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 [4]).

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 looses efficiency if other jobs are running concurrently.

A malleable task (as defined in [5]) 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 amount 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 did 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 as possible of the original sorting algorithm. For the rest of the article we will call our malleable sorter MALMS and the Multiway Merge Sort from the MCSTL will be called STLMS.

The main result of this article is that MALMS is nearly 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. So the internal scheduler and the necessary changes to the Multiway Merge Sort to add the internal scheduler produce a small overhead on an otherwise empty system but help a lot if other jobs are present.

In Section II we will 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 will be described in Section III. The experimental comparisons under different circumstances are presented in the experimental Section IV.

II. Algorithm

Each sorting algorithm considered in this work (MALMS and STLMS) gets its input as an array of n elements to be sorted. The output is an 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 packets throughout the paper we use k as the number of work packets. For simplicity of presentation we assume that $\frac{n}{k}$ is integer for the rest of the paper. For STLMS the number of work

packets is the same as the number of threads. For MALMS the number of work packets is an optimization parameter. More work packets bring better adaptivity and malleability but also more overhead. We present an experimental evaluation of the overhead in section IV-B.

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 $k-1$ splitters 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 packet 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 packet 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 amount of used cores can be controlled from the outside.

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 packets of the first phase the sequential sort from the STL is used. `std::sort(...);` calls introspective sort [6] 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 $\Theta(\frac{n}{p}\log\frac{n}{k})$.

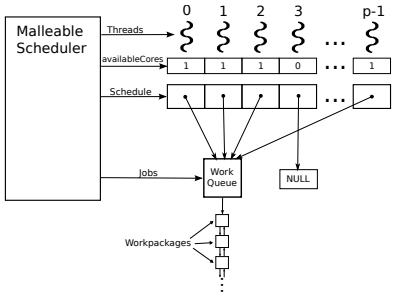


Fig. 1. Overview of the design and the principle of operation of the malleable scheduler.

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 $\Theta(\frac{k^2}{p}\log^2\frac{n}{k})$.

A work package of the third phase consists of merging k sorted sequences into one sorted sequence. We use the loser tree [4] 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.

Both machines are running a Linux Kernel version 2.6.32-38-generic x86_64 and a GCC version 4.4.3. 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	2x 4x 16 KiB
L2 Cache	2x 4x 512 KiB
L3 Cache	2x 2 MiB (shared)
Memory	16 GiB
Memory architecture	UMA
Kernel version	2.6.32-38-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.

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()` [7], 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 realtime signals as defined in the man pages of `signal.h` [8]. 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 50 times. The running times were aggregated using a 10% trimmed mean to eliminate irregularities resulting from background work by the operating system. 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.

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.

E. CPU hotplugging

Some experiments use CPU hotplugging to disable single CPU cores for use by the operating system and its scheduler. As the hotplugging was very slow (deactivate a core with STLMS running 0.7s, for MALMS running 0.4s) and the time to switch on or off a core was even depending on which sorting algorithm was running, we only used the hotplugging for experiments where the amount of processors used for sorting was constant for the whole runtime.

CPU hotplugging is a feature build into the Linux kernel [9] which enables dynamically disabling CPU cores at run-time of the operating system. To disable a CPU core a "0" (to re-enable a "1") is written into the virtual file `/sys/devices/system/cpu/cpu`X`/online`, where X is the ID of the core.

F. 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 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 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.

The length of the time slots can be controlled from outside. 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. For STLMS and MALMS with various values for k the running time and the absolute speedup is shown.

The absolute speedup values for STLMS 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 all following experiments are conducted using at least 10^5 elements.

Throughout all input sizes, the STLMS performs better or as good as MALMS due to more overhead by the malleable scheduler within MALMS. However, starting at 10^6 elements, the overhead for MALMS is small. The more workpackages (k) are used for MALMS the more overhead comes into play due to more workpackages in the queue, and thus more synchronization overhead within the queue.

