2013 IEEE 27th International Symposium on Parallel & Distributed Processing

Malleable Sorting

Patrick Flick, Peter Sanders, Jochen Speck
Department of Informatics
Karlsruhe Institute of Technology
Karlsruhe, Germany
{sanders,speck}@kit.edu

***Abstract*—*Malleable* jobs can adapt to varying degrees of available parallelism. This is an interesting approach to more flexible usage of parallel resources. For example, malleable jobs can be scheduled optimally and efficiently where more restricted forms of parallel jobs are NP-hard to handle. However, little work has been done on how to make fundamental computations malleable. We study how this can be done for sorting. Our algorithm is an adaptive version of Multiway Merge Sort and outperforms a state-of-the art implementation in the multi core STL when the number of available cores fluctuates.**

I. Introduction

Sorting is one of the most important basic algorithms. Many computers spend much of their time sorting some values and a lot of research considers the efficiency of sorting (see [5]).

Sorting is a CPU-time intensive part of many programs. As multi core CPUs are now common on average systems and the number of cores per CPU is increasing, parallel sorting becomes a common subroutine in many applications. A typical library routine for parallel sorting is the Multiway Merge Sort from the MCSTL [11] which is also part of many other libraries. The Multiway Merge Sort is very efficient on a system with no other jobs running, but it is known from the PhD-thesis of Johannes Singler [10] (one of the MCSTL-authors) that it loses efficiency if other jobs are running concurrently.

A malleable task (as defined in [6]) is a task where the number of assigned cores can be changed from outside during the execution. In scheduling theory one often finds papers concerned with the scheduling of malleable tasks see [2] and [1]. Tasks of this type would be a perfect fit for new systems with resource aware scheduling strategies, which will use the available resources more efficiently. For an example of the plan of an adaptive system see [12]. Resource aware scheduling means here that the number of cores, which are assigned to a task, depends also on the current workload of the system. As many systems today usually execute more than one task in parallel, these systems may benefit from the ideas of resource aware scheduling.

Also scheduling will be easier with malleable tasks. If you can choose the number of processors assigned to a job only once, when the job starts, the scheduling problem becomes NP-hard even with preemption allowed [3]. But if we have malleable jobs with a concave speedup function, we can find the optimal schedule in polynomial time [1]. We do not prove that the new malleable sorting fulfills all restrictions for polynomial time optimal scheduling but it seems to be much closer than other sorting algorithms.

Our goal in this work was to show that if we add an internal scheduler to Multiway Merge Sort, which also gets information about the current system load, we can significantly improve the system efficiency in the situation of another job running in parallel. We see this work as a step towards resource aware systems, which will give each job a certain amount of resources, which can be changed over time. With the right kind of jobs, which are able to adapt to changing amounts of resources, these systems can improve efficiency significantly.

In order to add an internal scheduler to the Multiway Merge Sort we had to change it a little, but we tried to use as many parts of the original sorting algorithm as possible. For the rest of the article we call our malleable sorter MALMS and the Multiway Merge Sort from the MCSTL is called STLMS. Additionally we compared MALMS with the parallel sorting algorithm from Intel's TBB[1]. For the rest of the article we call the parallel sorter from TBB just TBB.

The main result of this article is that MALMS is as good as STLMS if there is no other task on the system, but if there is another task running on the system MALMS shows a big advantage. Therefore the internal scheduler and its integration into the Multiway Merge Sort result in a small overhead on an otherwise empty system, but improve performance on a system with other active jobs. TBB is much slower on an empty system, but for a loaded system it is sometimes faster and sometimes slower than MALMS. We also analyzed how much the sorting algorithms slow down other tasks running on the same system.

In Section II we introduce the algorithm and the general structure of MALMS and give a short introduction to the algorithm behind STLMS. Some details of our implementation and the experimental setup is described in Section III. The experimental comparisons under different circumstances are presented in the experimental Section IV.

II. Algorithm

In this section we describe MALMS and STLMS and we especially describe the changes made in STLMS in order to build MALMS.

Each sorting algorithm (MALMS and STLMS) gets its input as an array of n elements to be sorted. The output is an

[1]http://threadingbuildingblocks.org/

1530-2075/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPS.2013.90

array with the same elements but the elements are stored in a nondecreasing sequence.

Both algorithms are quite similar as it was our goal to "add" malleability to the STLMS rather than to implement a whole new algorithm. The plan was to add flexibility and to inherit the speed of the original algorithm. For this reason we give a short description of both algorithms in the next three breaks.

Both algorithms organize their work in work packages, throughout the paper we use k as the number of work packages. For simplicity of presentation we assume that $\frac{n}{k}$ is integral for the rest of the paper. For STLMS the number of work packages is the same as the number of threads. For MALMS the number of work packages is an optimization parameter. More work packages bring better adaptivity and malleability but also more overhead.

Both algorithms have three phases. In the first phase the input array is split into k equal sized packets which are sorted independently using a sequential sorting algorithm. In the second phase $k-1$ splitter keys are computed. Each splitter splits all of the k sorted sequences into an upper and an lower part. Additionally the total number of elements in all sequences, which are between the r-th and the $r+1$-th splitter is $\frac{n}{k}$. There are also $\frac{n}{k}$ elements below the first and above the $k-1$-st splitter. In the third phase all elements between the r-th and the $r+1$-th splitter are merged into a sorted sequence. The concatenation of these sequences is the final result.

