A simple improvement of the work-stealing scheduling algorithm

Željko Vrba, Pål Halvorsen, Carsten Griwodz
Simula Research Laboratory, Oslo, Norway
Department of Informatics, University of Oslo, Norway
{*zvrba,paalh,griff*}*@ifi.uio.no*

Abstract

Work-stealing is the todays algorithm of choice for dynamic load-balancing of irregular parallel applications on multiprocessor systems. We have evaluated the algorithm's efficiency on a variety of workloads, including scatter-gather workloads, which occur in common algorithms such as MapReduce. We have discovered that work-stealing scheduling suffers serious scalability problems with fine-grained parallelism because of contention over run-queues. We therefore propose a simple modification to the work-stealing algorithm that significantly improves its performance on scatter-gather workloads, without any negative impact on other types of workloads.

1. Introduction

There exist many computations that can be relatively easily parallelized, but whose subtasks have irregular CPU demands, e.g., sorting, rendering or video-encoding. Such applications can be parallelized by using frameworks such as Cilk [1] or Threading Building Blocks [2] (TBB). These frameworks encourage application developers to fine-grained parallelism, i.e., to create many more subtasks than there are available CPUs. The division of work among subtasks is often imperfect, and the framework must provide an efficient run-time that can efficiently map ready tasks to CPUs, thus dynamically balancing the workload. One of the simplest, yet best-performing in practice, dynamic load-balancing algorithms for shared-memory architectures is work-stealing [3], which is also used by Cilk and TBB.

In our previous papers [4], [5], we have described Nornir, a run-time environment for executing Kahn process networks in which we have implemented the work-stealing scheduler and a scheduler based on graph-partitioning [6]. We have found that the work-stealing scheduler yields superior performance as long as the ratio of useful work and context-switch overhead is ≥ 75 [7].

In this paper, we analyze performance of the work-stealing scheduling algorithm [3] when used for scheduling scatter-gather workloads, i.e., workloads that use barriers, such as the frequently-used MapReduce framework. Our evaluations show that contention over the run-queues can cause severe performance degradation when subjected to finely-grained scatter-gather workloads. We therefore propose a simple modification to the work-stealing algorithm. Our modification significantly improves the algorithm's performance on scatter-gather workloads, without any negative impact on other types of workloads.

2. Work stealing

The work stealing algorithm uses a m:n scheduling model, where a kernel-level thread is created for each CPU in the system. Each thread has an own run-queue holding tasks that are ready for execution. A thread takes the next task to be executed from the *front* of the queue, and the task runs uninterrupted until it blocks. When a task is about to block, it invokes a user-mode context-switch routine to switch to the thread's scheduling routine which selects the next task to run. A blocked task can be unblocked only by an another, already running, task. An unblocked task is placed at the front of the run-queue belonging to the CPU on which the unblocking task is running. In other words, a thread accesses *its own* run-queue *only at the front*, as shown in in figure 1.

When the thread's own run-queue is empty, the thread enters a busy-waiting loop in which it chooses a random thread ("victim") and tries to steal a task from the *back* of its run-queue. The loop can finish in two ways:

1) The victim's run-queue is not empty, so the steal attempt succeeds, and the stolen task is dispatched.

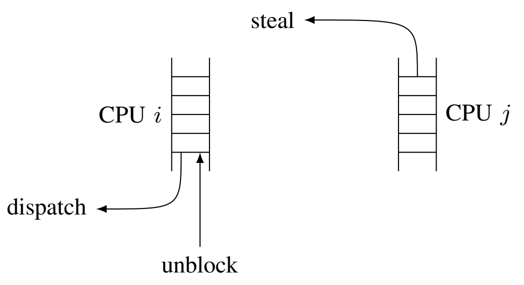

Figure 1. Queue operations in work stealing. CPU i accesses its own queue only at the front, while it steals tasks from other CPUs only from the back.

2) There are no more ready tasks, so the program is finished, and the thread exits. (Consequently, *all* threads will exit.)

If neither is the case, the thread calls the `sched_yield` function to *yield* before starting the next loop iteration.

