POSTER: The Performance Impact of Thread Packing on Synchronization-Intensive Applications

Jinsu Park
UNIST
jinsupark@unist.ac.kr

Seongbeom Park
UNIST
amita90@unist.ac.kr

Myeonggyun Han
UNIST
hmg0228@unist.ac.kr

Woongki Baek
UNIST
wbaek@unist.ac.kr

***Abstract*—Thread packing (TP) is a widely-used technique to improve the efficiency of parallel systems. Despite extensive prior works, relatively little work has been done to investigate its performance inefficiencies. To bridge this gap, we quantify its performance impact on synchronization-intensive applications and identify the root causes of its performance inefficiencies.**

***Keywords*-Thread Packing; Performance Characterization;**

I. Introduction

Dynamic concurrency control is an effective technique for high-performance and energy-efficient computing [3, 4, 5, 6, 7, 10]. With dynamic concurrency control, the concurrency level of the target parallel application is dynamically controlled based on the runtime information such as the load intensity of the target application to satisfy the user-specified requirements (e.g., performance, power consumption).

Among the various techniques for dynamic concurrency control, thread packing (TP) [3] is one of the most widely-used techniques [3, 4, 5, 6, 7, 10]. With TP, the OS or runtime system dynamically packs the threads of the target parallel application to an equal or smaller number of cores than the thread count based on the runtime information to improve the overall performance and energy efficiency. One of the main advantages of TP is its high applicability as it can be applied to parallel applications that lack the capability of dynamically changing the thread count.

Despite the extensive prior works that employ TP for dynamic concurrency control, relatively little work has been done to investigate the performance inefficiencies of TP. Without the thorough performance characterization of TP, dynamic concurrency control based on TP is likely to achieve suboptimal performance and energy efficiency. To bridge this gap, we present the in-depth performance characterization of TP and identify the root causes of its performance inefficiencies.

II. Experimental Methodology

System Configuration: We use a 16-core NUMA system with two 8-core Intel E5-2640 CPUs.

Benchmarks: We employ five synchronization-intensive multithreaded benchmarks (Table I), which frequently use the barrier synchronization primitives, from the PARSEC [2], SPLASH [9], and NPB [1, 8] benchmark suites. The thread count of each benchmark is set to the allocated core count for thread reduction[1] and the total core count of the underlying system for thread packing.

Table I: Evaluated synchronization-intensive benchmarks

Benchmark	Dataset	Synch. interval length in ms
Barnes (BA) [9]	Native	596.62 (coarse)
Block TD solver (BT) [8]	C	66.69 (coarse)
Canneal (CA) [2]	Native	2.42 (fine)
Conjugate grad. (CG) [8]	C	4.10 (fine)
Discrete 3D FFT (FT) [8]	C	248.53 (coarse)

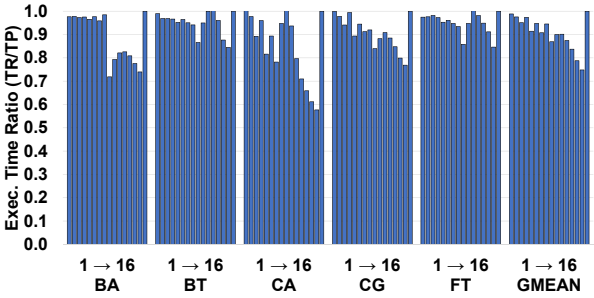

Figure 1: Performance comparison of thread reduction and thread packing with 16 threads and various core counts

Table I shows the synchronization characteristics of the synchronization-intensive benchmarks. The synchronization interval length of each benchmark is defined as the average time (in milliseconds) between two consecutive barriers. If the synchronization interval length of a benchmark is short (or long), it is classified as the one that employs fine-grain (or coarse-grain) synchronization. The synchronization interval length of each benchmark in Table I is collected by executing it with 16 cores and 16 threads.

III. Characterizing Thread Packing

We present the quantitative performance comparison of thread reduction (TR) [3] and thread packing (TP) using

[1]In line with the prior work [3], we refer to a technique that statically sets (reduces) the thread count of the target application to the allocated core count as *thread reduction* (TR). Note that TR itself lacks the capability of dynamic concurrency control as the degree of parallelism (i.e., the thread count) is fixed during the entire execution of the target application.