Both algorithms split the work in each of these phases into k work packages. In the first phase the sorting of each of the k packets of input elements is a work package. In the second phase, finding one of the $k-1$ splitters makes up a work package. A work package of the third phase consist of the merging of all elements between the r-th and the $r+1$-th splitter.

Now we describe the management of work packages in MALMS. For each phase a single thread prepares all work packages and puts them into a queue. After this is done, each worker thread takes one package from the queue and works on it. When a worker thread has finished its work package it takes a new one from the queue or if the queue is empty it waits until all worker threads have finished their packages. When all workers have finished their work packages the next phase starts.

The malleability is organized by the malleable scheduler. The malleable scheduler manages the queue of work packages, the threads and the signaling. The malleable scheduler has one thread per core on the system. Each thread is assigned to one core. All worker threads visit the malleable scheduler in the following manner: First the thread checks if it is blocked. If it is blocked then it goes to sleep. If the thread is not blocked it fetches a work package and starts working on it. If MALMS gets the signal to use a currently not used core p, the malleable scheduler wakes up the thread assigned to p. This thread immediately takes a work package from the queue and starts working on it. If MALMS is ordered to release a currently used core p, it blocks the thread assigned to p when it tries to take a new work package. If the work packages are small enough, we have a malleable job whose number of used cores can be controlled from the outside.

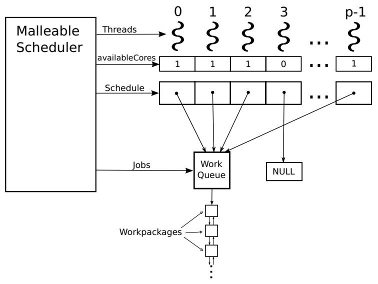


Fig. 1. Overview of the design and the principle of operation of the malleable scheduler.

A. Basic Algorithms used in the Three Phases

In this section we describe in short the algorithms used by MALMS in the three phases.

In the work packages of the first phase, the sequential sort from the STL is used. `std::sort(...);` calls introspective sort [7] which has a good worst case behavior but is not stable. The complexity of each work package in this phase is in $\Theta(\frac{n}{k}\log\frac{n}{k})$. For p parallel threads and p divides k the time complexity of the first phase is in $\Theta(\frac{n}{p}\log\frac{n}{k})$.

The splitting in the second phase is very important for this algorithm because there are more splitters, which have to be computed, than in STLMS. Each work package consists of the computation of the r-th splitter and its position in all k sorted sequences of the first phase ($r \in \{1,\ldots,k-1\}$). The total number of elements below the r-th splitter has to be $\frac{r\cdot n}{k}$. The 0-th splitter is defined to be directly below the smallest element and the k-th splitter is defined to be directly above the largest element of all work packages from the first phase.

The splitting algorithm maintains three arrays of size k. In each array it keeps one position in each sequence. Now we explain how the r-th splitter is computed. The number of elements below the splitter must be $s=\frac{r\cdot n}{k}$. The array `lower` is initialized with the positions of the 0-th splitter, the array `upper` is initialized with the positions of the k-th splitter, and the array `current` is initialized empty. For the splitting the following step is repeatedly performed:

1) For all nonempty sequences get the median and attach the number of elements in the sequence as a weight to it.
2) Compute the weighted median w of the medians of the sequences.
3) For each sequence find the splitting point where w would fit into the sorted sequence by binary search and store it in `current`.
4) Count the elements between `lower` and `current` in c.
5) If c is larger than s then `upper` is replaced with `current` else `lower` is replaced with `current` and $s=s-c$.

6) Repeat the step if there are more than $k+16$ elements remaining between `lower` and `upper` together in all sequences.

The remaining elements are then sorted and the r-th splitter and its positions are computed. The complexity of each work package in this phase is in $\Theta(k \log^2 \frac{n}{k})$. For p parallel threads the time complexity of the second phase is in $\Theta(\frac{k^2}{p} \log^2 \frac{n}{k})$. The unusual value of $k+16$ comes from the fact that we first used the splitting algorithm from Frederickson and Johnson [4] and kept the criterion to end the loop when we switched to our own splitting algorithm.

A work package of the third phase consists of merging k sorted sequences into one sorted sequence. We use the loser tree [5] implementation which is also part of the STLMS. The complexity of each work package in this phase is in $\Theta(\frac{n}{k} \log k)$. For p parallel threads and p divides k, the time complexity of the third phase is in $\Theta(\frac{n}{p} \log k)$.

III. Realization and Testing Environment

A. Machines for Experiments

The Machine used for experimental evaluation is a system composed of two quad-core Intel Xeon 5345 (Woodcrest) processors. Both sockets access the memory through a shared memory controller. Thus this machine has a uniform memory architecture (UMA), which simplifies the experimental evaluation because we do not have to consider NUMA effects. Table I shows the technical details of the machine.

The machine is running a Linux Kernel version 2.6.32-45-generic x86_64[2] and a GCC version 4.4.3. The TBB version is 4.1 Update 1. The Linux scheduler of the system has the standard parameters of the installation. For STLMS it is solely responsible for the distribution of work among the cores.