3. Scheduling in Nornir

Nornir [4], [5] is a run-time environment for executing Kahn process networks [8] (KPNs). A KPN is represented by a directed graph where nodes (tasks) represent computation and arcs (channels) represent communication between tasks. For practical purposes (see [4], [5] for details), channels are assigned finite capacities, and a task *blocks* when it tries to read a message from an empty channel, or when it tries to send a message to a full channel. They may ohterwise run concurrently, and may be scheduled with any *fair* scheduling algorithm. In Nornir, channels are protected from concurrent accesses with a busy-waiting mutex: if a task attempts to lock an already locked mutex, it *yields* to the scheduler, which can then schedule another task.

Nornir uses the m:n scheduling algorithm, where many user-space tasks are mapped to few kernel-level threads. Nornir spawns as many threads as there are CPUs in the machine, binds each thread to its own CPU, and uses the basic idea of the work-stealing algorithm for load-balancing. Scheduling is cooperative, which means that a task must voluntarily yield or block before another task can be scheduled on the same CPU.

We have implemented two variants of the work-stealing algorithm; the difference between them is placement of a newly *unblocked* task. The **original** algorithm, as described in [3] places the task on the same CPU as the task that is doing the unblocking. The **modified** algorithm places the task on the last CPU that it was executed on. Both variants operate in the LIFO manner, i.e., each CPU will execute the most recently unblocked task. This modification actively distributes workload among CPUs, instead of making CPUs to look for more work. Thus, with the original algorithm, a task may be migrated to another CPU by being stolen or unblocked, while with our modified algorithm, a task may only be migrated by being stolen.

In our work-stealing implementation, we have used mutexes to protect the CPU's run-queues instead of the non-blocking queue of Arora et al. [3]. The reason for this is two-fold: 1) simpler implementation that does not negatively impact scalability on up to 8 CPUs (the results of Saha et al. [9] show that even a single, *centralized* queue does not limit scalability on up to 8 CPUs), 2) the non-blocking queue presented in [3] supports concurrent insertions on only one end, so we could not have used it to implement our modification to the original algorithm.

4. Evaluation

We have evaluated the original and modified work-stealing algorithms (further referred to as WS-CUR and WS-LAST,[1] respectively) on the scatter-gather workload as well as on a number of other workloads [5]. Since both algorithm variants give similar performance results on applications other than scatter-gather, we present here only the insights gained from the experiments on the scatter-gather workloads.

Scatter-gather is a synthetic workload that models real situations, such as communication between stages of a MapReduce computation. It is implemented as a KPN running on Nornir, where a single central task (p_0) is communicating with n worker tasks, as shown in figure 2. p_0 executes m *rounds* of the following procedure. First, it sends to each worker a single message representing a workload of w CPU seconds. Upon receipt of a message, a worker spends w seconds of CPU time, and then it sends a reply message back to p_0. In the meantime, p_0 waits to receive a reply from *all* workers, and then begins the next round of distributing work to workers, thus acting as a barrier.

Workers spend CPU time by executing wT_0 iterations of a loop that divides two 64-bit integers. Here, T_0 is the number of loop iterations that uses one second of CPU time on our benchmark machine. To control parallelism granularity, the workload per message w is determined according to the formula $w = T/d$, where both T and d (work-division factor) are user-specified parameters. Thus, the total workload executed by the workers is $W = nmw = nmT/d$ CPU seconds. A

1. The names are mnemonic: the original (CURRENT) algorithm unblocks a task to the current CPU of the unblocking task, whereas the modified algorithm unblocks a task to the LAST CPU it ran on.

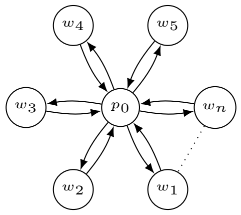

Figure 2. Scatter-gather topology.

more detailed discussion of this benchmark and the results presented here can be found in [10].

We have run the benchmarks on an otherwise idle 2.6 GHz AMD Opteron machine with 4 dual-core CPUs, 64 GB of RAM running Linux kernel 2.6.27.3. The benchmark program has been compiled as 64-bit with GCC 4.3.2 and maximum optimizations (`-m64 -O3 -march=opteron`). Each data point is calculated from the *median*[2] of 9 consecutive measurements of the total real (wall-clock) running time. This metric is most representative because it accurately reflects the real time needed for task completion, which is what the end-users are most interested in. Our investigation is comprised of four different experiments, which we present individually.

