Work-stealing with Configurable Scheduling Strategies

Martin Wimmer Jesper Larsson Träff
Faculty of Informatics, Research Group Parallel Computing
Vienna University of Technology
1040 Vienna/Wien, Austria
{wimmer,traff}@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

Work-stealing systems are typically oblivious to the nature of the tasks they are scheduling. They do not know or take into account how long a task will take to execute or how many subtasks it will spawn. Moreover, task execution order is typically determined by an underlying task storage data structure, and cannot be changed. There are thus possibilities for optimizing task parallel executions by providing information on specific tasks and their preferred execution order to the scheduling system.

We investigate generalizations of work-stealing and introduce a framework enabling applications to dynamically provide hints on the nature of specific tasks using *scheduling strategies*. Strategies can be used to independently control both local task execution and steal order. Strategies allow optimizations on specific tasks, in contrast to more conventional *scheduling policies* that are typically global in scope. Strategies are *composable* and allow different, specific scheduling choices for different parts of an application simultaneously. We have implemented a work-stealing system based on our strategy framework. A series of benchmarks demonstrates beneficial effects that can be achieved with scheduling strategies.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management—Scheduling; D.3.2 [*Programming Languages*]: Language Classifications—concurrent, distributed, and parallel languages

General Terms Algorithms

Keywords Work-stealing, scheduler hints, strategies, priorities

1. Introduction

Work-stealing is a popular way to schedule parallel work-loads of independent tasks [4] and is used by well-known frameworks such as Cilk [3], Cilk++ [9], Intel Threading Building Blocks [8], X10 [5] and others. Standard work-stealing schedulers are oblivious to most properties of individual tasks and treat tasks equally. When this is a drawback, specialized work-stealing systems can apply specific optimizations, taking knowledge of the tasks into account. Such systems can be useful for specific applications, but may be difficult to compose with other applications running at the same time.

PPoPP'13, February 23–27, 2013, Shenzhen, China.
ACM 978-1-4503-1922-5/13/02.

The execution order for local tasks in a work-stealing system is determined by the data structures used for storing the tasks. While the execution order provided by work-stealing deques [1] is good for some applications, there are application kernels for which other execution orders are better. Search-based algorithms can profit from prioritization to explore the most promising branches early. Other applications benefit from giving preference to tasks that access data already in the cache [12]. Locality-aware scheduling policies are also often used [7]. Another heuristic is to prioritize tasks on the critical path [11].

2. Scheduling Strategies

We introduce the concept of *scheduling strategies* as a way of informing the scheduling system about properties of *individual tasks*. Strategies also provide means to prioritize tasks without losing any generality of the scheduler. In addition, strategies are composable, so regardless of which strategies/types of strategies are combined, the scheduling behavior is always well-defined.

2.1 Spawn to call

Strategies make it possible to perform a conversion of task spawns to synchronous function calls based on properties of the task to be spawned and the state of the system. Our system maintains a *transitive weight* estimate of the work that will be generated by a task and its descendants. Below a certain threshold the spawn is converted to a function call. This can depend dynamically on, for example, the number of tasks in the local task queue.

2.2 Number of tasks to steal

Using the transitive weight the number of tasks stolen can be chosen to better reflect the amount of work that will be generated by these tasks, which can improve load balance [2]. Strategies can control the steal operation in this fashion.

2.3 Priority

Strategies can be used to suggest an execution order to the scheduling system, by giving the user a means to prioritize tasks. An application specific execution order of tasks leads to higher efficiency (performance, memory usage, quality of the results) compared to a fixed execution order like *last-in-first-out*.

2.4 Locality

Together with the notion of a *place*, which denotes an execution unit together with its supporting data-structures, the prioritization mechanism can be used to implement per task (spatial and temporal) locality optimizations.

