MCSTL: The Multi-core Standard Template Library*

Johannes Singler, Peter Sanders, and Felix Putze

Universität Karlsruhe
{singler,sanders,putze}@ira.uka.de

Abstract. Future gain in computing performance will not stem from increased clock rates, but from even more cores in a processor. Since automatic parallelization is still limited to easily parallelizable sections of the code, *most* applications will soon have to support parallelism *explicitly*. The *Multi-Core Standard Template Library (MCSTL)* simplifies parallelization by providing efficient parallel implementations of the algorithms in the C++ Standard Template Library. Thus, simple recompilation will provide partial parallelization of applications that make consistent use of the STL. We present performance measurements on several architectures. For example, our sorter achieves a speedup of 21 on an 8-core 32-thread SUN T1.

1 Introduction

Putting multiple cores into a single processor increases peak performance by exploiting the high transistor budget of modern semiconductor technology. The performance per Watt can also be improved, in particular when used together with reduced clock speeds and moderate instruction level parallelism. Dual-core processors being omnipresent, in the near future, many-cores will be used in virtually all areas of computing, ranging from mobile systems to supercomputers.

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

This paper addresses a third alternative — easy-to-use libraries of parallel algorithm implementations. While this approach has been successful in numerics for a long time, it has not yet made its way into the mainstream of non-numeric programming. We present our initial work on the **M**ulti-**C**ore **S**tandard **T**emplate **L**ibrary. Parallelizing the C++ **S**tandard **T**emplate **L**ibrary [2] is a good starting point since it is part of the C++ programming language and offers a widely-known, simple interface to many useful algorithms. Programs that use the STL can thus be partially parallelized by recompilation using the MCSTL.

* This is the full and updated version of a Poster Extended Abstract presented at PPoPP 2007 [1].

A.-M. Kermarrec, L. Bougé, and T. Priol (Eds.): Euro-Par 2007, LNCS 4641, pp. 682–694, 2007.

Parallelizing the STL is not an new idea. STAPL [3,4] 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 (e. g. `p_find` in [3]).

The MCSTL limits itself to shared memory systems and thus can offer features that would be difficult to implement efficiently on distributed memory systems, e. g. fine-grained dynamic load-balancing. Except for a few inherently sequential algorithms like binary search, the MCSTL will eventually parallelize all algorithms in the STL following the original semantics and working on ordinary STL random access iterators (e. g., STL vectors or C arrays). This approach brings its own challenges. "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 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. Another issue is that the MCSTL should coexist with other forms of parallelism. The operating system will use some of the computational resources, multiple processes may execute on the same machine and there might be some degree of high-level parallelization using multi-threading within the application. These methods are the easiest way to leverage the power of multi-core processors but might not suffice to fully saturate the machine. In this context, the MCSTL should have dynamic load balancing even when static load balancing would be enough on a dedicated machine. Moreover, parallel algorithms that achieve some limited speedup at the cost of a great increase of total work should be avoided. A better solution are algorithms that use only as much parallelism as can be *efficiently* exploited. Our algorithms use some heuristics to decide on the level of parallelism. This form of efficiency is also important with respect to energy consumption.

More Related Work. Recently, another shared memory STL library has surfaced. MPTL [5] parallelizes many of the simple algorithms in STL using elegant abstractions. However, it does not implement the more complicated parallel algorithms `partition`, `nth_element`, `random_shuffle`, `partial_sum`, `merge`. MPTL has a "naïve" parallelization of quicksort using sequential partitioning. Similarly, there is only a "naïve" implementation of `find` that does not guarantee any speedup even if the position sought is far away from the starting point. MPTL offers a simple dynamic load balancer based on the master worker scheme and fixed chunk sizes.

Most of the algorithms we present here are previously known or can be considered folklore even if we do not cite a specific reference. We view the main contribution of this paper as selecting good starting points and engineering efficient implementations. Interestingly, many of our algorithms were originally developed for distributed memory parallel computing. Very often, such algorithms

naturally translate into shared memory algorithms with good cache locality and few costly synchronization operations. The key ideas behind our sorting algorithms are also not new [6,7] but not widely known. It is somewhat astonishing that although there are virtually hundreds of papers on parallel sorting, so few notice that multiway merging can be done with exact splitting, and that the partition in quicksort can be parallelized without changing the inner loop.