Experiment 1: We have first compared the running time the scatter/gather benchmark on 8 CPUs over the running time on 1 CPU. The experiment was run with $n \in \{50, 250, 500, 750, 1000\}$ workers, and work division varying over the set $d \in \{100, 1000, 10000, 20000, \ldots, 10^5\}$. The number of rounds m was computed as $m = \lfloor 50d/n \rfloor$ so that the total amount of work distributed by p_0 is held constant at $W = 50$ CPU seconds. Figure 3 shows the relative speedup on 8 CPUs; for clarity, only the results for $n \in \{50, 250\}$ are presented, but the other experiments show identical behavior. From the figure, it is obvious that for $d \geq 10000$ the performance of WS-CUR starts degrading much more rapidly than the performance of WS-LAST. At $d = 10^5$, the speedup is barely greater than 1 under the WS-CUR algorithm, but it is still above 4 under the WS-LAST algorithm. The speedup decreases as d increases because the useful work performed by each worker decreases, which causes the threads to engage in stealing more often. This, in turn, causes greater contention over the run-queues, and prolongs the thread's waiting time on the

2. We have investigated also correlation with other quantities, such as rate of steal attempts, that we cannot discuss extensively for space reasons. Using median allowed us to use the values of other measured variables *as is*. Mean value would not correspond to any particular measurement, and it would be unclear how other variables could be interpolated. The observed variation in running times is small, so the difference between using median and mean is negligible.

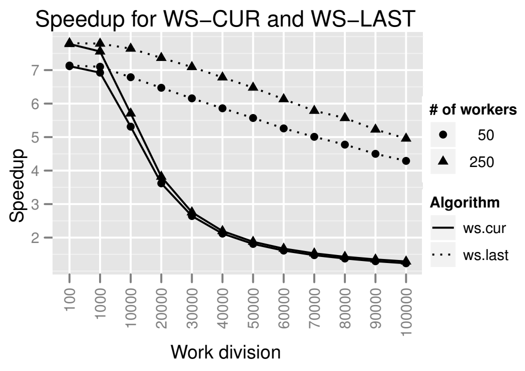

Figure 3. Scatter-gather speedup (8 over 1 CPUs).

run-queue's mutex. For example, at $d = 10000$, WS-CUR performs $\sim 4.2 \cdot 10^5$ steal attempts per second, while WS-LAST performs $\sim 1.1 \cdot 10^5$ steal attempts per second, which is almost four times lesser rate.

We can also see that speedup of the scatter/gather benchmarks increases with the number of workers, independently of work division and the work-stealing variant. The largest increase in speedup is at the transition from 50 to 250 workers. Since 50 workers is already a much larger number than the number of CPUs used in the experiment (8), we have designed another experiment to find the causes of this behavior.

Experiment 2: In this experiment, we further investigate the relation between the number of workers and speedup, which is clearly visible in figure 3. We have fixed work division to $d = 10000$, corresponding to $\sim 100\mu$s of work per message, the number of rounds to $m = 12500$, and we have varied the number of workers over the set $n \in \{8, 16, 32, 64, 128, 192, 256\}$. We have chosen these numbers based on the results shown in figure 3: $d = 10000$ is the critical value at which speedup starts decreasing rapidly, yet it is still reasonably high. The number of rounds m was determined so that the total amount of work performed by all workers for $n = 8$ is at least 10 seconds. In general, the total workload in this benchmark can be calculated as $W = n \cdot 12500 \cdot T/10^4 = 1.25n$ CPU seconds, which also sets the lower bound for running time on 1 CPU. Having more workers or rounds would lead to unreasonably long running times on 1 CPU because the number of rounds and work division are held constant. This is unlike the previous experiment, where the number of rounds m was adjusted in order to hold W constant. Consequently, as n (resp. W) increases, the ratio of useful work to scheduling overheads also increases.

