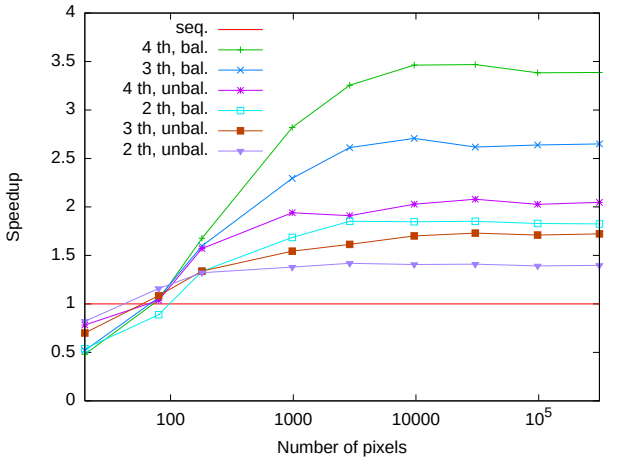


**Figure 1.** Mandelbrot fractal for $n$ pixels and a maximum of 1000 iterations per pixel on a 4-way Opteron.

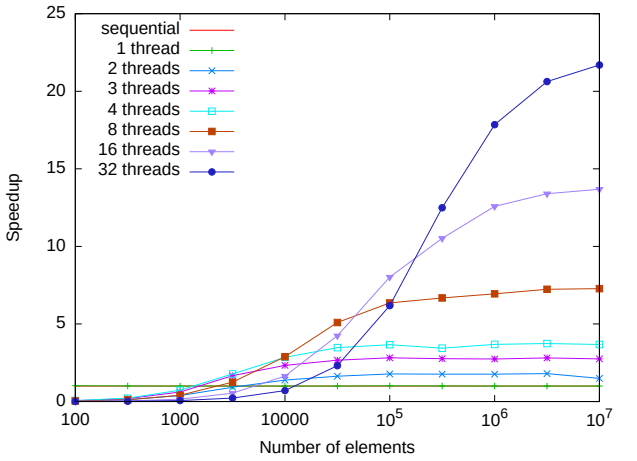


**Figure 2.** Sorting of pairs of 64-bit integers (mergesort).

in the algorithmic setting. The Sun T1 processor shows speedups far exceeding the number of cores when using multiple threads per core. Before, there have only been few experimental results in such a library setting.

Currently, we are investigating the parallelization of data structures operations, starting with bulk updates.

## References


[1] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. M. Amato, and L. Rauchwerger. STAPL: An Adaptive, Generic Parallel C++ Library. In *LCPC*, pages 193–208, 2001.

[2] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

[3] P. Sanders. Random permutations on distributed, external and hierarchical memory. *IPL*, 67(6):305–310, 1998.

[4] P. Tsigas and Y. Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on SUN Enterprise 10000. In *11th PDP*, page 372, 2003.

[5] P. J. Varman, S. D. Scheufler, B. R. Iyer, and G. R. Ricard. Merging Multiple Lists on Hierarchical-Memory Multiprocessors. *JPDC*, 12(2):171–177, 1991.