Notation. In general, n will refer to the problem size, m to some secondary problem quantity. Often, we are dealing with a sequence $[S[0], \ldots, S[n-1]]$. There are p threads that run in parallel, numbered 0 through $p-1$.

2 Algorithms

We plan to implement all the algorithms provided in the STL for which parallelization looks promising. Figure 1 summarizes the current status of the implementation.

Algorithm Class	Function Call(s)	Status	w/LB	w/oLB
Embarrassingly Parallel	`for_each`, `generate(_n)`, `fill(_n)`, `count(_if)`, `transform`, `unique_copy`, `min/max_element`, `replace(_copy)(_if)`	impl	yes	yes
Find	`find(_if)`, `mismatch`, `equal`, `adjacent_find`, `lexicographical_compare`	impl	yes	nww
Search	`search(_n)`	impl	yes	nww
Numerical Algorithms	`accumulate`, `partial_sum`, `inner_product`, `adjacent_diff`.	impl	nww	yes
Partition	*(stable_)*`partition`,	impl	yes	nww
Merge	`merge`, `multiway_merge`, *`inplace_merge`*	impl	pl	yes
Partial Sort	`nth_element`, `partial_sort`*(_copy)*	impl	yes	pl
Sort	`sort`, `stable_sort`	impl	impl	yes
Shuffle	`random_shuffle`	impl	yes	nww
Set Operations	`set_union`, `set_intersection`, `set_(symmetric_)diff`.	impl	nww	yes
Vector	*`valarray operations`*	pl		
Containers	*`vector`, `(multi_)map/set`, `priority_queue` operations*	pl		
Heap	*`make_heap`, `sort_heap`*	pl		

Fig. 1. Considered STL functions. already **impl**emented, (except *italicized*), **pl**anned, **n**ot **w**orth**w**hile, wLB / w/oLB = with / without dynamic load-balancing

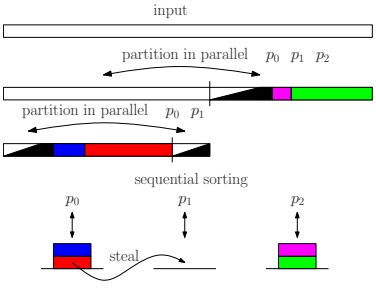

Fig. 2. Schema of parallel balanced quicksort. The ramped parts are already sorted, while the dark grey parts are currently partitioned. The colored parts are remembered on the stack and wait for being (stolen and) processed.

Embarrassingly Parallel Computation. Several STL algorithms may be viewed as the problem of processing n independent atomic jobs in parallel, see the row "Embarrassingly Parallel" in Figure 1 for a list. This looks quite easy

on the first glance. For example, we could simply assign $\leq \lceil n/p \rceil$ jobs to each thread. Indeed, we provide such an implementation in order to scale down to very small, fine-grained, and uniform inputs on dedicated machines. However, in general, we cannot assume anything about the availability of cores or the running time of the jobs which might reach from a few machine instructions (quite typical for STL) to complex computations. Hence, we can neither afford dynamic load balancers that schedule each job individually nor should we a priori cluster chunks of jobs together. *Random Polling* or *randomized work stealing* is a way out of this dilemma.[1]

Initially, each thread gets $\leq \lceil n/p \rceil$ consecutive jobs defined by a pair of iterators. A busy thread processes one job after the other. When a thread is done, it looks for more work. If all jobs are already finished, it terminates and so does the algorithms. Otherwise, it tries to steal half of the jobs from a randomly chosen other thread. We implement this without intervention of the victim. Note that this way, the jobs will be dynamically partitioned into a small number of consecutive intervals (this is important for (parallel) cache efficiency), very large jobs will never irrevocably be clustered together, and threads without a processor core assigned to them will lose jobs to stealing (and thus active) threads. To achieve a good compromise between worst-case performance and overhead, the user can optionally devise the algorithm to process the jobs in indivisible chunks of a certain size. Using known techniques (e. g. [10,11]), it can be shown that almost perfect load balancing can be achieved at the cost of only a logarithmic number of work stealing operations per thread.