B. Overhead for malleability depending on k

The running time of the algorithm depends heavily on the parameter k, i.e. the number of workpackages. In particular, the running time of the splitting phase is in $\Omega(k^2/p)$ because k^2 splitting elements are calculated. An experiment in which k is steadily increased from 8 up to 800 shows the different running times of the three phases of the algorithm (see Figure 2).

The running time of the run-formation phase decreases slightly but remains almost constant. The running time for the merge step increases slightly, which matches its logarithmic growth in k. While the running time for the splitting phase makes only a small contribution to the total running time of 0.78% for $k = 8$ and still a minor 5.6% for $k = 48$, its running time increases rapidly to pass the 50% for $k = 368$ and reaching a contribution of 75.5% for $k = 800$ workpackages.

This shows, that 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.

C. Comparison with hotplugged cores

This experiment compares both algorithms when cores are disabled during the whole runtime using CPU hotplugging. The STLMS is still executed with 8 Threads which are then distributed among the active cores by the operating system scheduler. MALMS on the other hand uses only the active cores via its pinned threads and distributes the workpackages onto the active cores.

The running times for MALMS and the STLMS are plotted in Figure 3. MALMS uses $k = 48$ workpackages while sorting 10^6 integers on the eight-core machine.

A perfect speedup (PERFECT) is calculated and plotted besides MALMS. The sequential value for the perfect speedup is set equal to the sequential case for MALMS when seven cores are blocked and only one core remains active. The results show that MALMS adapts better. The running time of MALMS and the amount of available cores are proportional to each other, and the running time shows a stable, predictable behavior. The running time for the STLMS on the other hand behaves in a more unpredictable fashion.

When 1, 2 or 3 cores are disabled, the 8 Threads of the STLMS distribute badly onto the remaining cores and thus MALMS achieves a relative advantage of 84%, 82% and 31% respectively. For 4 and 5 disabled cores, MALMS still has an advantage even though the 8 threads of the STLMS should distribute well onto 4 remaining cores. When 7 cores are disabled, thus only one core remains active and both algorithms are executed sequentially, the overhead of MALMS becomes of importance with a 16% disadvantage relative to the STLMS.

We also did the comparison with hotplugged cores for STLMS using more than 8 threads and thus using more workpackages. But STLMS with 24, 48 and 100 threads was just generally slower than STLMS with 8 threads. In particular, the disadvantage of STLMS when 1, 2 or 3 cores are disabled could not be reduced.

The experiment with hotplugged cores shows that the STLMS can not always use the available cores efficiently if their number does not fit to the number of threads even when the number of available cores is constant during the execution. MALMS does not have this problem.

D. Comparison with dynamically loaded cores

As the measurements in this experiment have a bigger variance than the measurements in the other experiments we performed one test with the same parameters 100 times and we will present boxplots.

In order to test if the advantage comes from using the load information or from the smaller workpackages, we performed the tests also for MALMS_noinfo which is just MALMS where the receiving of signals from Loadtask is disabled.

Input Size	10^2	10^3	10^4	10^5	10^6	10^7	10^8
STDSORT	172	62	66	80.0	94.9	111.0	126.0
STLMS	4 530	485	64	21.7	18.3	20.0	22.0
STLMS SU	0.04	0.13	1.03	3.7	5.19	5.52	5.72
MALMS $k = 8$	9 942	974	108	25.5	18.8	19.6	22.0
MALMS $k = 8$ SU	0.02	0.06	0.61	3.14	5.06	5.63	5.71
MALMS $k = 24$	15 470	1 515	144	28.3	20.0	20.7	23.2
MALMS $k = 24$ SU	0.01	0.04	0.46	2.84	4.75	5.33	5.43
MALMS $k = 48$	22 287	2156	208	32.7	20.9	21.1	23.5
MALMS $k = 48$ SU	0.01	0.03	0.32	2.45	4.54	5.24	5.37

TABLE II
RUNNING TIME FOR STDSORT, STLMS 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.