In figure 4, we can see that speedup increases with the number of workers n; at $n = 256$, WS-CUR has

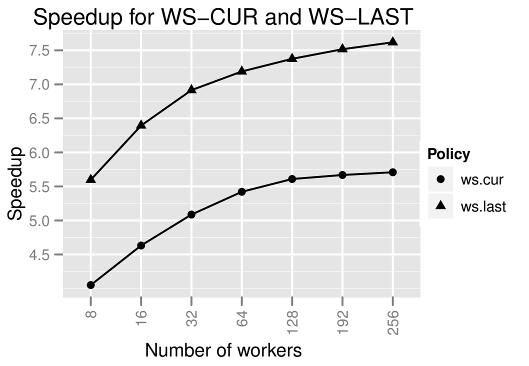

Figure 4. Scatter/gather on 8 CPUs: speedup and steal rate vs. number of workers.

a speedup of 5.7, and WS-LAST has speedup of 7.6. The general trend is that the *rate* of the increase diminshes with n, and that increasing n further would not significantly affect speedup. However, the performance improvement with increasing n is limited, and it did not meet our expectations. We thus conjecture that the key factor in determining efficiency of work-stealing on the scatter-gather workload is CPU time spent on processing a single message.

Experiment 3: With this experiment, we attempt to validate the previous conjecture. We have fixed the number of rounds to $m = 12500$, varied the number of workers over the set $n \in \{16, 50, 250\}$ and work per message over the set $w \in \{25, 50, 100, 200, 400, 800\}$ microseconds. In terms of the scatter/gather benchmark description, we have fixed work division to $d = 10000$ and varied T, which was fixed to 1 in the previous experiments, over the set $T \in \{0.25, 0.5, 1, 2, 4, 8\}$.

We have first noticed that the total running time grows *slower than linearly* with each doubling of work per message w until a certain threshold (see figure 5). The effect is most visible at 16 workers, where we can observe that for $T < 4$, doubling its value leads to an increase on y-axis which is less than 1. Since the y-axis shows the base-2 logarithm of the running time, this means that a two-fold increase in w leads to a *less than two-fold* increase in the running time. This is because CPUs start spending more time executing useful work than spinning in the stealing loop. Under WS-CUR, the effect is visible for all worker counts, while under WS-LAST, the effect is less pronounced at 50 workers and almost non-existent at 250 workers.

In figure 5, we can also see the relation between speedup and CPU time per message. Both algorithms exhibit a significant performance improvement as T increases from 25μs to 100μs, which is correlated with the drop in steal rate. Performance continues to increase as w grows toward 800μs per message, but much more so for the WS-CUR algorithm than for the WS-LAST algorithm. In other words, WS-LAST approaches the maximum speedup at lower values of w which indicates that it is better suited for executing finely-grained parallel applications.

This experiment thus confirms the conjecture that the efficiency of work-stealing is dependent on processing time per message. However, another anomaly has appeared: for $w = 800\mu$s, the speedup at 50 workers is less than the speedup at 16 and 250 workers. We investigate this anomaly in the next experiment.

Experiment 4: To investigate how the number of workers affects speedup, we have fixed work division to $d = 10000$, work per message to $w = 800\mu$s, and varied the number of workers from 16 to 48 in steps of 1. We have investigated only the WS-LAST (modified) algorithm, since the previous experiments have shown that WS-LAST and WS-CUR have similar qualitative performance characteristics, except that WS-CUR yields lower speedup.

Figure 6 shows the speedup of WS-LAST on 8 CPUs, with the plot exhibiting a distinct sawtooth shape. The speedup achieves local minimums for $n \bmod 8 = 1$ and local maximums for $n \bmod 8 = 0$. This indicates that the workers are executed in *batches*. In each round, $\lfloor n/8 \rfloor$ batches are executed in which all 8 CPUs are busy executing workers. If n is not evenly divisible by 8, there will be an *overflow batch* in which only $n \bmod 8$ CPUs are busy executing workers, while the other CPUs are attempting to steal work from other CPUs. However, no additional work exists because the central process (p_0) starts a new round only after all workers have processed their messages in the current round. The overflow batch, if it occurs, stalls $8 - n \bmod 8$ CPUs, which decreases the total speedup in proportion with the number of the stalled CPUs. Thus, the number of workers in scatter-gather workloads should be divisible by the number of CPUs.

We can also see from the figure that speedup maximums are relatively constant, while speedup minimums *increase* with n. This is because the fraction of the total work W performed just by the overflow batch decreases as n increases, which also decreases the time that the stalled CPUs spend in waiting. Thus, the ratio of waiting time and the total running time decreases, which leads to greater speedup.