Name	Xeon
CPU Model	Intel Xeon 5345
#Sockets	2
#Cores	8 (4 per Socket)
#Threads	8
Architecture	Woodcrest
Frequency	2.33 GHz
L1 Cache L2 Cache L3 Cache	2x 4x 16 KiB 2x 4x 512 KiB 2x 2 MiB (shared)
Memory	16 GiB
Memory architecture	UMA
Kernel version	2.6.32-45-generic x86_64
GCC version	4.4.3

TABLE I
Technical data of the machine used for the experiments.

B. Thread Pinning

The malleable scheduler uses pinned threads. Each thread is pinned to a certain CPU core, thus the Linux scheduler cannot schedule any thread on any other CPU core than the one it is pinned to.

[2]Different kernel versions seem to lead to slightly different results.

The threads are pinned by setting their CPU affinity mask, which determines the set of CPU cores on which the thread is eligible to run. The CPU affinity mask is set via the Linux function `sched_setaffinity()` [8], in order that each thread is only eligible to run on exactly one CPU core, thus pinning them to that CPU core.

C. Signaling System

The malleable scheduler manages the signals over a dedicated signal handler thread. We use Linux real-time signals as defined in the man pages of `signal.h` [9]. We use one signal for adding a core and another signal for releasing a core. The core number for which the action should be taken is sent as the accompanying value of the signal.

D. Method of measurement

For the experiments each tests was run 100 times. In the experiment with the unloaded machines the running times had a small variance so we present the average values. As the variances in the experiment with dynamically loaded cores were much bigger, we present boxplots there. The input data for the experiments consists of random uniformly distributed 32-bit integer elements. The input sizes are powers of 10.

The running times are measured using the POSIX compatible call:
`clock_gettime(CLOCK_REALTIME,...)`, which is independent from the position (core/socket) of the calling thread and enables timing with nanosecond precision as reported by `clock_getres(CLOCK_REALTIME, ...)`.

For each test a new process is created, which reads a randomly generated input file, and measures the running time for one of the algorithms. The measured time includes initialization and starting of threads for the malleable scheduler for MALMS. This way it includes the overhead from starting the malleable scheduler as well as the initialization of the OpenMP thread pool, which creates the threads for the MCSTL inside the call to the MCSTL sort routine. This is more than fair, because the MCSTL sort routine needs by far less time to sort 100 elements with 8 threads, than the initialization of the malleable scheduler in MALMS needs. Also all overheads from TBB were included in the measurements.

E. Loadtask

The idea of a malleable sorter fits best into a system where the availability of resources changes over time.

For the main experiment we used a task which was running at the same time as MALMS, TBB and STLMS. The basic unit of work of the Loadtask is going over a basic array of 1000 integers and performing three integer operations on each. This basic unit of work is called a loop. These loops are counted to have a measurement for the amount of work done in the Loadtask. To have a significant cache footprint (like normal jobs) we have 10000 different basic arrays per core which are visited repeatedly.

When Loadtask loads a core it sends a signal (if MALMS is running) and starts a thread pinned on this core which starts to execute loops. When Loadtask unloads a core it sends a signal (if MALMS is running) and stops the thread pinned to this core. The running of Loadtask depends on time slots where the length of the time slots is part of the experiment. All load and unload operations are done at the beginning of a time slot. Which cores are loaded or unloaded is controlled by a deterministic pattern, which is repeated after some time slots, until the end of the sorting task. All patterns here consist of four time slots.

The sorting algorithms to be tested are started at the same time when Loadtask first starts blocking cores. This is controlled via Linux real-time signals. With the same system we make sure that Loadtask stops immediately after the sorting algorithm has finished. Loadtask sends signals to the sorting algorithms when it starts to use a core and when it releases a core.

IV. Experiments

A. Comparison with no system load

The first experiment compares the algorithms in a system with no further active processes. Table II shows the results for sorting 32-bit uniformly distributed integers. The first row shows the running times of the sequential GCC std::sort implementation, which are used for comparison and for evaluating the absolute speedups. In order to have a better comparison between different input sizes the running time is given as t/n where t is the total running time in nanoseconds and n the input size. For STLMS, TBB and MALMS with various values for k the running time and the absolute speedup is shown.

The absolute speedup values for STLMS, TBB and MALMS show that for input sizes up to 10^4 32-bit integer elements, the sequential GCC std::sort performs at least equally well and parallelization overhead dominates the running time for the parallel algorithms. Hence parallelization of sorting algorithms does not make sense with less than 10^5 elements.

On an otherwise empty machine and with growing input sizes starting with 10^6 elements TBB can not compete with MALMS nor with STLMS. For an input size of 10^6 STLMS is 59.5% faster than TBB and even MALMS with $k = 100$ is 33.1% faster than TBB for this input size. For the input size of 10^7 the corresponding values are 33.3% and 27.5% which TBB is slower. So for an empty machine TBB is the much slower algorithm.

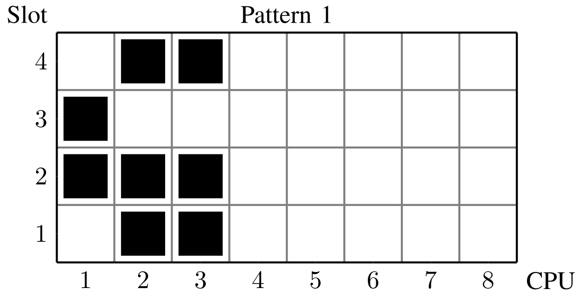
For most of the relevant input sizes, MALMS with small values of k performs as good as STLMS. For larger k the overhead of MALMS is increasing. However, for larger inputs, the overhead for MALMS becomes more and more irrelevant. The more work packages (k) are used for MALMS the more overhead comes into play due to more work packages in the queue, and thus more synchronization overhead within the queue. Also the computational effort for the splitting algorithm grows quadratically in k. In particular, the running time of the splitting phase is in $\Omega(k^2/p)$ because k^2 splitting elements are calculated. The effect of growing k can be seen in Table II for different values of k.