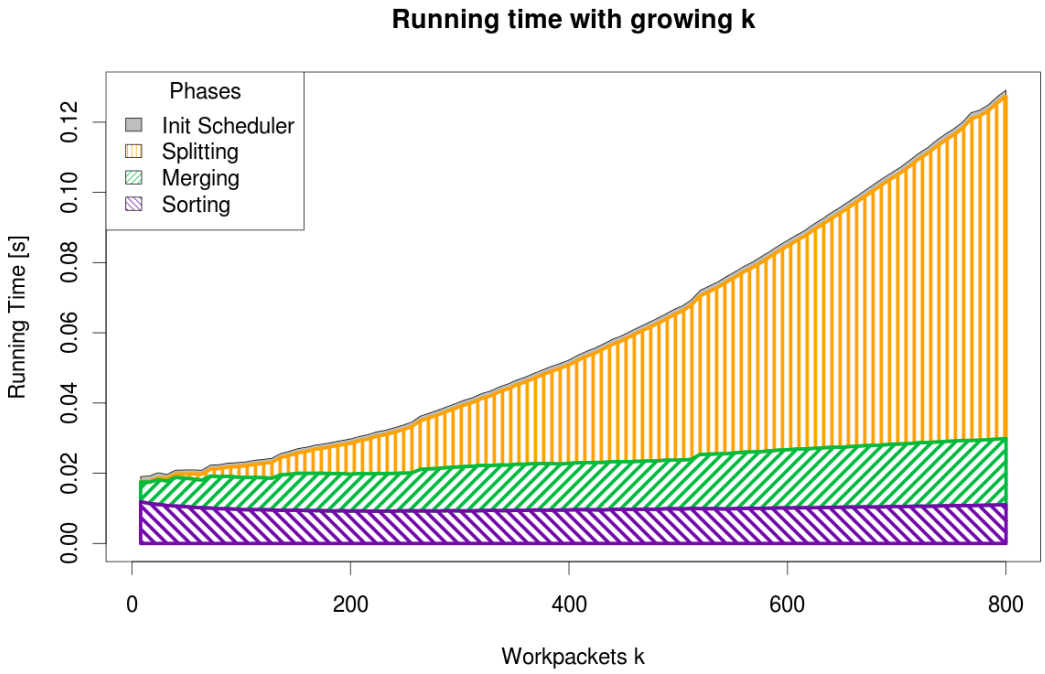
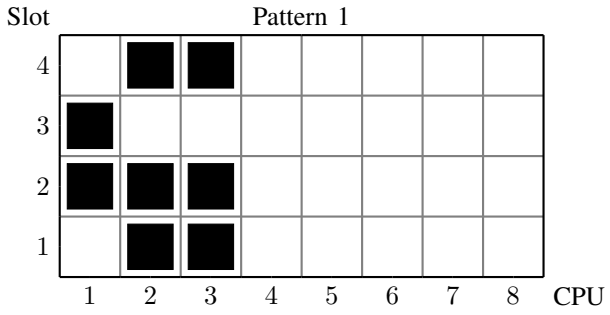


Fig. 2. Running times for the different phases while sorting $n = 10^6$ integers with MALMS using $k = 8 \ldots 800$.

In this experiment we have one other task (Loadtask) running on our system. It changes the used cores and the amount of used cores regularly over time. We startet with pattern 1 which models the behviour 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 will 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 $k = 100$ workpackages and Loadtask running pattern 1 we get a clear advantage of MALMS over STLMS: On average STLMS needs 190% more time than MALMS, and the average speed of Loadtask running in parallel to the STLMS reaches only 77.8% of the speed when it runs in parallel to MALMS. MALMS_noinfo slows down Loadtask about the same as

