POSTER: Managing Application Parallelism via Parallel Efficiency Regulation

Sharanyan Srikanthan, Princeton Ferro, Sayak Chakraborti, Sandhya Dwarkadas
University of Rochester, USA
srikanth,pferro,schakr11,sandhya@cs.rochester.edu

Abstract

Modern multiprocessor systems contain a wealth of compute, memory, and communication network resources, such that multiple applications can often successfully execute on and compete for these resources. Unfortunately, good performance for individual applications in addition to achieving overall system efficiency proves a difficult task, especially for applications with low parallel efficiency (speedup per utilized computational core). Limitations to parallel efficiency arise out of factors such as algorithm design, excess synchronization, limitations in hardware resources, and sub-optimal task placement on CPUs.

In this work, we introduce MAPPER , a Manager of Application Parallelism via Parallel Efficiency Regulation. MAPPER monitors and coordinates all participating applications by making two coupled decisions: how much parallelism to afford to each application, and which specific CPU cores to schedule applications on. While MAPPER can work for generic applications without modifying their parallel runtimes, we introduce a simple interface that can be used by parallel runtime systems for a tighter integration, resulting in better task granularity control. Using MAPPER can result in up to 3.3X speedup, with an average performance improvement of 20%.

CCS Concepts • **Software and its engineering → Process management**; • **Computer systems organization → Multicore architectures**;

1 Introduction

Parallel application scalability can be hampered due to two broad factors: system capability and algorithmic design. System capability is dependent on the type of resources available. Demand for those resources from both the parallel application and other active applications can result in resource saturation. Resource saturation often leads to performance reduction due to resource contention and the resulting excessive queuing delays, diminishing marginal utility. Algorithmic design can result in load imbalance, contention for synchronization, inter-task dependencies, and data sharing. The impact of these factors on performance is dependent on system properties.

In this work, we introduce MAPPER , a Manager of Application Parallelism via Parallel Efficiency Regulation. MAPPER monitors and coordinates all participating applications by making two coupled decisions: how much parallelism to afford to each application, and which specific CPU cores to schedule applications on. MAPPER contains four distinct elements:

- Hardware performance counter-based monitoring used to determine resource needs and bottlenecks.
- Prioritized compute resource allocation based on application resource needs and QoS/fairness guarantees.
- Task mapping to allocated resources to reduce compute, bandwidth, and memory contention, bandwidth utilization, and latency of data access.
- An optional task control within application runtimes.

2 MAPPER Implementation

2.1 Application monitoring

MAPPER transparently monitors applications using low-overhead hardware performance counters. MAPPER uses metrics derived from hardware performance counters commonly available on modern processors to detect and separate the causes behind poor scalability. We use five hardware performance counters and six performance events to compute four performance metrics [5, 6].

2.2 Resource allocation and mapping

The scalability analysis then informs the best resource allocation to meet combined application and system-wide efficiency goals. Resource allocation is achieved by controlling the number of hardware execution contexts allocated to an application. Resource mapping (which specific hardware contexts an application is allowed to use) is used to reduce compute, bandwidth, and memory contention, bandwidth utilization, and latency of data access. Resource allocation and mapping are accomplished using Linux cgroups.

The MAPPER daemon pools knowledge of individual co-running applications' scaling characteristics to prioritize

PPoPP '19, February 16–20, 2019, Washington, DC, USA
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6225-2/19/02.
https://doi.org/10.1145/3293883.3301497

resource allocation for applications that use them more efficiently. Applications are initially granted a fair share of hardware contexts depending on the system load. Applications are guaranteed a certain quality of service, defined as being with a certain percentage of their performance at this fair share. By introducing rare disturbances to applications based on observed hardware bottlenecks, followed by exploration for a performance maxima, the parallel efficiency of an application at different core counts (number of cores used) is collected. MAPPER uses parallel efficiency information to calculate *spare cores*. The spare core count is the number of cores an application can give up while still meeting its quality of service target.

Applications are prioritized according to their parallel efficiency. Periodically, spare cores from applications that scale poorly are reallocated to applications with higher parallel efficiency. Applications that fall below their quality of service target will have their spare cores returned to them.