A good choice for k is vital to the performance of MALMS. While a big k creates a huge overhead for the splitting phase, a k that is chosen too small has an impact on the adaptability of the algorithm. An optimal choice for k depends upon the input size and type, the machine and other running processes and their activity profile.

B. Comparison with dynamically loaded cores

As the measurements in this experiment have a bigger variance than the measurements in the unloaded case we present boxplots[3].

In order to test if the advantage of MALMS comes from using the load information or from the smaller work packages, we performed the tests also for MALMS.noinfo which is just MALMS where the receiving of signals from Loadtask is disabled. All tests in this section were run with $k = 100$ work packages.



The black boxes show the times and the cores where Loadtask is running.

In this experiment we have one other task (Loadtask) running on our system. It changes the used cores and the number of used cores regularly over time. We begin with pattern 1, which models the behavior of a small job, which uses some cores heavily and others not at all. The time slots when a core is used are marked with black boxes.

The runtime of the sorting algorithm is not the only interesting measurement here. It is also interesting how much the work of the other task is hindered by the sorting algorithm. Hence we also measured the number of loops Loadtask was able to perform while running in parallel to the tested sorting algorithm. The number of loops is a measure for the work performed by Loadtask. Due to the different running times of the different sorting algorithms we only give the number of loops divided by the running time of the sorting algorithms.

For $n = 10^6$ integers to sort, 2ms time slots and $k = 100$ work packages and Loadtask running pattern 1 we get a clear advantage of MALMS over STLMS: On average STLMS needs 127% more time than MALMS, and the average speed of Loadtask running in parallel to the STLMS reaches only 70.4% of the speed compared to the case when it runs in parallel to MALMS. MALMS.noinfo slows down Loadtask about the same as STLMS but can complete the sorting faster

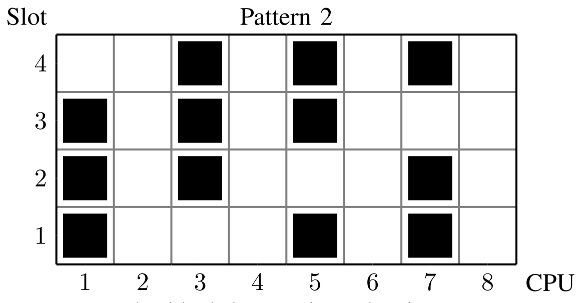
[3]Big line in the middle of the box: Median, Size of box: Two middle quartiles. The whiskers reach to the most extreme value which is not farther than 1.5 times the box length away from the box. All values farther away are marked as outliers.

Input Size	10^3	10^4	10^5	10^6	10^7	10^8
STDSORT	63.4	66.5	79.9	94.9	111	126
TBB	139	152	37.7	31.0	28.1	25.1
TBB SU	0.0457	0.439	2.12	3.06	3.93	5.02
STLMS	481	67.2	22.0	19.5	21.1	23.6
STLMS SU	0.132	0.989	3.64	4.87	5.24	5.35
MALMS $k = 8$	1020	112	26.5	19.1	20.4	22.6
MALMS $k = 8$ SU	0.0620	0.596	3.02	4.98	5.44	5.57
MALMS $k = 24$	1560	147	28.7	20.4	20.8	23.4
MALMS $k = 24$ SU	0.0406	0.452	2.79	4.64	5.32	5.39
MALMS $k = 48$	2250	211	33.2	20.9	21.1	23.5
MALMS $k = 48$ SU	0.0282	0.314	2.41	4.54	5.23	5.36
MALMS $k = 100$	3870	375	46.6	23.3	22.1	24.2
MALMS $k = 100$ SU	0.0164	0.178	1.71	4.07	5.02	5.20
MALMS $k = 200$	6850	731	88.3	29.8	23.2	23.5
MALMS $k = 200$ SU	0.0093	0.0910	0.905	3.19	4.76	5.35
MALMS $k = 400$	13600	1760	237	52.0	28.8	25.4
MALMS $k = 400$ SU	0.0047	0.0378	0.337	1.82	3.84	4.96

TABLE II
RUNNING TIME FOR STDSORT, STLMS, TBB AND MALMS FOR SORTING 32-BIT UNIFORMLY DISTRIBUTED INTEGERS. THE ABSOLUTE SPEEDUP (SU) IS CALCULATED FOR STLMS AND MALMS USING STDSORT AS SEQUENTIAL ALGORITHM. THE RUNNING TIME IS GIVEN AS t/n WHERE t IS THE TOTAL RUNNING TIME IN NANOSECONDS.