Find. Function `find_if` finds the first element in a sequence that satisfies a certain predicate. The functions `find`, `adjacent_find`, `mismatch`, `equal`, and `lexicographical_compare` can be immediately reduced to `find_if`. On the first glance, this looks like just another embarrassingly parallel problem. However, a naïve parallelization may not yield any useful speedup. Assume the first matching element is at position m in a sequence of length n. The sequential algorithm needs time $O(m)$. A naïve parallelization that splits the input into p pieces of size n/p needs time $\Omega(n/p) = \Omega(m)$ if $m = n/p - 1$. In practice, we might even see speedup $\ll 1$ if m is so small that the overhead for coordinating threads becomes overwhelming. Hence, our algorithm starts with a *sequential* search for the first m_0 steps. Only then it starts assigning blocks of consecutive sequence positions to the p threads. Consumption of these blocks is dynamically load-balanced using the fetch-and-add primitive. A thread that finds the element, signals this by grabbing all the remaining work. There is still a difficult trade-off here. Assigning small blocks is good because all threads will learn about termination quickly. On the other hand, small blocks are bad because there is some overhead for a fetch-and-add operation. Therefore, our implementation combines both advantages by starting with a block size of $\underline{m}$ and increasing it by a factor g until a value of $\bar{m}$ is reached. The tuning parameters $\underline{m}$ and $\bar{m}$ allow a flexible compromise between fast termination for small m and low overhead for

[1] The method goes back at least to [8], using it for loop scheduling is proposed in [9]. An elegant analysis for shared memory that coined the term work stealing is in [10].

large m. The execution time of our algorithm is independent of n and the term dependent on m is an optimal $O(m/p)$. We need only a single synchronization at the end.

Partial Sum. When computing prefix sums, we synchronize only twice, instead of $\log p$ times, as done by typical textbook algorithms (e. g. [12]). After splitting the sequence into $p+1$ parts, the partial sums of part 0 and the totals sums of parts $1..p-1$ are computed in parallel. After processing these intermediate results in a sequential step, the partial sums of parts $1..p$ are computed. To compensate for the first thread possibly taking longer because of having to write back the results in the first step, the user can specify a dilatation factor d. The total running time then is $O(n/p+p)$, the maximum speedup achievable is $(p+d)/(1+d)$, i. e. $(p+1)/2$ for $d=1$.

Sorting and its Kindred. *Partition.* Given a *pivot predicate* P, we are asked to permute $[S[0],\ldots,S[n-1]]$ such that we have $P(S[i])$ for $i<m$ and $\neg P(S[i])$ for $i \geq m$. This routine is part of the STL and the most important building block for quicksort, selection, etc. We use a parallel algorithm similar to the one by Tsigas and Zhang [7], which has many advantages. Its inner loop is the same as in sequential quicksort, it works in-place, and it is dynamically load-balanced.

The sequential algorithm scans S from both ends until it finds two elements $S[i]$ and $S[j]$ that belong to the "other" side respectively. It swaps $S[i]$ and $S[j]$ and continues scanning. The parallel algorithm works similarly. However, each thread reserves two chunks of a certain size B from each end. It performs the partitioning of those two chunks, until one of them runs empty. If the left chunk runs empty, it reserves a succeeding block using a fetch-and-add primitive. Symmetrically, if the right size runs empty it reserves a preceding block. This process terminates when there are less than B elements left between the left and the right boundary. When all threads have noticed this condition, there is at most one chunk per thread that is partly unprocessed. After calculating various offsets sequentially, each thread swaps its unprocessed part to the "middle" of the sequence. Those remaining elements are treated recursively in this manner, with fewer threads, ending in the sequential call for less than B elements. The running time of this algorithm is bounded by $O(n/p+Bp)$.

m^{th} *Element*[2]. Using the above parallel partitioning algorithm, it is easy to parallelize the well known quickselect algorithm: partition around a pivot chosen as the median of three. If the left side has at least m elements, recurse on the left side. Otherwise recurse on the right side. Switch to a sequential algorithm when the subproblem size becomes smaller than size $2Bp$ where B is the tuning parameter used in `partition`. We get total expected execution time $O(\frac{n}{p}+Bp\log p)$.