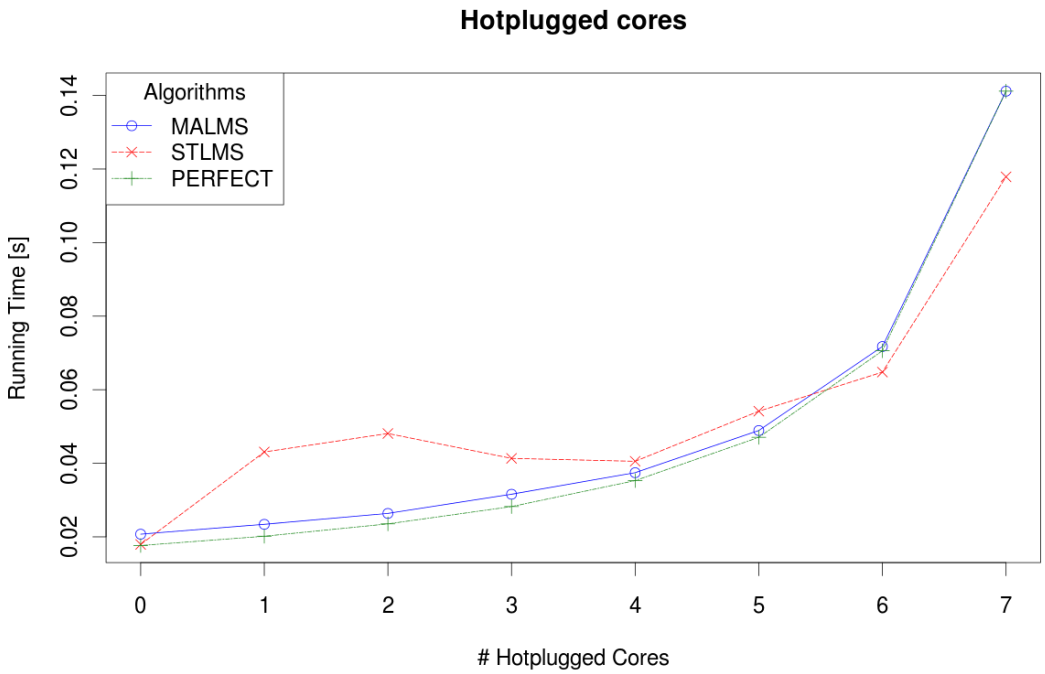


Fig. 3. Running time for sorting $n = 10^6$ integers while cores are disabled via hotplugging. Using $k = 48$ workpackages for MALMS.

STLMS but can coplete the sorting faster. The boxplots of this experiment are given in Figure 4. We also tested with $n = 10^7$ integers to sort 6ms time slots and $k = 100$ workpackages and Loadtask running pattern 1. This produced similar results. On average STLMS needs 43.7% more time than MALMS, and the average speed of Loadtask running in parallel to the STLMS reaches only 93.2% of the speed when it runs in parallel to MALMS (see Figure 5). In general the advantage of MALMS over STLMS is reduced for larger sorting workloads.

As we can see from Figure 4 and Figure 5 MALMS has the additional advantage of a much smaller variance. Hence this shows that the usage of MALMS can improve the global efficiency and reliability of the system. Because of the small variance of runtimes of MALMS this shows a significant improvement of MALMS over STLMS if another job runs in parallel.

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. We tested with $n = 10^7$ integers to sort 6ms time slots and $k = 100$ workpackages and Loadtask running pattern 2 (see Figure 6). On average STLMS needs 23.4% more time than MALMS, and the average speed of Loadtask running in parallel to the STLMS reaches only 81.2% 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. We also tested with $n = 10^6$ integers to sort 2ms time slots and $k = 100$ workpackages and Loadtask running pattern 2. STLMS used 71.8% more time than MALMS on average, and the average speed of Loadtask running in parallel to STLMS was reduced to 85.3% compared to Loadtask running in parallel to MALMS. [**Bilder fuer den kleinen Test mit pattern 2?]** ⊗