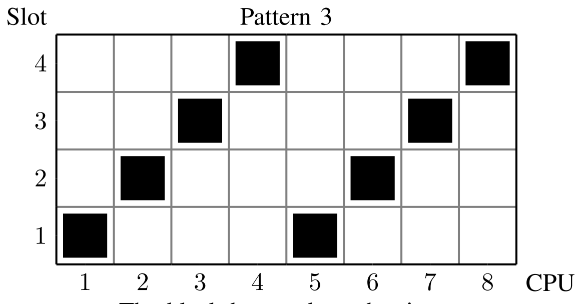
(98% faster). Here TBB is only 3.7% slower than MALMS and Loadtask running parallel to TBB is only 2% slower compared to running parallel to MALMS. The box plots of this experiment are given in Figure 2. We also tested with $n = 10^7$ integers to sort, 6ms time slots and $k = 100$ work packages and Loadtask running pattern 1. This produced similar results. On average STLMS needs 39.5% more time than MALMS, and the average speed of Loadtask running in parallel to the STLMS reaches only 89.5% of the speed compared to Loadtask running in parallel to MALMS. TBB is 7.7% slower than MALMS and Loadtask running in parallel to TBB is only 0.8% slower on average (see Figure 3). In general the advantage of MALMS over STLMS is reduced for larger sorting workloads. Also MALMS has no big advantage compared to TBB in such a dynamic workload.

For other patterns as pattern 2, which also work much on some cores and do not use others, we get similar results for the advantage of MALMS over STLMS. But for such a regular and widespread usage of the machine through other tasks the performance of TBB and MALMS is very similar. We tested with $n = 10^7$ integers to sort, 6ms time slots and $k = 100$ work packages and Loadtask running pattern 2 (see Figure 4). On average STLMS needs 17.5% more time than MALMS, and the average speed of Loadtask running in parallel to the STLMS reaches only 89.6% of the speed when it runs in parallel to MALMS. As this pattern contains more work for Loadtask the values for the normalized work are higher than for pattern 1. When we compare TBB to MALMS we get that TBB is 3.4% faster and Loadtask runs with the same speed in parallel of both (only 0.2% slower running in parallel to TBB). We also tested with $n = 10^6$ integers to sort, 2ms time slots and $k = 100$ work packages and Loadtask running pattern 2. STLMS used 193% more time than MALMS on average, and the average speed of Loadtask running in parallel to STLMS was reduced to 76.9% compared to Loadtask running in parallel to MALMS. TBB was a little bit (2.3%) faster in this case but Loadtask running in parallel to TBB was 11% slower compared to running in parallel to MALMS (see Figure 5).



The black boxes show the times
and the cores where Loadtask is running.

Unfortunately MALMS is not better than STLMS for all patterns. If we use a pattern like pattern 3 where the work of Loadtask is evenly distributed among cores we get different results. Interestingly the result of the comparison to TBB depends heavily on the size of the input.



The black boxes show the times
and the cores where Loadtask is running.

As we can see from Figure 6 and Figure 7 the variance of MALMS is now bigger (at least for the $n = 10^6$ instance). In the case of $n = 10^6$ and 2ms time slots STLMS was on

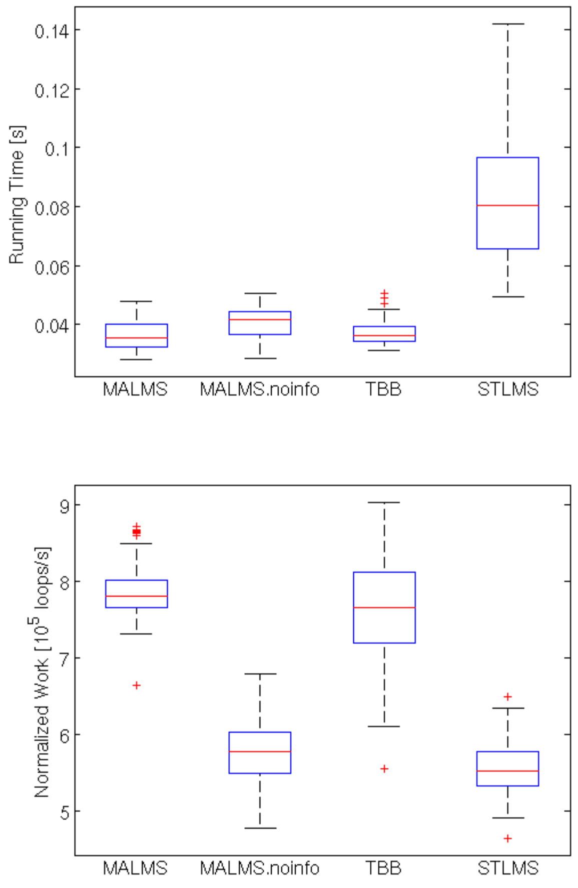


Fig. 2. Results with parallel running Loadtask (pattern 1) for sorting $n = 10^6$ integers, 2ms time slots and $k = 100$ work packages. The second picture shows the performance of Loadtask.