Multi-Sequence Partitioning. Given k *sorted* sequences $S_1,\ldots,S_k$ and a *global rank* m, we are asked to find splitting positions $i_1,\ldots,i_k$ such that $i_1+\cdots+i_k=m$ and $\forall j,j' \in 1..k : S_j[i_j-1] \leq S_{j'}[i_{j'}]$. The function `multiseq_partition` is

[2] For consistency with our notation, where n is the input size, we use m for the requested rank, although the STL function is called `nth_element`.

not part of the STL, but useful for many of the subsequent routines based on merging. Our starting point is an asymptotically optimal algorithm by Varman et al. [6] for selecting the element of global rank m in a set of sorted sequences. It is fairly complicated, and to our knowledge, has been implemented before only for the case that the number of sequences is a power of two and all sequences have the same length $|S_j| = 2^k - 1$ for some integer k.

Explicit care has been taken of the case of many equal elements surrounding the requested rank. To allow stable parallelized merging based on this partitioning, the splitter positions may not be in arbitrary positions in the equal subsequence. In fact, there must not be more than one sequence S_j having a splitter "inside" the equal subsequence. All S_i with $i < j$ must have the splitter at the end of it, all S_i with $i > j$ must have the splitter at its beginning. The running time amounts to $O(k \log k \cdot \log \max_j |S_j|)$.

Merging. Given two sorted sequences S_1 and S_2, STL function `merge` produces a sorted sequence T containing the elements from S_1 and S_2. We generalize this functionality to multiple sorted sequences $S_1,\ldots,S_k$. This is an operation known to be very effective for both cache efficient and parallel algorithms related to sorting (e. g., [13,14,15]).

We can reduce parallel multiway merging to sequential multiway merging using calls to multi-sequence partition with global ranks $\{m/p, 2m/p, \ldots, (p-1)m/p, m\}$, and splitting accordingly. Our implementation of sequential multiway merging is an adaptation of the implementation used for cache-efficient priority queues and external sorting in [13,15]: For $k \leq 4$, we use specialized routines that encode the relative ordering of the next elements of each sequence into the program counter. For $k > 4$, a highly tuned tournament tree data structure keeps the next element of each sequence. The total execution time of our algorithm is $O(\frac{m}{p} \log k + k \log k \cdot \log \max_j |S_j|)$.

Sort. Using the infrastructure presented, we can implement two different parallel sorting algorithms:

(Stable) Parallel Multiway Mergesort: Each thread sorts $\leq \lceil n/p \rceil$ elements sequentially. Then, the resulting sorted sequences are merged using parallel multiway merging. Finally, the result is copied back into the input sequence. The algorithm runs in time $O(\frac{n \log n}{p} + p \log p \cdot \log \frac{n}{p})$. Our implementation of parallel multiway merging allows stable merging — it will always take S_i with the minimum i available when it encounters equal elements. Hence, by using a stable algorithm for sequential sorting we get a stable parallel sorting algorithm.

Load-Balanced Quicksort: Using the parallel partitioning algorithm from Section 2, we can obtain a highly scalable parallel variant of quicksort, as described in [7]. Although this algorithm is likely to be somewhat slower than Parallel Multiway Mergesort, it has the advantage to work in-place and to feature dynamic load balancing.

The sequence is partitioned into two parts recursively. After the sequence has been split to p parts, each thread sorts its subsequence sequentially. However,

the length of those subsequences may differ strongly, so normally the overall performance would be poor. To overcome this problem, we implemented the quicksort variant using lock-free double-ended queues to replace the local call stacks. After partitioning a subsequence, the longer part is pushed onto the top end of the local queue, while the shorter part is sorted recursively. When the recursion returns, a subsequence is popped from the top end of the local queue. If there is none available, the thread steals a subsequence from another thread's queue. It pops a block from the bottom end of the victim's queue. Since this part is probably relatively large, the overhead of this operation is compensated for quite well. If the length of the current part is smaller than some threshold, no pushing to the local queue is done any more, so the remaining work is done completely by the owning thread.