5. Discussion

We have evaluated performance of the original (WS-CUR) and modified (WS-LAST) work-stealing algorithms, with the main goal being to understand the

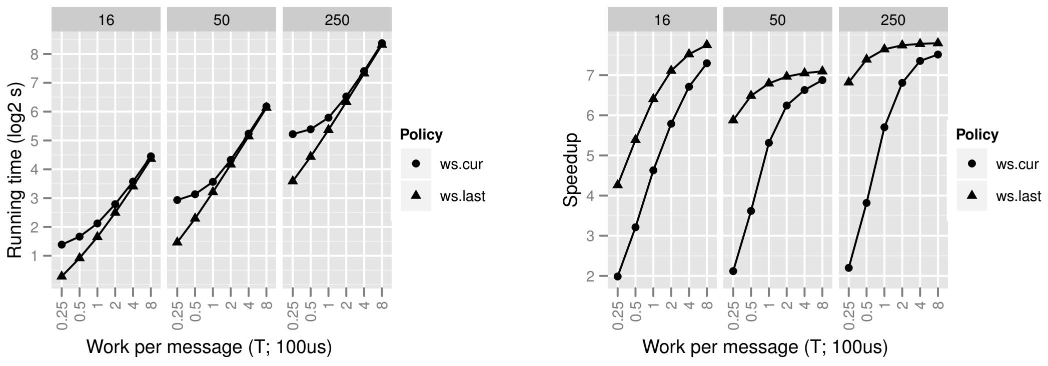

Figure 5. Scatter/gather on 8 CPUs with 16 workers: speedup and steal rate vs. work per message.

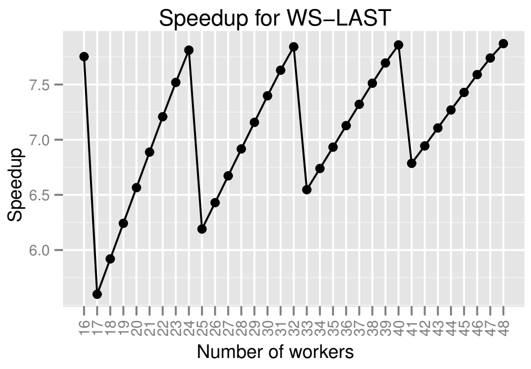

Figure 6. Scatter/gather speedup under WS-LAST on 8 CPUs with 16–48 workers and $T = 8$

performance of work-stealing under different workloads and different parallelism granularities. In this respect, our evaluation complements the work of Saha et al. [9]. Even though they emphasize that fine-grained parallelism is important when there are many CPUs in the system, they have not attempted to *quantify* parallelism granularity. By using a cycle-accurate simulator, they investigated the scalability of work-stealing on a CPU having up to 32 cores, where each core executes 4 hardware threads round-robin. Their main finding is that contention over run-queues generated by work-stealing can limit, or even worsen, application performance as new cores are added, and they suggest that static load-balancing be used in such cases.

We deem that their results do not give a full picture about WS performance, because contention depends on three additional factors, neither of which is discussed in their paper, and all of which can be used to reduce contention:

- Decreasing the number of CPUs to match the *average parallelism* available in the application.
- Overdecomposing an application, i.e., increasing the total *number of tasks* in the system proportionally with the number of CPUs.
- Increasing the *amount of work* a task performs before it blocks again.

The average parellelism is a property intrinsic to the problem at hand and little can be done to change it. Nevertheless, even if the number of CPUs is less than or equal than the application's average parallelism, the choice of other two factors can significantly influence the overall performance.

The scatter-gather workload, which we have used to investigate work-stealing performance, is a common pattern that uses barrier synchronization and which occurs in applications such as MapReduce. The performance of this pattern is hard to characterize, and we had to perform several different experiments to fully understand it. There is a big qualitative difference between WS-CUR and WS-LAST on this workload: as work per message w falls below 1000μs, both algorithms degrade in performance, but WS-CUR much more so than WS-LAST. As w drops from 1000μs to 100μs with 50 workers, the speedup drops from 6.9 to 5.3 under WS-CUR, but much less, from 7.1 to 6.8, under WS-LAST. The trend continues: as w grows further, the speedup under WS-CUR appears to decay exponentially, but under WS-LAST it drops only linearly (see figure 3). The same figure shows that speedup depends on the number of workers n: there is a significant gain as n increases from 50 to 250, and somewhat smaller gain as n increases toward 1000 (not shown in the figure for clarity).