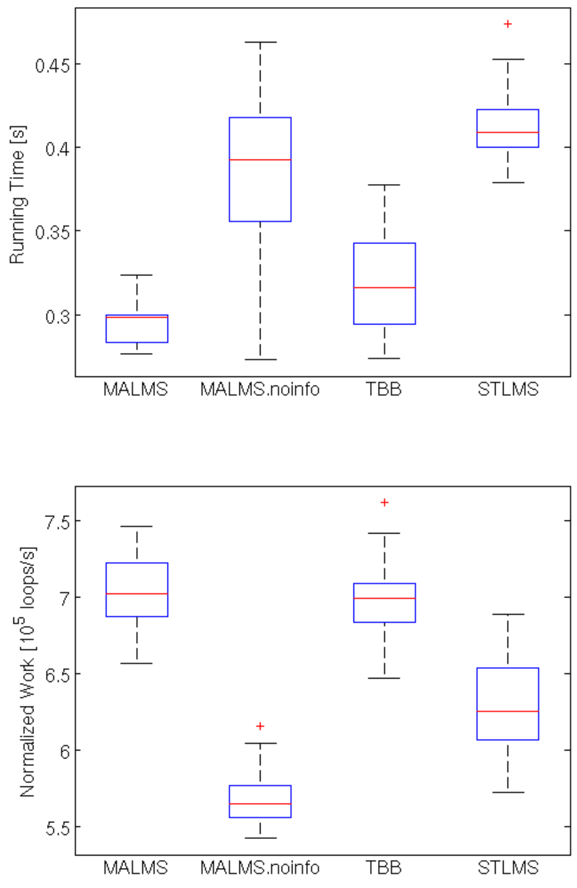


Fig. 3. Results with parallel running Loadtask (pattern 1) for sorting $n = 10^7$ integers, 6ms time slots and $k = 100$ work packages. The second picture shows the performance of Loadtask.

average 4.9% faster compared to MALMS and in case of $n = 10^7$ and 6ms time slots STLMS is 4.3% faster than MALMS on average. The comparison with TBB leads to less uniform results. In the case of $n = 10^6$ and 2ms time slots TBB was on average 17.3% faster compared to MALMS but in case of $n = 10^7$ and 6ms time slots MALMS is 17.6% faster than TBB on average. For the performance of Loadtask we get the surprising result that for both cases STLMS which does not adapt to Loadtask leads to a slightly higher performance of Loadtask compared to MALMS, in case of $n = 10^6$ and 2ms time slots Loadtask is 2.7% faster in case of $n = 10^7$ and 6ms time slots Loadtask is 1.5% faster. For TBB compared to MALMS it is the other way round. If Loadtask is running in parallel to MALMS it is faster than when running in parallel to TBB, in case of $n = 10^6$ and 2ms time slots Loadtask is 14.5% faster in case of $n = 10^7$ and 6ms time slots Loadtask is 2.8% faster.

A clear result of these 6 experiments with Loadtask running in parallel to the sorting algorithm is that STLMS is clearly slower than MALMS and TBB in case of jobs with unbalanced utilization of the different cores. In this case it also hinders Loadtask much more than TBB or MALMS. The running time comparison between TBB and MALMS leads to less clear results. In 5 of 6 cases their boxes in the boxplot overlap which means that at least $1/4$-th of the runs of both algorithms are faster than $1/4$-th of the runs of the other algorithm (in most cases the results of both algorithms are even more similar). Given the variance of the experiments it seems to be difficult to say which algorithm is better in these 5 cases. In case of pattern 3 $n = 10^7$ and 6ms time slots there seems to be a real advantage of MALMS over TBB.

For the performance of Loadtask we get a similar picture for the three algorithms like for their own speed. In case of Loadtask utilizing the different cores in an unbalanced manner, STLMS clearly hinders Loadtask more than TBB or MALMS. In 3 of 6 cases the Loadtask performance is about the same when run in parallel to MALMS or TBB but in the other 3 cases Loadtask performs better when it runs in parallel to

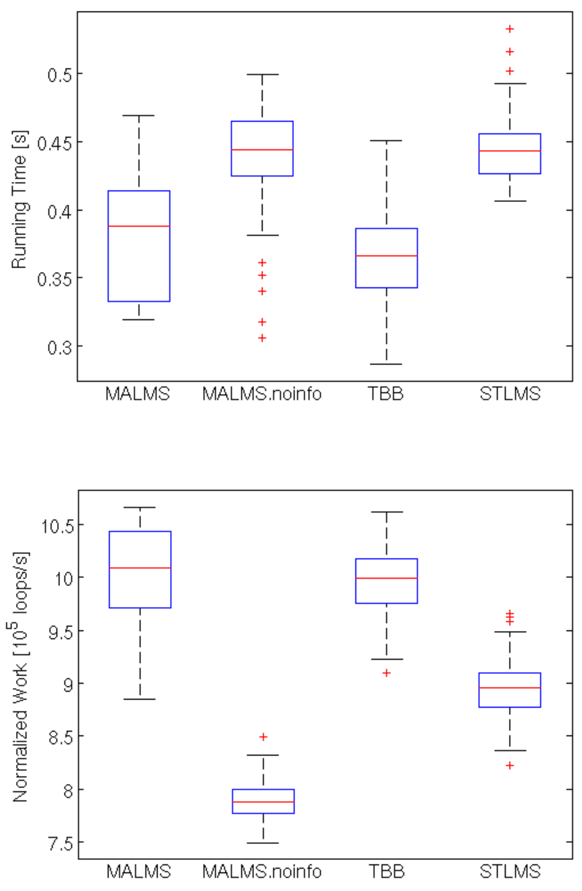


Fig. 4. Results with parallel running Loadtask (pattern 2) for sorting $n = 10^7$ integers, 6ms time slots and $k = 100$ work packages. The second picture shows the performance of Loadtask.