The functionality required for the double-ended queue is quite restricted, as is its maximum size. Thus, a circular buffer held in an array and atomic operations like fetch-and-add and compare-and-swap suffice to implement such a data structure, with all operations taking only constant time.

If an attempt to steal a block from another thread is unsuccessful, the thread offers the operating system to yield control using an appropriate call. This is necessary to avoid starvation of threads, in particular if there are less (available) processors than threads. There could still be work available albeit all queues are empty, since all busy threads might be in a high-level partitioning step.

The total running time is $O(\frac{n \log n}{p} + Bp \log p)$, ignoring the load-balancing overhead. Figure 2 shows a possible state of operation.

Random Shuffle. We use a cache-efficient algorithm random permutation algorithm [16] which extends naturally to the parallel setting: In parallel, throw each element into one out of k random bins per thread, pool the corresponding bins, and permute the resulting bins independently in parallel (using the standard algorithm). We use the Mersenne-Twister random number generator [17] which is known to have very good statistical properties even if random numbers are split into their constituent bits, as we do.

3 Software Engineering

Goals. A major design goal of the MCSTL is to provide parallelism with hardly any effort from the user. The interface fully complies to the C++ Standard Template Library in terms of syntax. However, the MCSTL has quite high requirements when it comes to semantics. For example, the operations called in algorithms like `for_each` must be independent and non-interfering with each other. Also, the order of execution cannot be guaranteed any more. The user should call the appropriate algorithms like `accumulate`, if this is undesired. Also, some programs might rely on invariants of algorithms that are not guaranteed by the standard. A common example is `merge`. For the standard merging algorithm, both sequences can overlap without doing any harm. However, this is fatal for the parallel algorithm.

Library Particularities. Many parallel algorithms are published that show excellent scalability results. However, the experiments are usually either limited to a platform, a specific data type as input, and / or assumptions on the input data type. The code is hard to re-use since it is specialized to a specific machine and specific needs.

In contrast to this, a library implementation of a parallel algorithms must be more general. First of all, it need be parameterizable by the data type the input consists of. Secondly, the basic operations are defined by the user, for some algorithms. The data type can carry additional semantics, e. g. the comparison operator might be redefined, or the assignment operator and the copy constructor. The library may not violate any consequences following from this. Also, the running time of the functors may be arbitrarily distributed. For some operations, e. g. prefix sum and accumulation, we assume the execution time to be about the same for each call, while for `for_each`, no such assumption is made.

Implementation. Any parallel library needs a foundation that supports concurrent execution. For our library, this foundation should be both efficient and platform-independent. We have chosen to use OpenMP 2.5 [18]. Additionally, a thin platform-specific layer provides access to a small number of efficiently implemented primitive atomic operations like fetch-and-add and compare-and-swap. Although these primitives are still compiler specific, there is already enough convergence that one can obtain a portable library using this functionality.

Using the MCSTL. Using the MCSTL in a program is extremely simple, just add a few compiler command line options. The library will then use default values for deciding whether to parallelize an algorithm, based on the automatically determined number of cores.

However, if the user wants to customize the calls for maximum performance in any setting, he can do so at little effort. The library allows setting for each algorithm the minimal problem size from which the library should call the parallel algorithm. Also, the number of threads used can be easily specified. The algorithm alternatives and tuning parameters are also accessible to the user. If an algorithm must be executed sequentially by all means, the programmer specifies this by adding `mcstl::sequential_tag()` to the end of the call. The original STL version will then be called without any runtime overhead since the decision is made at compile-time through function overloading.

4 Experimental Results

Testing Procedures. We have tested on four different platforms, namely a Sun T1 (8 cores, 1.0 GHz, 32 threads, 3 MB shared L2 cache), a AMD Opteron 270 (2 cores, 2.0 GHz, 1 MB L2 cache per core), an 2-way Intel Xeon 5140 (Core 2 architecture, 2×2 cores, 2.33 GHz, 2×4 MB shared L2 cache per processor), and a 4-way Opteron 848 (4×1 core, 1.8 GHz, 1 MB L2 cache per processor). For

comparability, all programs were compiled with the same GCC 4.2 snapshot[3], which features a reliable implementation of OpenMP.