Further experiments have shown that the best speedup is obtained with relatively large work granularity ($w \geq 100\mu$s), but in those cases n must be large or divisible by the number of CPUs N. When n

is small ($n < 50$), and *not* divisible by N, the speedup can be drastically limited, as shown in figure 6. Also, as figure 5 indicates, WS-LAST reaches the speedup plateau for smaller w values than WS-CUR, which indicates that WS-LAST is better-suited to fine-grained parallelism, at least with scatter/gather workloads.

As expected, we have also observed a consistent inversely-proportional relationship between speedup and the steal rate (not shown in the figures due to the lack of space). On 8 CPUs, a satisfactory speedup (≥ 7) is obtained as long as all CPUs together perform fewer than $\sim 10^5$ steal attempts per second.

WS-CUR and WS-LAST have otherwise almost the same performance characteristics for other workloads. Based on the above discussions, we give several guidelines for using work-stealing scheduling algorithms:

- WS gives good performance if each task performs at least 100μs (~ 100 times larger than transaction cost) of work, independently of communication patterns between tasks, between it is waken up and it blocks again.
- Alternatively, the rate of steal attempts, summed over all CPUs, should be held under 10^5 per second. This depends on the application's communication patterns and is hard to control directly; in some cases even more fine-grained parallelism can be achieved, down to 10μs of work per message.
- The speedup of the scatter/gather applications is heavily influenced by the number of worker tasks. We thus recommend that the number of workers be made divisible by the number of CPUs in cases where they all perform approximately equal amount of work.

Finally, another contribution of our experiments is demonstrating that work-stealing can in certain scenarios perform worse than theoretical analyses [3] indicate, even with relatively many workers and coarse-grained parallelism.

6. Conclusions

In this paper, we have studied performance of the work-stealing scheduling algorithm [3] on scatter-gather workloads. These workloads are common, and we have shown that they are particularly elusive to efficient scheduling with work-stealing. We have proposed a simple improvement to the work stealing algorithm that can bring significant performance benefits to scatter-gather workloads, without negative impact on other classes of workloads. Our improvement actively distributes the load and significantly reduces reduces contention over the run-queues. Reducing contention will be even more important for future applications as computer systems are equipped with increasingly many cores.

References

[1] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Montreal, Quebec, Canada, Jun. 1998, pp. 212–223.

[2] Intel Corporation, "Threading building blocks," http://www.threadingbuildingblocks.org.

[3] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *Proceedings of ACM symposium on Parallel algorithms and architectures (SPAA)*. New York, NY, USA: ACM, 1998, pp. 119–129.

[4] Ž. Vrba, P. Halvorsen, and C. Griwodz, "Evaluating the run-time performance of kahn process network implementation techniques on shared-memory multiprocessors," *Complex, Intelligent and Software Intensive Systems, International Conference*, pp. 639–644, 2009.

[5] Ž. Vrba, P. Halvorsen, C. Griwodz, P. Beskow, and D. Johansen, "The nornir run-time system for parallel programs using kahn process networks," in *6th International Conference on Network and Parallel Computing (NPC)*. IEEE Computer Society, October 2009, pp. 1–8.

[6] U. Catalyurek, E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, 2007, also available as Sandia National Labs Tech Report SAND2006-6450C.

[7] Ž. Vrba, H. Espeland, P. Halvorsen, and C. Griwodz, "Limits of work-stealing scheduling," in *Job Scheduling Strategies for Parallel Processing (14th International Workshop, JSSPP 2009)*. Springer Berlin / Heidelberg, May 2009, pp. 280–299.

[8] G. Kahn, "The semantics of a simple language for parallel programming." *Information Processing*, vol. 74, 1974.

[9] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang, "Enabling scalability and performance in a large scale cmp environment," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 73–86, 2007.

[10] Ž. Vrba, "Implementation and performance aspects of kahn process networks," Ph.D. dissertation, Faculty of Mathematics and Natural Sciences, University of Oslo, 2009, no. 903.