POSTER: Reducing the Burden of Parallel Loop Schedulers for Many-Core Processors

Mahwish Arif
Queen's University Belfast
m.arif@qub.ac.uk

Hans Vandierendonck
Queen's University Belfast
h.vandierendonck@qub.ac.uk

Abstract

This work proposes a low-overhead half-barrier pattern to schedule fine-grain parallel loops and considers its integration in the Intel OpenMP and Cilkplus schedulers. Experimental evaluation demonstrates that the scheduling overhead of our techniques is 43% lower than Intel OpenMP and 12.1x lower than Cilk. We observe 22% speedup on 48 threads, with a peak of 2.8x speedup.

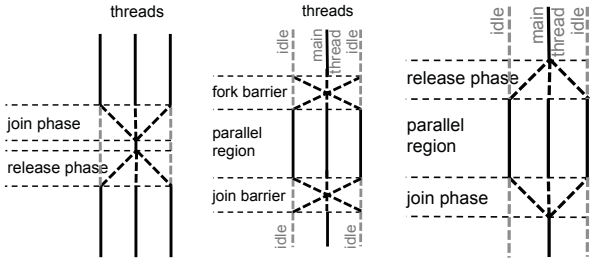
1 Introduction

While Moore's Law remains active, every new processor generation has an increasing number of CPU cores. Scheduling and distributing work load on large scale shared-memory machines becomes increasingly important to make efficient use of the hardware. The runtime overhead caused by scheduling, work distribution and synchronization [1] can make some parallel codes *too fine-grain* to make parallel execution worthwhile. This overhead, growing with the degree of parallelism, can affect the scalability of schedulers.

This work focuses on fine-grain, *micro-second-scale* parallel loops, comparable in duration to the overhead of state-of-the-art schedulers on current hardware. We reason on commonly used loop scheduling techniques and propose a "half-barrier" pattern to remove redundant synchronisation.

2 Contribution

Static scheduling of parallel loops requires the following steps: The master thread 1) divides the loop iteration range among available worker threads, and 2) sends work descriptions to the workers. 3) Workers initialize local copies of reduction variables and execute work sent by the master 4) The master thread waits for the workers to complete, and partial results are reduced for reduction variables.



(a) Barrier structure (b) Parallel loop (c) Half-barrier loop

Figure 1. Schematic structure of threads and synchronization in parallel loops and barriers

The synchronization in step 2 and 4 is implemented using barriers. Barriers involve a *join* and a *release* phase (Figure 1(a)) that respectively records the arrival of threads, and signals threads to enter the next phase of computation. Typically, at least two barriers are executed per parallel loop (Figure 1(b)). The Intel OpenMP runtime implements reductions on top of a barrier-like construct, which effectively introduces an additional barrier.

For the parallel loop model, the worker threads are associated to a specific master which makes some synchronization steps redundant. This implies that worker threads are idle and available for work at the start of parallel regions, and are independent of one another. Hence, we can skip the join part of fork barrier as the threads do not need to wait for each other. Similarly, once the workers notify master about completion of parallel work, there is no need for the master to send acknowledgement. Hence, the release part of join barrier can also be skipped, reducing the synchronization overhead from two barriers (Figure 1(b)) to one barrier (Figure 1(c))

We use a scalable tree barrier algorithm [2] and tune it to the organisation of our evaluation machine.

The half-barrier can be leveraged further to optimise reductions. For static OpenMP loops with reduction variables, the Intel OpenMP runtime executes a tree barrier, in addition to the full barriers at the start and end of the loop, to aggregate per-thread results in the join phase of the tree barrier. Our runtime optimizes performance further by merging the reduction operation with the final join half-barrier. This results in two half-barriers for a parallel loop with reductions, compared to three full-barriers with Intel OpenMP runtime.

We provide an efficient implementation of Cilk reducers for fine-grain loops and allocate their thread-local copies

PPoPP '18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-4982-6/18/02.
https://doi.org/10.1145/3178487.3178517

Table 1. Characterizing scheduler burden

	d (μs)
Fine-grain tree	5.67
Fine-grain centralized	7.55
Fine-grain tree with full-barrier	12.00
OpenMP static	8.12
OpenMP dynamic	31.94
Cilk	68.80

statically at the start of the loop while retaining their programming interface and non-commutativity. The views are reduced as part of pairwise thread synchronization in the join phase of the barrier. This results in exactly $P-1$ reduction operations for P threads, as opposed to baseline Cilk for which operations may be significantly higher.