In the following subsections, we show how the running time relates to the original sequential algorithm provided by the corresponding STL, expressed as speedup. Unless stated explicitly, the Sun T1 is considered. For testing, the parallel execution of all algorithms was forced, to also show the results for small inputs which would not have executed in parallel, normally.

Input Data. Unless stated otherwise, we used uniform random input data for our experiments. All test were run at least 30 times, the running times were averaged. The input data varies with every run, and is generated immediately before executing the algorithm, so it may (partly) reside in cache. This is a realistic setting which in fact favors the sequential version because it can access the data right away, while the many threads of the parallel implementation have to communicate the data via the slow main memory.

Embarrassingly Parallel Computations. We tested the algorithms for performing embarrassingly parallel computations by computing the Mandelbrot fractal, on the 4-way Opteron. For each pixel, for up to i iterations of a computation with complex numbers. Since the computation is interrupted as soon as the point known to be outside the Mandelbrot set, the computation time for the different pixels differs greatly. Speedup of up to 3.4 is achieved for $i = 10000$ iterations per pixel with the dynamical load balancing, whereas static load balancing only gains a factor of at most 2. This shows the superiority of the dynamic load balancing for jobs with highly varying running time.

Find. The naïve parallel implementation of find performs very badly and is far from achieving good speedup, as shown in Figure 3. For the parameters chosen, $m_0 = 1000, \underline{m} = 10000, \bar{m} = 64000, g = 2$, we report an interesting insight. The growing block variant (gb) performs best for most cases. It is only superseded by the fixed size block (fsb) variant in a narrow segment, where the parameters are just right by chance. When switching to the parallel processing there is speedup < 1, for small inputs. Speedup then goes up steeply, too almost full speddup, as long as there are enough cores available.

Partial Sum. The speedups for `partial_sum` (see Figure 5) are only about half of the number of processor, as predicted, because the parallel algorithm performs almost twice the work than the sequential one.

Partition / m$^{\text{th}}$ Element. `partition` gains linear speedup for up to eight threads, and benefits from multi-threading up to a factor of 15, as shown in Figure 4. However, there is no speedup at all for small input sizes. This is because there is only little computation for integers, so overhead comes in badly.

Sort. With large elements, the multiway mergesort (mwms) achieves speedup up to 20 on the 8-core T1 (see Figure 7). This is quite impressive, the multi-threading is utilized extensively. With as many threads as cores, speedup 7.5

[3] Revision 114849, 2006-06-21.

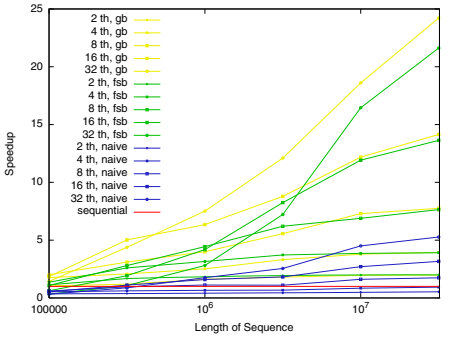

Fig. 3. Finding a 32-bit integer at random position, using different algorithm variants: growing block size (gb), fixed-size blocks (fsb), and naive