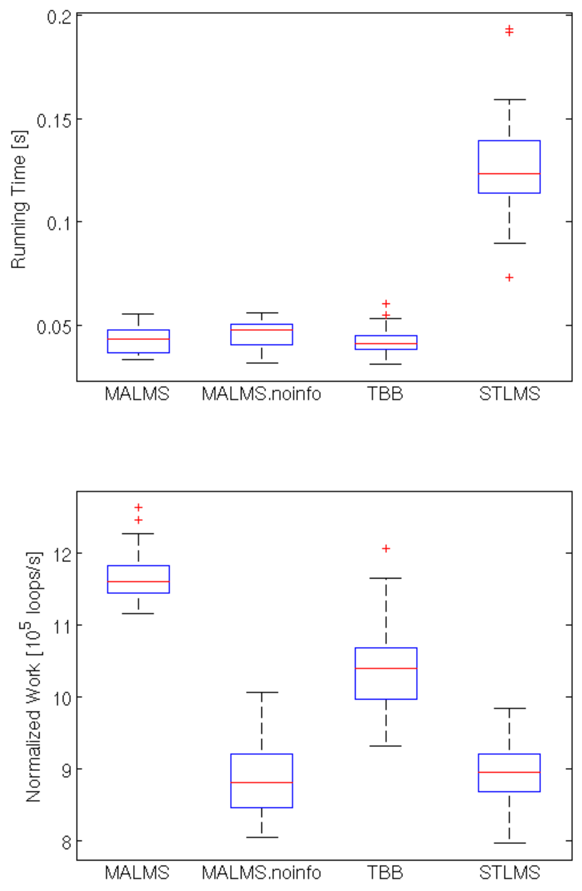


Fig. 5. Results with parallel running Loadtask (pattern 2) for sorting $n = 10^6$ integers, 2ms time slots and $k = 100$ work packages. The second picture shows the performance of Loadtask.

MALMS instead of TBB.

As patterns like pattern 3 seem to be less likely than patterns like pattern 1 or 2 we get that the comparison with dynamically loaded cores leads to the result that MALMS and TBB are a better choice than STLMS in such a scenario. The comparison between MALMS and TBB leads to no clear result in this scenario although MALMS seems to have an advantage in some cases.

In general load patterns are quite important for the result of the comparison between the different algorithms. We selected one where all the work of Loadtask is run on a small number of cores on one CPU, one where the work is more distributed but also different cores are used in a different intensity and one pattern where all cores are used in the same way. As the number of possible patterns is quite high we restricted ourselves to these three patterns in order to be able to interpret the results.

As it is one of the main goals of this work to change STLMS in order to make it faster in the presence of other jobs, it is an interesting question whether this improvement comes from the smaller work packages or from the usage of load information. In 3 of 6 cases the speed of MALMS and MALMS.noinfo is about the same whereas in the other 3 cases MALMS is clearly faster than MALMS.noinfo. Hence the usage of load information seems to be beneficial at least in some cases. The performance of Loadtask profits in all but one case from the usage of load information. Only in case of pattern 3, $n = 10^7$ and 6ms time slots, the performance of Loadtask is similar for MALMS and MALMS.noinfo. In all the other cases MALMS has a clear big advantage in the performance of Loadtask compared to MALMS.noinfo. Thus we can conclude that a big part of the improvements of MALMS over STLMS comes from the usage of load information in MALMS.

V. Conclusion

We were able to build a malleable sorter (MALMS) which is a connection between the theory of malleable tasks and the solution of a practical problem. Because of the performance

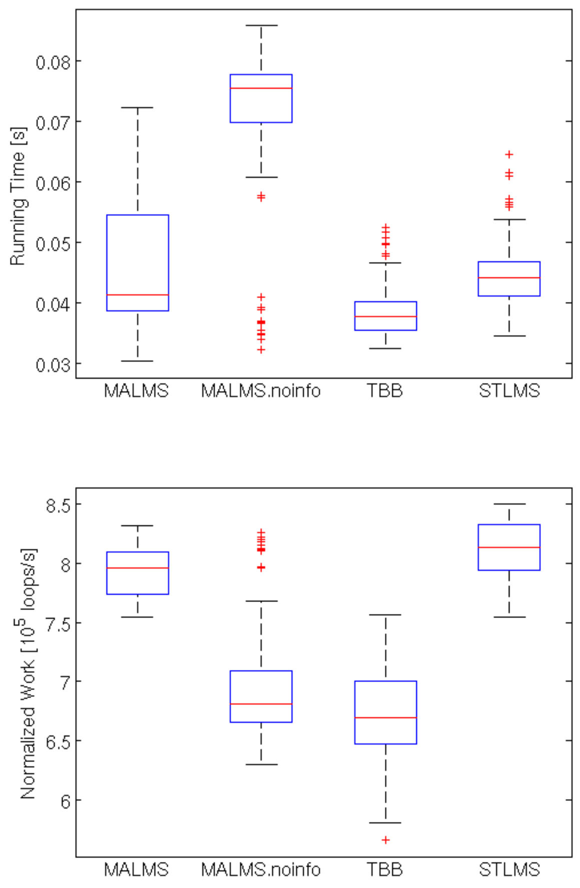


Fig. 6. Results with parallel running Loadtask (pattern 3) for sorting $n = 10^6$ integers, 2ms time slots and $k = 100$ work packages. The second picture shows the performance of Loadtask.