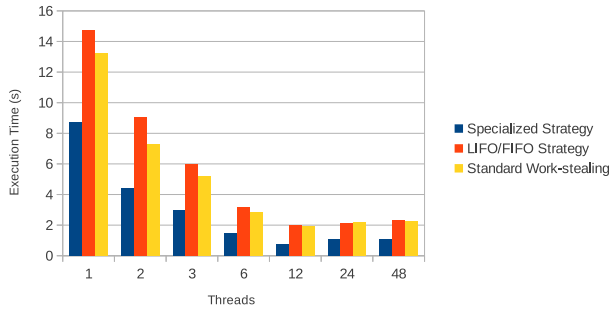

Figure 1. Graph bipartitioning for unweighted graphs. Problem size $n = 39$, density 50%. Results obtained on a 4×12-core AMD Opteron 6168.

2.5 Composability

We achieve *composability* of algorithms that use different (per task) strategies by organizing strategies into a hierarchy, which imposes an order on strategies of different types. Tasks with different strategies are prioritized by the strategy of their common ancestor.

3. Applications and Results

We have used a number of (kernel) applications to illustrate advantages and flexibility of scheduling strategies.

3.1 Graph Bipartitioning

Branch-and-bound algorithms can be implemented in a task-parallel fashion, and benefit from an execution order determined by a weight assigned to each subproblem task. We used strategies to solve the graph-bipartitioning problem using an easily computable lower bound. Figure 1 shows the effect of prioritization vs. a non-prioritized task-execution (LIFO/FIFO), as well as a standard work-stealing system without strategy support.

3.2 Prefix sum

For the prefix-sums problem strategies can be used to make a parallel algorithm (which performs about a factor two more operations than the trivial sequential algorithm) adapt towards the sequential performance when other applications are running at the same time. Strategies lead to better performance when the prefix-sums computation is performed concurrently as part of a larger application.

3.3 Unbalanced Tree Search

The fine-grained Unbalanced Tree Search (UTS) benchmark [10] reduces the scheduler overhead using the more flexible spawn to call conversion that is possible with strategies.

3.4 Triangle strip generation

Using an implementation of the so called SGI algorithm [6], this benchmark explores the simultaneous use of two different prioritization strategies to achieve better results faster.

3.5 Single-source shortest path

The straight-forward parallelization of Dijkstra's single-source shortest path algorithm requires prioritization and can be parallelized in a simple way with strategies.

3.6 Quicksort

This standard example can also benefit from strategies. Good cache behavior is expected if locally spawned tasks are executed depth-first, and the shorter subsequence is processed first. When stealing tasks, the largest subsequences should be stolen first to reduce interference. When enough tasks have been generated, further spawns should be converted to calls. This can all be achieved with strategies.

4. Conclusion

We introduced a dynamic scheduling strategy framework for work-stealing schedulers in order to enable application dependent scheduling decisions. This includes decisions on the execution and stealing order of tasks, as well as on when to merge tasks at runtime. These decisions can reduce scheduling overhead, as well as make the execution more efficient and adaptive. We found considerable improvements to a series of kernel benchmarks.

Acknowledgments

This research was partly funded by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 248481 (project PEPPHER, `www.peppher.eu`).

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3):321–347, 2002.

[2] P. Berenbrink, T. Friedetzky, and L. A. Goldberg. The natural work-stealing algorithm is stable. In *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 178–187, 2001.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[4] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA, pages 519–538, New York, NY, USA, 2005. ACM.

[6] F. Evans, S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In *Proceedings of the 7th conference on Visualization'96*, pages 319–326. IEEE, 1996.

[7] Y. Guo, J. Zhao, V. Cavé, and V. Sarkar. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, 2010.

[8] A. Kukanov and M. J. Voss. The foundations for scalable multi-core software in Intel Threading Building Blocks. *Intel Technology Journal*, 11(4), 2007.

[9] C. E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[10] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C. Tseng. UTS: An unbalanced tree search benchmark. *Languages and Compilers for Parallel Computing*, pages 235–250, 2007.

[11] F. Song, A. YarKhan, and J. Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 19:1–19:11, New York, NY, USA, 2009. ACM.

[12] B. Weissman. Performance counters and state sharing annotations: a unified approach to thread locality. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VIII, pages 127–138, New York, NY, USA, 1998. ACM.