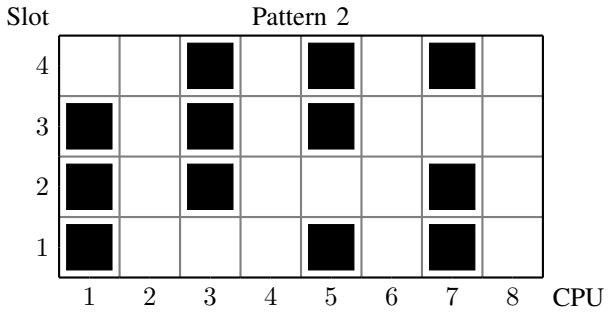
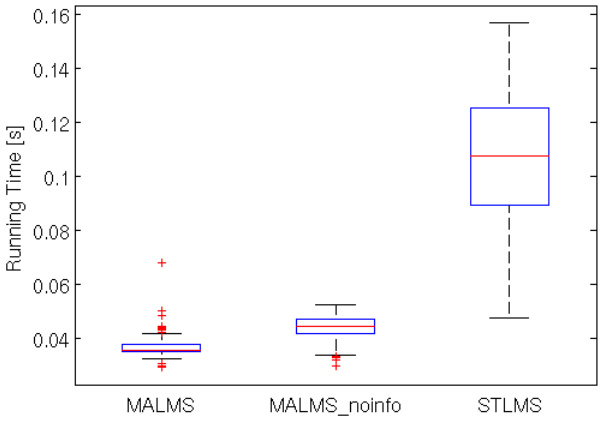


Figure 6 also shows that using the load information is very important, as MALMS_noinfo is even slower as STLMS in the case of $n = 10^7$ integers and pattern 2.

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.



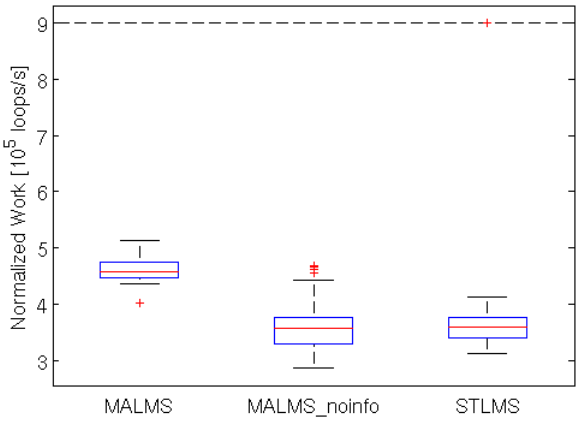
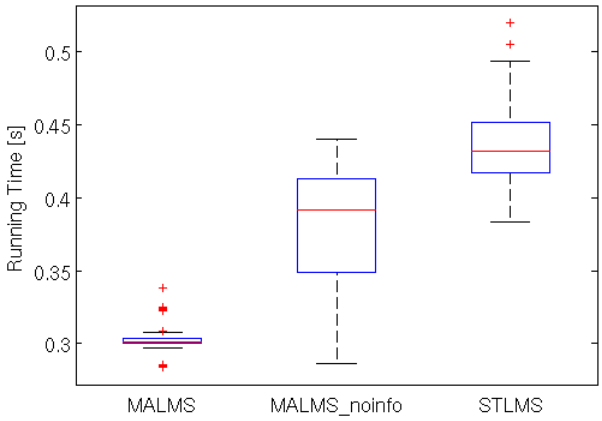


Fig. 4. Results with parallel running Loadtask (pattern 1) for sorting $n = 10^6$ integers 2ms time slots and $k = 100$ workpackages. The second picture shows the performance of Loadtask.



