Data Structures for Task-based Priority Scheduling

Martin Wimmer Francesco Versaci
Jesper Larsson Träff
Faculty of Informatics, Parallel Computing
Vienna University of Technology
1040 Vienna/Wien, Austria
{wimmer,traff,versaci}@par.tuwien.ac.at

Daniel Cederman Philippas Tsigas
Computer Science and Engineering
Chalmers University of Technology
412 96 Göteborg, Sweden
{cederman,tsigas}@chalmers.se

Abstract

We present three lock-free data structures for priority task scheduling: a priority work-stealing one, a centralized one with ρ-relaxed semantics, and a hybrid one combining both concepts. With the single-source shortest path (SSSP) problem as example, we show how the different approaches affect the prioritization and provide upper bounds on the number of examined nodes. We argue that priority task scheduling allows for an intuitive and easy way to parallelize the SSSP problem, notoriously a hard task. Experimental evidence supports the good scalability of the resulting algorithm.

The larger aim of this work is to understand the trade-offs between scalability and priority guarantees in task scheduling systems. We show that ρ-relaxation is a valuable technique for improving the first, while still allowing semantic constraints to be satisfied: the lock-free, hybrid k-priority data structure can scale as well as work-stealing, while still providing strong priority scheduling guarantees, which depend on the parameter k. Our theoretical results open up possibilities for even more scalable data structures by adopting a weaker form of ρ-relaxation, which still enables the semantic constraints to be respected.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management—Scheduling

Keywords Task-parallelism, priority scheduling, k-priority data structure, work-stealing, parallel single-source shortest paths

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.

ACM 978-1-4503-2656-8/14/02.
http://dx.doi.org/10.1145/2555243.2555278

1. Preliminaries

A `C++` implementation of our data structures and applications is available for download as part of the open source task-scheduling framework Pheet [3, 5][1]. Implementation details, proofs and further references are available in the accompanying technical report [4].

2. ρ-relaxation

In order to improve the scalability of the proposed data structures, we adopt a ρ-relaxation scheme, as introduced by Afek et al. [1], which allows up to ρ items in the data structure to be *ignored*, based on their recency. We say that an item in a priority queue is ignored whenever some other item with lower rank is returned by a `pop`.

The centralized k-priority queue satisfies ρ-relaxation in the following sense: a `pop` operation is allowed to ignore items added by the latest $\rho = k$ `push` operations (in the worst case, the top k by priority). In the hybrid k-priority queue `pop` operations are allowed to ignore the latest k items added by each thread, which implies that, being P the number of threads, up to $\rho = Pk$ items might be ignored.

3. Data structures

3.1 Work-stealing

We adapt work-stealing to priority scheduling, by using local priority queues per thread instead of deques [3]. This preserves the scalability of work-stealing, while imposing local prioritization on the tasks. Due to the decentralized nature of work-stealing, no global priority ordering can be imposed and thus no guarantees can be given on the priority of tasks that are being executed.

3.2 Centralized k-priority data structure

A global priority queue provides strong guarantees on the priority of tasks, but can suffer from congestion. We reduce congestion by adopting ρ-relaxation, which is realized by splitting the data structure into two components: (i) a local

[1] `http://www.pheet.org`

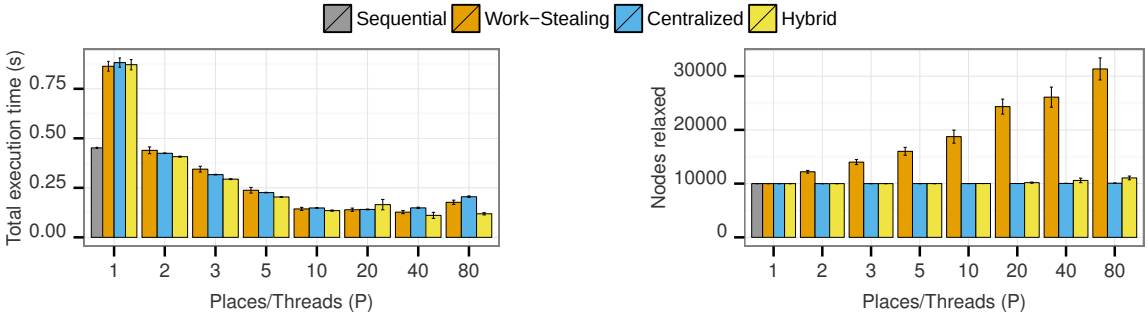

Figure 1. Total execution time and number of nodes relaxed for varying P ($n = 10000$, $k = 512$, $p = 50\%$).