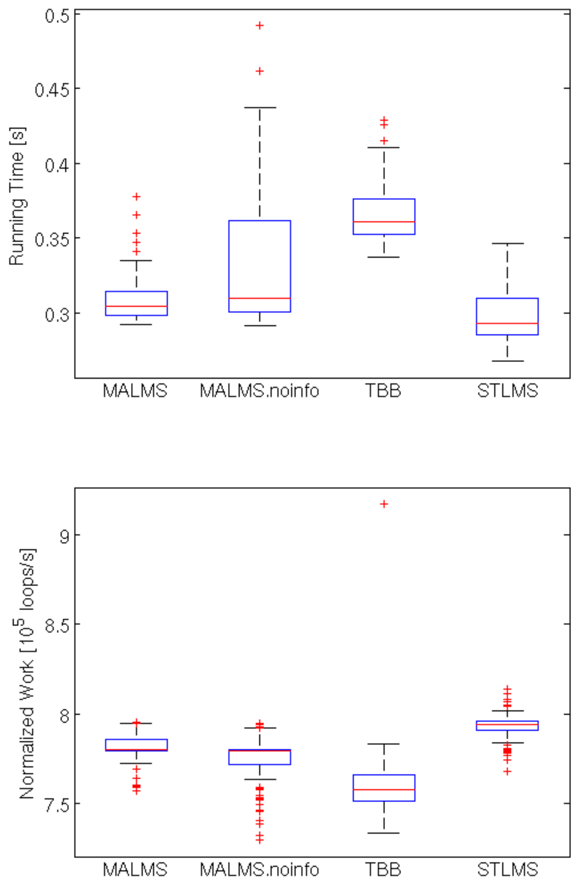


Fig. 7. Results with parallel running Loadtask (pattern 3) for sorting $n = 10^7$ integers, 6ms time slots and $k = 100$ work packages. The second picture shows the performance of Loadtask.

of our new sorter, this is much more than an example for a malleable job. The comparison with the Multiway Merge Sort from the STL shows that our sorter provided with the load information is much faster if there is another job running in parallel to the sorter on the same system (at least for usual load patterns like 1 and 2). Although our sorter has more overhead than STLMS, it is equally fast on an otherwise empty system. This makes MALMS an interesting sorting algorithm, which is sometimes much faster than STLMS and has otherwise a comparable speed.

We also compare MALMS to the parallel sorting algorithm included in Intel's TBB, but this comparison is not the main goal of this work. The result of this comparison is that TBB has a comparable speed in the loaded case and is much slower in the unloaded case compared to MALMS.

We also showed that the information MALMS gets about other running jobs has a relevant share in its performance. Additionally the retreat from cores used by other jobs makes these jobs faster. Altogether we could show that using system load information inside an application can increase system efficiency. Also we could show that a malleable job can have an advantage over a normal parallel job and hence looking at malleable jobs is not only interesting from the perspective of scheduling.

For further research it would be interesting to find other widely used algorithms that can be made malleable with similar gains.

It will also be interesting to look into details which were not considered in this work. One example are different performance counters as possible reasons for the results of this work (cache hits, context switches or waiting times inside the algorithms). Also the parameters of the Linux scheduler or the kernel version may have some influence on the results.

Another interesting question is, if the usage of load information in a sorting algorithm similar to TBB can improve its performance (or the performance of Loadtask) when running in parallel to Loadtask.

It might also be interesting to build a moldable (number of cores used by the algorithm is fixed at starting time of the algorithm) version of STLMS which is also more adaptive than the original STLMS.

Acknowledgment

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

References

[1] Jacek Blazewicz, Mikhail Y. Kovalyov, Maciej Machowiak, Denis Trystram, and Jan Weglarz. Preemptable malleable task scheduling problem. *IEEE Transactions on Computers*, 55:486–490, 2006.

[2] Jacek Blazewicz, Maciej Machowiak, Jan Weglarz, Mikhail Y. Kovalyov, and Denis Trystram. Scheduling malleable tasks on parallel processors to minimize the makespan: Models and algorithms for planning and scheduling problems. *Annals of Operations Research*, 129:65–80(16), July 2004.

[3] Jianzhong Du and Joseph Y.-T. Leung. Complexity of scheduling parallel task systems. *SIAM J. Discrete Math.*, 2(4):473–487, 1989.

[4] Greg N. Frederickson and Donald B. Johnson. The complexity of selection and ranking in x + y and matrices with sorted columns. *Journal of Computer and System Sciences*, 24(2):197 – 208, 1982.

[5] D. E. Knuth. *The Art of Computer Programming—Sorting and Searching*, volume 3. Addison Wesley, 2nd edition, 1998.

[6] J. Y-T. Leung, editor. *Handbook of Scheduling*. CRC, 2004.

[7] David R. Musser. Introspective sorting and selection algorithms. *Softw. Pract. Exper.*, 27:983–993, August 1997.

[8] Linux Man Pages. Man page to sched_setaffinity.

[9] Linux Man Pages. Man page to signal.h.

[10] Johannes Singler. *Algorithm Libraries for Multi-Core Processors*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2010.

[11] Johannes Singler, Peter Sanders, and Felix Putze. MCSTL: The multi-core standard template library. In Anne-Marie Kermarrec, Luc Boug, and Thierry Priol, editors, *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 682–694. Springer Berlin / Heidelberg, 2007.

[12] Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting. Invasive computing: An overview. In Michael Hübner and Jürgen Becker, editors, *Multiprocessor System-on-Chip – Hardware Design and Tool Integration*, pages 241–268. Springer, Berlin, Heidelberg, 2011.