Critical Path-Based Thread Placement for NUMA Systems

ChunYi Su
Virginia Tech
Blacksburg, VA, USA
sonicat@vt.edu

Dong Li
Oak Ridge National Lab
Oak Ridge, TN, USA
lid1@ornl.gov

Dimitrios Nikolopoulos*
FORTH-ICS
Heraklion, Crete, GREECE
dsn@ics.forth.gr

Matthew Grove
Virginia Tech
Blacksburg, VA, USA
mat@vt.edu

Kirk Cameron
Virginia Tech
Blacksburg, VA, USA
cameron@vt.edu

Bronis R. de Supinski
LLNL
Livermore, CA, USA
bronis@llnl.gov

ABSTRACT

Multicore multiprocessors use Non Uniform Memory Architecture (NUMA) to improve their scalability. However, NUMA introduces performance penalties due to remote memory accesses. Without efficiently managing data layout and thread mapping to cores, scientific applications, even if they are optimized for NUMA, may suffer performance loss. In this paper, we present an algorithm that optimizes the placement of OpenMP threads on NUMA processors. By collecting information from hardware counters and defining new metrics to capture the effects of thread placement, the algorithm reduces NUMA performance penalty by minimizing the *critical path* of OpenMP parallel regions and by avoiding local memory resource contention. We evaluate our algorithm with NPB benchmarks and achieve performance improvement between 8.13% and 25.68%, compared to the OS default scheduling.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Threads

General Terms

Algorithms,Performance,Management

Keywords

Multicore processors, NUMA, Thread Placement, OpenMP, Critical Path, Shared Resource Contention

1. INTRODUCTION

Non Uniform Memory Architecture (NUMA) has been widely adopted in current systems. Many recent shared-memory multicore multiprocessors, such as IBM Power 7 and the Intel Single-chip Cloud Computer (SCC), use NUMA to dedicate different memory lanes to different processor cores and to distribute system DRAM between processors. Compute nodes of many high-end systems such as the Cray XMT, also use NUMA to provide more memory bandwidth per socket.As the number of cores per node increases, NUMA becomes the memory organization of choice for sustaining scalability in the memory system.

Although NUMA is becoming prominent, it has been the source of performance problems for multi-threaded applications. First, optimizing data locality on NUMA is challenging. Performance optimization for NUMA systems typically relies on optimizing data locality. Such locality optimization may be achieved either with NUMA-aware data placement or with NUMA-aware thread placement to maximize the local data access. However, doing this can potentially create contention on the cache hierarchy and memory controllers, which in turn limits application's scalability [2]. Hence, balancing the local and remote data accesses on NUMA systems is challenging. Second, NUMA may break performance and power optimizations, such as Dynamic Concurrency Throttling (DCT) [1]. DCT dynamically adjusts the number of threads between parallel regions, based on a performance model that predicts the best concurrency configuration for each parallel region. Appropriately selecting the thread number and thread placement for each parallel region can lead to both performance and power improvement for multi-threaded applications. However, when adjusting the concurrency configuration across parallel regions on NUMA systems, the optimized data localization can be easily broken. Third, the conventional OS schedulers do not provide adequate support to solve NUMA issues. OS schedulers emphasize fairness, system throughput and responsiveness, thus ignoring the implication of data locality when scheduling threads.

In this paper, we propose three algorithms to optimize thread placement on NUMA systems. Using hardware event counters that collect data on local and remote memory accesses for each thread during sampled iterations of application code, the three algorithms attempt to map threads close to their associated data. The first algorithm enumerates all possible thread mappings and chooses the one that maximizes local memory accesses. This algorithm, however, does not scale due to its time complexity, which increases with the number of feasible thread to core mappings. Also it does not consider local memory contention. The second algorithm solves these problems. It uses parallel radix sorting to order thread-to-core mappings. Radix sort reduces the time complexity of the algorithm. To avoid local mem-

*Also with the Department of Computer Science, University of Crete, Heraklion, Crete, Greece.

Copyright is held by the author/owner.
PMBS'11, November 13th, 2011, Seattle, Washington, USA.
ACM 978-1-4503-1102-1/11/11.