In addition to decisions on the number of hardware contexts to allocate to each application, MAPPER also carefully maps application threads to specific hardware contexts based on CPU, memory, cache, and interconnect resource needs using heuristics similar to those used by the SAM system [5, 6].

2.3 Optional runtime task control

Information on the allocated hardware contexts may be used to dynamically change task creation within the runtime. MAPPER provides an interface to query resource allocation and exchange progress metrics, which the parallel application runtime can use to control the degree of parallelism of the application. We demonstrate the benefits of task control by modifying the OpenMP runtime system.

3 Evaluation

Experiments were conducted on a quad socket machine with each socket containing an Intel Xeon CPU E7-4820 v3 CPU. Each CPU has 10 physical cores with 2 hardware contexts on each core. We use different benchmark applications gathered from Parsec-v3 [2], OMPSCR-v2 [3], and NAS Parallel Benchmarks [1]. We also use data-sharing intensive graph-based applications (from GraphChi toolkit) that perform machine learning, data mining, pattern recognition, and computer vision [4]. We contrast MAPPER with Linux and two other allocation techniques: hill climbing and fair share.

We performed 3 different evaluations: applications whose runtime were integrated with MAPPER, applications without runtime integration, and a combined application evaluation. Figure 1 shows the impact of MAPPER for combinations of modified/integrated OpenMP applications and applications using other runtimes. Co-running applications were chosen to maximize the diversity of application characteristics analyzed. Minimum QoS for applications was set at 85% of fair share performance. Relative to Linux, MAPPER achieves an

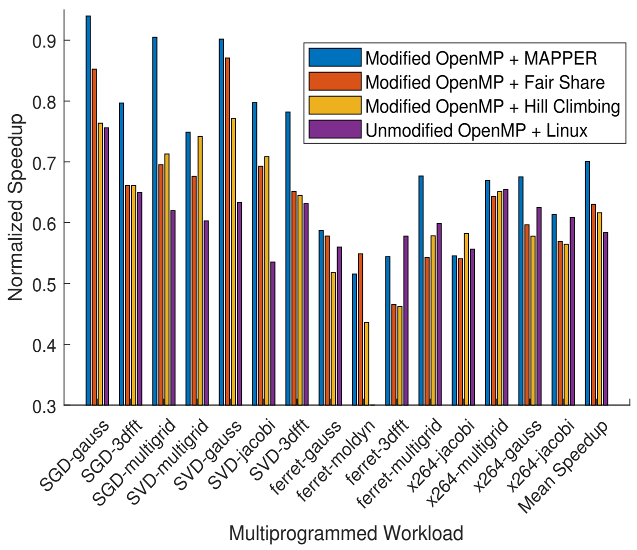

Figure 1. Geometric mean of speedups (relative to standalone performance) achieved by applications in multiprogrammed workloads.

average speedup of 1.2 while maintaining QoS (Linux violates QoS in 23% (compared to 2% for MAPPER) of our test cases with the worst case achieving only 9% (compared to 81% for MAPPER) of fair share performance.).

4 Conclusions

MAPPER [1] demonstrates that combining a normalized efficiency metric that allows comparisons across applications with runtime information on resource demand allows applications to cooperate in achieving both individual QoS and overall system efficiency.

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. 1991. The NAS Parallel Benchmarks. *The International Journal of Supercomputing Applications* 5, 3 (1991), 63–73.

[2] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph.D. Dissertation. Princeton University.

[3] A. J. Dorta, C. Rodriguez, and F. de Sande. 2005. The OpenMP source code repository. In *13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. 244–250.

[4] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX, 31–46.

[5] Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. 2015. Data Sharing or Resource Contention: Toward Performance Transparency on Multicore Systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, 529–540.

[6] Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. 2016. Coherence Stalls or Latency Tolerance: Informed CPU Scheduling for Socket and Core Sharing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 323–336.

[1]This work was supported in part by U.S. National Science Foundation grants CNS-1319353 and CNS-1618497, and by a gift from Futurewei.