2011 International Conference on Parallel Architectures and Compilation Techniques

Improving Run-Time Scheduling for General-Purpose Parallel Code

Alexandros Tzannes
Dept. of Computer Science
U. of Maryland, College Park
tzannes@cs.umd.edu

Rajeev Barua
Dept. of Electrical & Computer Eng.
U. of Maryland, College Park
barua@umd.edu

Uzi Vishkin
Institute for Advanced Computer Studies
U. of Maryland, College Park
vishkin@umiacs.umd.edu

I. Abstract

Today, almost all desktop and laptop computers are shared-memory multicores, but the code they run is overwhelmingly serial. High level language extensions and libraries (e.g., OpenMP, Cilk++, TBB) make it much easier for programmers to write parallel code than previous approaches (e.g., MPI), in large part thanks to the efficient *work-stealing* scheduler that allows the programmer to expose more parallelism than the actual hardware parallelism. But when the parallel tasks are too short or too many, the scheduling overheads become significant and hurt performance. Because this happens frequently (e.g, data-parallelism, PRAM algorithms), programmers need to manually coarsen tasks for performance by combining many of them into longer tasks.

The need for manual coarsening has three harmful effects on the programmer's *productivity*: *(1)* it is time-consuming, *(2)* it requires programmer expertise, and *(3)* it damages *performance-portability*, as it typically results in *overfitting* to the specific target machine M, input data D [1], and calling context C used to perform the coarsening.

Our first contribution is to distinguish between two types of coarsening, and argue that distinct techniques should be employed in the two cases. First, when tasks are too short, the scheduling overhead *per-task* becomes a significant fraction of the actual work, and tasks need to be coarsened to ***amortize*** the scheduling costs by *increasing the task granularity*. This coarsening depends only on the scheduling cost per-task and the expected length of tasks, and should be done *statically*, so that excessively short tasks are never exposed to the run-time scheduler. Second, when there are too many tasks for a platform, they are typically made available for parallel execution then executed by their creator, which constitutes wasted scheduling overheads. Coarsening is then beneficial to ***reduce*** the amount of parallelism. But variables D, M and C affect the amount parallelism and are not always known at compile-time, in which case, coarsening decisions to reduce parallelism should be done *dynamically* (at run-time). Existing work-stealing schedulers do not perform such dynamic coarsening, making manual coarsening a necessity that sacrifices performance-portability.

Reducing parallelism also increases granularity, but, if there is not much parallelism, it may fail to amortize scheduling overheads. Symmetrically, amortizing scheduling overheads reduces parallelism, but may leave exposed an excessive amount of parallelism. For that reason we distinguish between these two types of coarsening.

LBS[2] is a work-stealing scheduler that reduces parallelism dynamically, by *postponing* task creation until needed, based on load conditions. But, in [2] we evaluated LBS on an experimental architecture with hardware support that allowed it to scale well to 64 cores. Follow-up work [1] noted the possibility of performance degradation on large machines, due to the deviation of LBS from the work-stealing mantra of *breadth-first thefts*, but was unable to demonstrate it on a 16-core platform. The same work also claimed that changing LBS to follow the mantra would require a redesign of their programming language.

Our second contribution is that we demonstrate experimentally the scaling issues of LBS on multicores (Fig.1) and show an easy way to follow the work-stealing mantra and restore scalability, with minimal additional overheads per-task: each worker (thread) keeps its postponed tasks in a linked list and, whenever needed, places the oldest ones onto the shared work-pool. We call the resulting scheduler *Breadth-First Lazy Scheduling (BF-LS)*. Fig.1 shows that BF-LS scales well and greatly outperforms existing approaches on uncoarsened code whilst maintaining performance-portability.

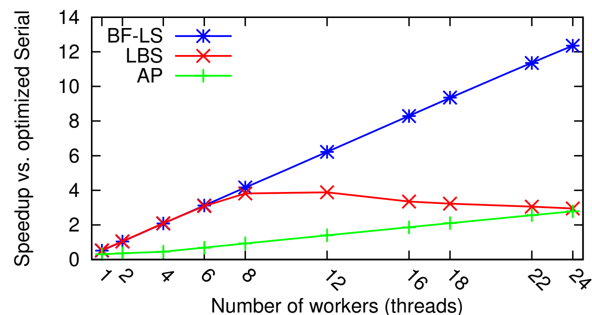

Figure 1. Scaling of BF-LS, LBS, and Auto-Partitioner (AP), the default scheduler in TBB, on a 24-core (4x Xeon E7450). The benchmark finds all solutions to placing 14 queens on an 14x14 chessboard, without them attacking each other, and it is uncoarsened.

References

[1] L. Bergstrom, M. Rainey, J. Reppy, A. Shaw, and M. Fluet, "Lazy Tree Splitting," in *Proc. of ICFP*, September 2010.

[2] A. Tzannes, G. Caragea, R. Barua, and U. Vishkin, "Lazy Binary-Splitting: a Run-Time Adaptive Work-Stealing Scheduler," in *Proc. of PPoPP*, January 2010.

1089-795X/11 $26.00 © 2011 IEEE
DOI 10.1109/PACT.2011.49

IEEE computer society