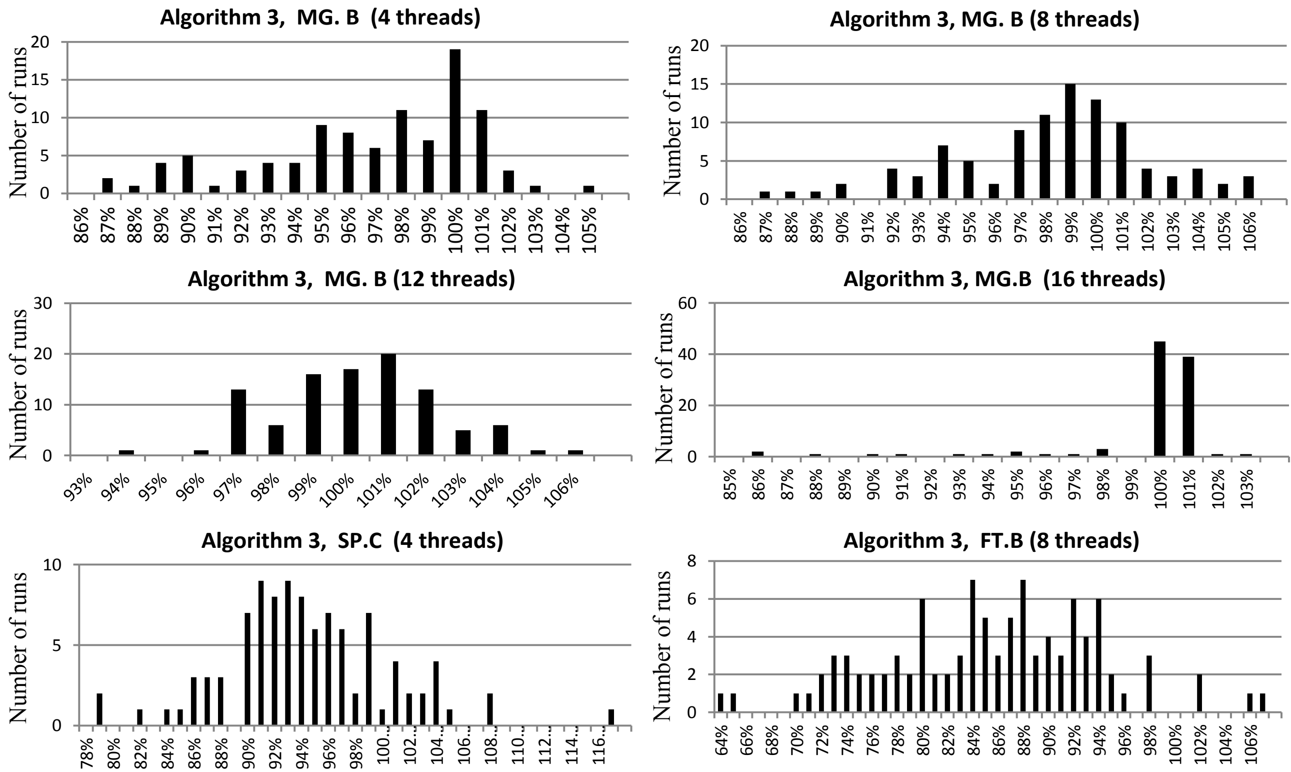

Figure 1: Performance comparison between Algorithm 3 and the system default. X-Axis: Execution time under Algorithm 3 divided by the execution time under the system default mapping

ory contention, the second algorithm sacrifices data locality for selected threads by moving those threads away from the data that they access more frequently, if doing so has a performance benefit based on the algorithm's performance prediction. The second algorithm, however, does not consider performance variation between threads. Due to load imbalance between threads and a varying level of resource contention between memory nodes, a thread may have longer execution time than the others (i.e., the thread may lie on the critical path). We derive a third algorithm that chooses thread mappings that lead to the minimal critical path. This third algorithm defines a new metric, the Impact Factor (IF), which captures the effects of thread placement on both local and remote memory nodes. By selecting a thread mapping with the lowest IF, the third algorithm minimizes the critical path.

2. RESULTS

We implement the three algorithms and apply them to the NAS Parallel Benchmarks Suite (OpenMP version 3.1) on a system with four quad-core AMD Opteron 8350 HE processors (16 cores in total) with Linux (version 2.6.32). Figure 1 displays the results for the third algorithm, compared to the OS default scheduling. We find that this algorithm performs well, no matter how many OpenMP threads are used to execute parallel regions. For the tests with 4, 8 and 12 threads, the third algorithm performs better than the system default scheduling on average by 8.13% and up to 25.68%, because the OS algorithm ignores the critical path, although the OS evenly distributes threads between cores. The third algorithm considers both potential local memory contention and the critical path. Thus, it chooses better thread mappings and improves performance.

3. CONCLUSIONS

NUMA architectures raise significant performance problems due to mismatch between data and thread placement. We propose NUMA-aware, thread placement algorithms guided by the hardware counter events. The best of these algorithms avoids local memory contention and takes into consideration the critical path to improve the performance of parallel applications on NUMA systems.

4. REFERENCES

[1] Curtis-Maury, M., Shah, A., Blagojevic, F., Nikolopoulos, D. S., de Supinski, B. R., and Schulz, M. Prediction Models for Multi-dimensional Power-Performance Optimization on Many Cores. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2008), PACT '08, ACM, pp. 250–259.

[2] Terboven, C., an Mey, D., Schmidl, D., Jin, H., and Reichstein, T. Data and Thread Affinity in OpenMP Programs. In *Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?* (New York, NY, USA, 2008), MAW '08, ACM, pp. 377–384.