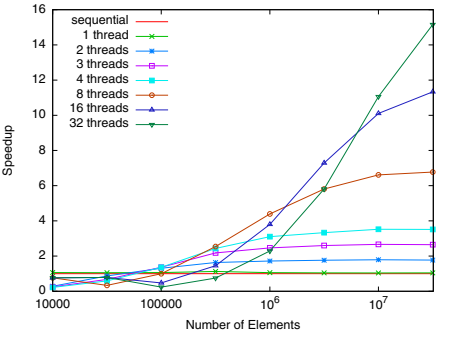

Fig. 4. Partitioning a sequence of 32-bit integers

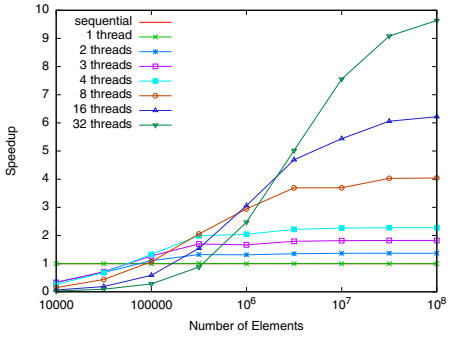

Fig. 5. Computing partial sums of 32-bit integers

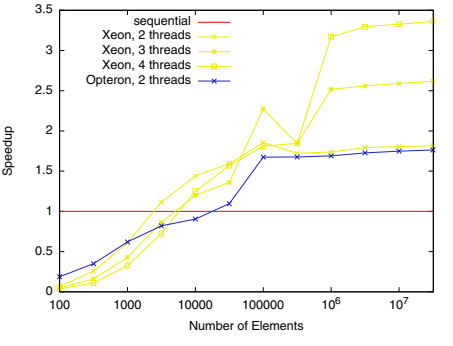

Fig. 6. Sorting 32-bit integers on the Xeon and the Dual-Core-Opteron

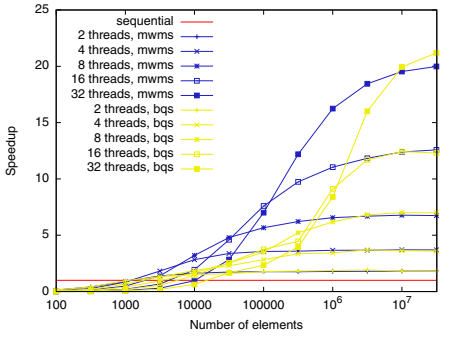

Fig. 7. Sorting pairs of 64-bit integers

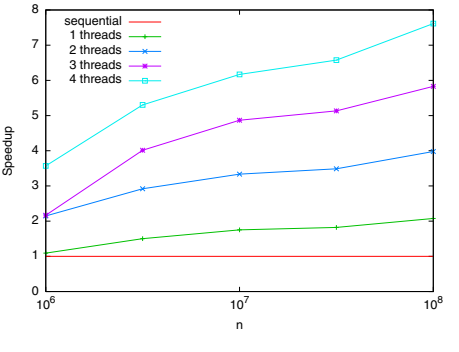

Fig. 8. Random shuffling of 32-bit integers on the 4-way Opteron

is possible. Also, speedup can be achieved with a small number of threads for as few as 3000 elements. Not only performing well in this practical settings, the MCSTL parallel multiway merge sorter does exact splitting and therefore guarantees these good execution times, even for worst-case inputs. Tests have shown that calculating the exact partition only costs negligible time.

The numbers for balanced quicksort (`bqs`) converge to about the same values, but more slowly, even beating mergesort in some cases, culminating in a speedup of 21. The algorithm works in-place and also is dynamically load-balanced.

With the former example, we also want to demonstrate the power efficiency of multi-core processors. For sorting more than 31.6 million integers, the T1 is about as fast as the 4-way Opteron running with 3 threads. However, those three processors consume about 246 W, while the T1 can be run with only 72 W. This yields a three-times better power-efficiency, although both processors are of the same generation.

The results for the Xeon, presented in Figure 6, show the great influence of the caches on the performances. For one processor, the speedups for two threads are excellent, and also scale down well. The two cores work together very well because they share the L2 cache. The break-even point is below 3000 integers. However, additional speedup by incorporating the other two cores is only achieved for an input data size at least as large as the L2 cache size (4 MB). Also, one can clearly see the gap in speedup between the two machines in the region between 1000 and 100000 input elements, where the Dual-Core-Opteron suffers from its separated L2 caches.

Random Shuffle. The performance for random shuffling, shown in Figure 8, profits from the cache-aware implementation that makes the sequential algorithm already twice as fast as the standard one. The speedup continues to scale with the number of threads, for inputs exceeding the cache.

5 Conclusion

We have demonstrated that most algorithms of the STL can be efficiently parallelized on multi-core processors. Simultaneous multithreading has also been shown to have a great potential 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.