We extend the Cilk work stealing algorithm to allow static scheduling for fine-grain loops and dynamic scheduling for coarse-grain loops by alternating a cycle of the random work stealing algorithm with polling in the half-barrier.

3 Experimental Evaluation

We evaluate our schedulers on a 4-socket 2.6 GHz Intel Xeon E7-4860 v2 machine with 12 physical cores, 30 MB L3 cache per socket, and CentOS 7.0. We use thread pinning and no hyper-threads. We use the Intel C/C++ compiler v. 14.0.0 for OpenMP and implemented compiler support for the Cilk runtime in clang v. 3.4.1. We calculate speedup against the sequential version of the benchmark.

We first use a micro-benchmark to measure loop scheduling overhead by varying the amount of work in the parallel loop. The speedup is measured for varying granularity of the loop for the OpenMP, Cilk and our fine-grain scheduler, and the scheduling burden is estimated using Amdahl's Law:

$$S = \frac{T}{d + T/48}$$

where T is the sequential execution time, S is the resulting speedup and d is the work distribution time estimated by a least-squares fit between the measurements and the model. The burden of our fine-grain scheduler is 43% lower than OpenMP and 12.1x lower than Cilk (Table 1). Using a half barrier in our scheduler reduces the scheduling delay further compared to a full barrier.

Figure 2 shows the performance of our fine-grain scheduler on Multidimensional Positive Definite Advection Transport Algorithm (MPDATA)[3], from the European Centre for Mid-range Weather Forecasting, on a grid with 5568 points and 16399 edges. The speedup of MPDATA with OpenMP stagnates with increasing parallelism (Figure 2 (left)) whereas the fine-grain scheduler increases performance by up to 22% (Figure 2 (right)) over the off-the-shelf Intel OpenMP runtime.

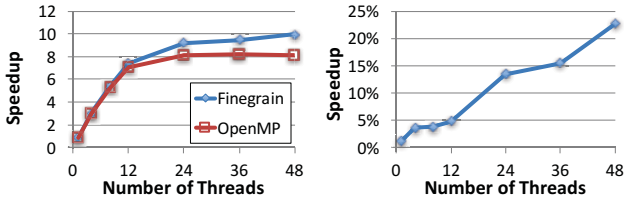
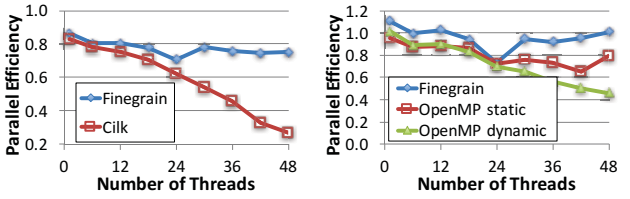


Figure 2. MPDATA: speedup of fine-grain and OpenMP schedulers (left); speedup of fine-grain scheduler over OpenMP (right)



(a) Linear regression, Cilk (b) Linear regression, OpenMP

Figure 3. Performance analysis of map reduce workloads

Figure 3 shows the performance of reductions in our Cilk and OpenMP runtimes using linear regression (medium input from Phoenix++ [4]). Our fine-grain scheduler results in a higher parallel efficiency than baseline Cilk and OpenMP schedulers (Figure 3), owing to the reduced scheduling overhead and efficient implementations of reduction. This leads to a best-case speedup of 2.8.

4 Conclusion

This paper demonstrates that scheduling overhead, increasing with the degrees of parallelism, limits the performance for applications with fine-grain loops. A scheduler tuned to fine-grain parallelism, embedded in the Intel OpenMP and Cilkplus runtimes, provides speedup for such loops by 22% over the baseline OpenMP and Cilk schedulers, which grows with increasing thread count.

References

[1] Y. He, C. E. Leiserson, and W. M. Leiserson. 2010. The Cilkview Scalability Analyzer. In *SPAA '10*. ACM, New York, NY, USA. DOI: http://dx.doi.org/10.1145/1810479.1810509

[2] J. M. Mellor-Crummey and M. L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.* 9 (February 1991). DOI: http://dx.doi.org/10.1145/103727.103729

[3] P. K. Smolarkiewicz *et al.* 2016. A finite-volume module for simulating global all-scale atmospheric flows. *J. Comput. Phys.* 314 (2016).

[4] J. Talbot, R. M. Yoo, and C. Kozyrakis. 2011. Phoenix++: Modular MapReduce for Shared-memory Systems. In *MapReduce '11*. ACM, New York, NY, USA. DOI: http://dx.doi.org/10.1145/1996092.1996095