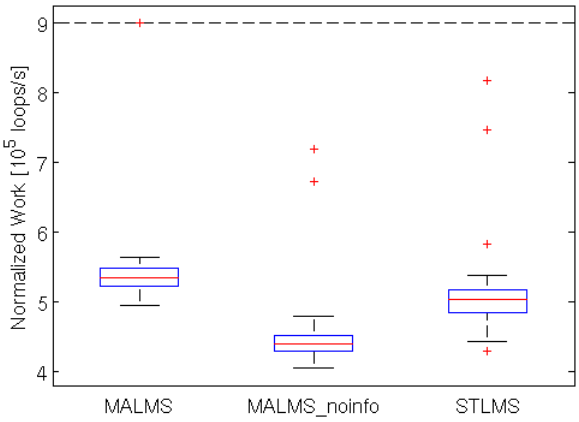
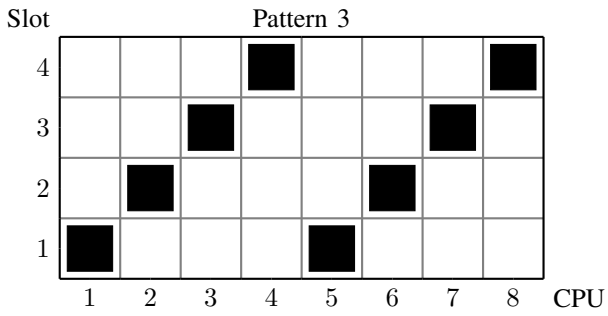


Fig. 5. Results with parallel running Loadtask (pattern 1) for sorting $n = 10^7$ integers 6ms time slots and $k = 100$ workpackages. The second picture shows the performance of Loadtask.



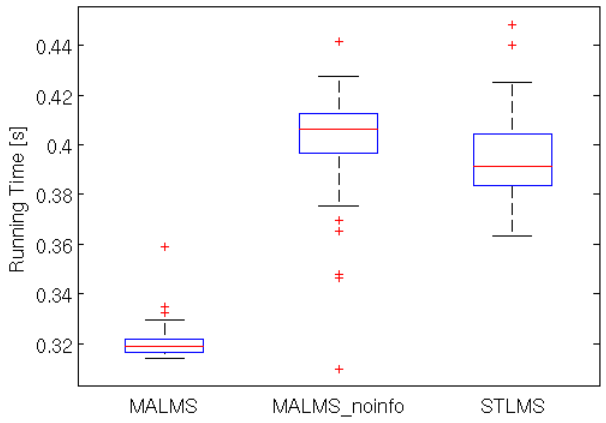
As we can see from Figure 7 and Figure 8 the variance of MALMS is now bigger. In the case of $n = 10^6$ and 2ms time slots we have a small average slowdown of 10.9% of STLMS compared to MALMS but in case of $n = 10^7$ and 6ms time slots STLMS is even 4.2% faster than MALMS on average. But due to the variance of both sorting algorithms in case of pattern 3 we can not tell which is really faster.

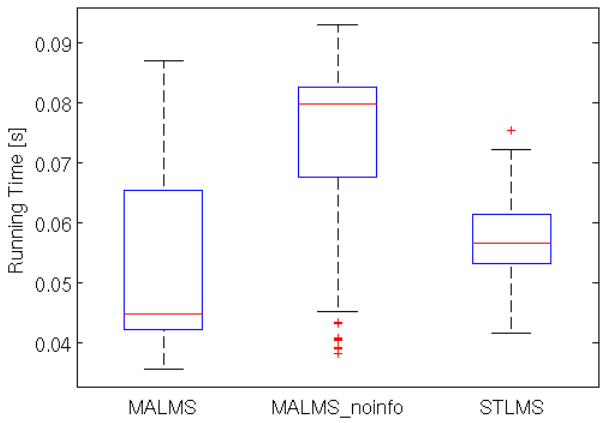
⊗ **[Ergebniszusammenfassung hier]**

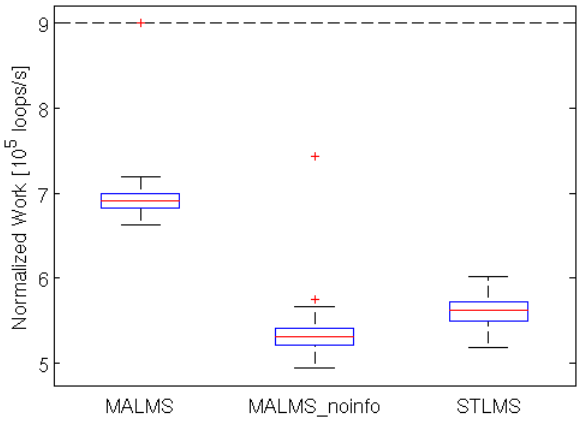
V. Conclusion

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 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. Although our sorter has more overhead than STLMS it is not much slower on an otherwise empty system. 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.