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00045

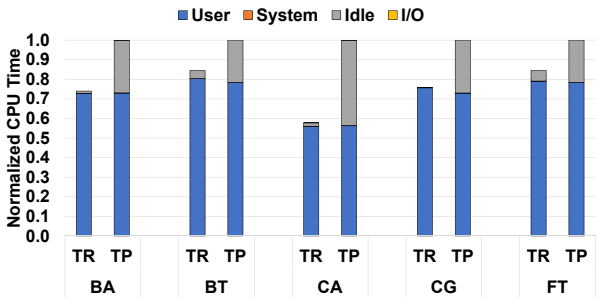

Figure 2: Execution time breakdowns of thread reduction and thread packing with 16 threads on 15 cores

various synchronization-intensive applications, core counts, and thread counts. We compare the performance of TR and TP with the five synchronization-intensive benchmarks and various core counts (i.e., from 1 to 16 cores). The thread counts of the TR and TP versions of each benchmark are set to the allocated core count and 16, respectively. Figure 1 shows the execution time ratio of the TR version to the TP version of each benchmark. In other words, the lower the ratio is, the worse the performance of TP is.

First, TP achieves significantly lower performance than TR when the allocated core count is not a divisor of the thread count of the target application. For instance, TP exhibits 33.7% longer execution time on average (geometric mean) than TR across the synchronization-intensive benchmarks when the allocated core is 15. This is because some of the cores are packed with more threads than the other cores when the core count is not a divisor of the thread count. Since the threads that are packed on the cores with more threads receive less per-thread computation resource, they execute slower than the other threads and become the overall performance bottleneck. In contrast, TR does not suffer from this performance pathology because each thread is allocated with its dedicated core, eliminating any imbalance in per-thread computation resource.

To gain a deeper insight into the overall performance trends, Figure 2 presents the execution time breakdowns of each version of the synchronization-intensive benchmarks, normalized to the TP version. Each bar consists of the CPU time spent for executing the instructions in the user mode (`User`), executing the instructions in the kernel mode (`System`), idling (`Idle`), and handling the I/O operations (`I/O`). Figure 2 shows that the `Idle` time accounts for a larger portion of the total execution time with TP than TR across most of the benchmarks because of the imbalance in per-thread computation resource.

Second, TP incurs more performance degradation with larger core counts. This is mainly because the difference in the per-thread computation resource across threads becomes larger with larger core counts. For instance, with 16 threads and 15 cores, the overloaded core is assigned with 2 threads and the other cores are assigned with 1 thread, resulting in the imbalance factor of 2. In contrast, with 16 threads and 7 cores, the overloaded core is assigned with 3 threads and the other cores are assigned with 2 threads, resulting in the imbalance factor of 1.5 (lower than that with 16 threads and 15 cores).

Third, TP incurs more performance degradation with the benchmarks (e.g., `CA`) that employ fine-grain synchronization. This is mainly because scheduling activities of the Linux kernel are performed at a coarse grain, failing to mitigate the performance degradation of TP.

We summarize the overall performance trends as follows:

- TP achieves significantly lower performance than TR across the evaluated synchronization-intensive benchmarks when the allocated core count is not a divisor of the thread count.
- TP incurs more performance degradation as the allocated core count increases.
- Applications that employ fine-grain synchronization tend to suffer from the performance pathologies of TP more seriously.
- While omitted for conciseness, our experimental results show that the performance pathologies of TP arise with a wide range of system scales.

IV. Conclusions and Future Work

In this paper, we characterize the performance of thread packing (TP) and identify the root causes of its inefficiencies. Our performance characterization results suggest that the performance inefficiencies of TP significantly limit its practicality and must be robustly addressed. As future work, we plan to investigate runtime techniques for improving the efficiency of TP.

Acknowledgements

This research was partly supported by NRF (NRF-2016M3C4A7952587, NRF-2018R1C1B6005961). Woongki Baek is the corresponding author.

References

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. "The NAS Parallel Benchmarks — Summary and Preliminary Results". In: Supercomputing '91.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: PACT '08.

[3] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda. "Pack & Cap: Adaptive DVFS and Thread Packing Under Power Caps". In: MICRO '11.

[4] M. Han, S. Yu, and W. Baek. "Secure and Dynamic Core and Cache Partitioning for Safe and Efficient Server Consolidation". In: CCGrid '18.

[5] R. Nishtala, P. Carpenter, V. Petrucci, and X. Martorell. "Hipster: Hybrid Task Manager for Latency-Critical Cloud Workloads". In: HPCA '17.

[6] J. Park, E. Cho, and W. Baek. "RMC: An Integrated Runtime System for Adaptive Many-core Computing". In: EMSOFT '16.

[7] H. Sasaki, S. Imamura, and K. Inoue. "Coordinated Power-performance Optimization in Manycores". In: PACT '13.

[8] S. Seo, G. Jo, and J. Lee. "Performance characterization of the NAS Parallel Benchmarks in OpenCL". In: IISWC '11.

[9] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations". In: ISCA '95.

[10] H. Zhu and M. Erez. "Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems". In: ASPLOS '16.