Future Work. Implementation of the MCSTL will continue with implementations of worthwhile functions that are not yet parallelized. We have just started to implement containers. We will start with complex operations (e. g. (re)construction, rehashing), advancing to the more fine-grained ones like priority queue updates.

We will add (optional) dynamic load balancing to more functions. In some case, like parallel multiway mergesort or prefix sum, this poses interesting algorithmic problems.

The MCSTL sometimes offers several implementations of functions. Of course, in an easy-to-use library we would like automatic support for selecting an implementation. Some work in this direction has been done in [4]. However, more work is needed because it is not sufficient to *select* an algorithm, we also have to *configure* it to use the right values for tuning parameters such as the value B in `partition`, and, most importantly, the number of threads to be used.

The MCSTL is available freely on our website [19], and can be used by everyone free of charge. We plan to integrate it with the external memory library STXXL [20]. Since in many situations the STXXL is compute bound rather than I/O bound, we expect significantly improved performance for various algorithmic problems on huge data sets.

References

1. Putze, F., Sanders, P., Singler, J.: The multi-core standard template library. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 144–145. ACM Press, New York (2007)
2. Plauger, P.J., Stepanov, A.A., Lee, M., Musser, D.R.: The C++ Standard Template Library. Prentice-Hall, Englewood Cliffs (2000)
3. 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: Dietz, H.G. (ed.) LCPC 2001. LNCS, vol. 2624, pp. 193–208. Springer, Heidelberg (2003), `http://parasol.tamu.edu/groups/rwergergroup/research/stapl/`
4. Thomas, N., Tanase, G., Tkachyshyn, O., Perdue, J., Amato, N.M., Rauchwerger, L.: A framework for adaptive algorithm selection in STAPL. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 277–288. ACM Press, New York (2005)
5. Baertschiger, D.: Multi-processing template library. Master thesis, Université de Genève (in French) (2006), `http://spc.unige.ch/mptl`
6. Varman, P.J., Scheufler, S.D., Iyer, B.R., Ricard, G.R.: Merging Multiple Lists on Hierarchical-Memory Multiprocessors. Journal of Parallel and Distributed Computing 12(2), 171–177 (1991)
7. Tsigas, P., Zhang, Y.: A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000. In: 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing, p. 372 (2003)
8. Finkel, R., Manber, U.: DIB – A distributed implementation of backtracking. ACM Transactions on Programming Languages and Systems 9(2), 235–256 (1987)
9. Sanders, P.: Tree shaped computations as a model for parallel applications. In: ALV'98 Workshop on Application Based Load Balancing (1998)
10. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. Journal of the ACM 46(5), 720–748 (1999)
11. Sanders, P.: Randomized Receiver Initiated Load Balancing Algorithms for Tree Shaped Computations. The Computer Journal 45(5), 561–573 (2002)
12. JáJá, J.: An Introduction to Parallel Algorithms. Addison-Wesley, Reading (1992)
13. Sanders, P.: Fast priority queues for cached memory. ACM Journal of Experimental Algorithmics 5 (2000)
14. Ranade, A., Kothari, S., Udupa, R.: Register Efficient Mergesorting. In: Prasanna, V.K., Vajapeyam, S., Valero, M. (eds.) HiPC 2000. LNCS, vol. 1970, pp. 96–103. Springer, Heidelberg (2000)

15. Dementiev, R., Sanders, P.: Asynchronous parallel disk sorting. In: 15th ACM Symposium on Parallelism in Algorithms and Architectures, pp. 138–148. ACM Press, New York (2003)
16. Sanders, P.: Random permutations on distributed, external and hierarchical memory. Information Processing Letters 67(6), 305–310 (1998)
17. Matsumoto, M., Nishimura, T.: Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation 8, 3–30 (1998)
18. OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 2.5 (May 2005)
19. Singler, J.: The MCSTL website (June 2006), `http://algo2.iti.uni-karlsruhe.de/singler/mcstl/`
20. Dementiev, R., Kettner, L., Sanders, P.: STXXL: Standard Template Library for XXL data sets. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 640–651. Springer, Heidelberg (2005)