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







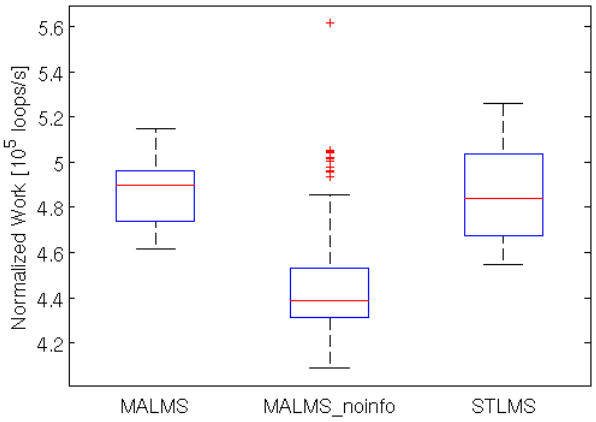


Fig. 6. Results with parallel running Loadtask (pattern 2) for sorting $n = 10^7$ integers 6ms time slots and $k = 100$ workpackages. The second picture shows the performance of Loadtask.

Fig. 7. Results with parallel running Loadtask (pattern 3) for sorting $n = 10^6$ integers 2ms time slots and $k = 100$ workpackages. The second picture shows the performance of Loadtask.

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

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

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

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

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

[9] Ashok Raj. CPU hotplug support in linux(tm) kernel.

[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.

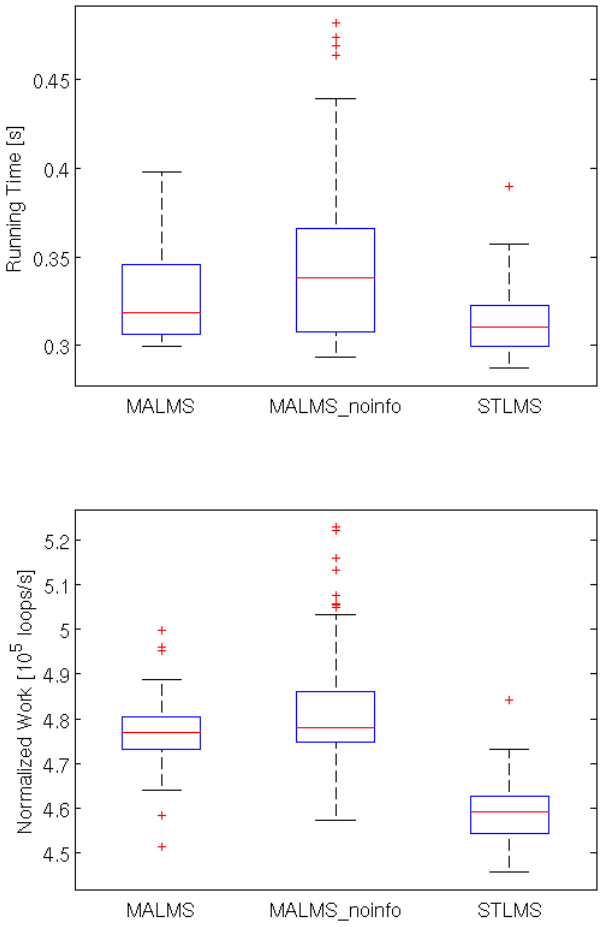


Fig. 8. Results with parallel running Loadtask (pattern 3) for sorting $n = 10^7$ integers 6ms time slots and $k = 100$ workpackages. The second picture shows the performance of Loadtask.