priority queue per thread with references to tasks, and (ii) a global array of tasks. All the tasks stored in the global array at some index $i <$ `tail` are visible to all the threads and referenced in their local priority queues. Newly created tasks are stored in the global array at a random position in the range $\{\texttt{tail}, \ldots, \texttt{tail} + k - 1\}$ (similarly to k-FIFO queues [2]), with the `tail` index being advanced whenever all the k positions are filled.

3.3 Hybrid k-priority data structure

The hybrid k-priority data structure combines the work-stealing and the ρ-relaxation ideas into a single data structure. It consists of three components: (i) a global list, storing tasks visible to all places, (ii) one local task list per thread, of length at most k, and (iii) one priority queue per thread, storing references to tasks in the global and local lists, ordered by priority. Newly created tasks are added to a local task list. When more than k tasks have been added to some local list, it is appended to the global one.

4. Evaluation

Our evaluation is based on a simple parallelization of Dijkstra's algorithm for SSSP, where nodes are speculatively relaxed to increase the available parallelism. Each thread selects the next node to relax based on its tentative distance value, by using the priority data structures presented in this work. Whenever a node is relaxed for which the tentative distance value is not final, this counts as *useless work*, since the node will have to be relaxed again. In our analysis using Erdős-Rényi random graphs we show that the useless work performed when using ρ-relaxed priority data structures can be bounded from above. The theoretical bounds are later verified by simulation.

We also performed an experimental evaluation of our three data structure implementations on an 80-core Intel Xeon system. We show the execution time and the total number of nodes relaxed for executions as a function of the number of threads P (Figure 1). The useless work can be computed by subtracting the number of nodes in the graph (10000) from the number of relaxed nodes. The ρ-relaxed data structures barely produce any useless work for $k \leq 512$, whereas for work-stealing the useless work exceeds the useful work for 20 threads and more.

The parallel implementations are also compared to a sequential implementation of Dijkstra's algorithm (shown only for one thread). Due to the small task granularity, the overhead for parallel execution on all data structures is relatively high, but for two or more threads the execution times drop below the sequential time.

5. Conclusion

We have developed three lock-free data structures for priority scheduling, each of them providing different trade-offs between scalability and guarantees concerning the execution order of tasks. We argue that the hybrid k-priority data structure offers the best compromise between scalability and amount of useless work performed.

Our evaluation shows that ρ-relaxation is a valuable technique to improve scalability, while still enabling strong guarantees. We understood more deeply which properties are required to obtain these guarantees and in future work we plan to further extend priority queues based on this insight: first results on such relaxed k-priority data structures look promising.

References

[1] Y. Afek, G. Korland, and E. Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *OPODIS*, pages 395–410, 2010.

[2] C. M. Kirsch, M. Lippautz, and H. Payer. Fast and scalable, lock-free k-FIFO queues. In *PaCT*, pages 208–223, 2013.

[3] M. Wimmer, D. Cederman, J. L. Träff, and P. Tsigas. Work-stealing with configurable scheduling strategies. In *18th ACM Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 315–316, 2013.

[4] M. Wimmer, D. Cederman, F. Versaci, J. L. Träff, and P. Tsigas. Data structures for task-based priority scheduling. CoRR abs/1312.2501, 2013.

[5] M. Wimmer, M. Pöter, and J. L. Träff. The Pheet task-scheduling framework on the Intel®Xeon Phi™coprocessor and other multicore architectures. In *MTAAP (IPDPS) Workshop*, 2013.