MANCHESTER 1824
The University of Manchester

The University of Manchester Research

NUMA-aware scheduling and memory allocation for data-flow task-parallel applications

DOI:
10.1145/2851141.2851193

Link to publication record in Manchester Research Explorer

Citation for published version (APA):
Drebes, A., Pop, A., Heydemann, K., Drach, N., & Cohen, A. (2016). NUMA-aware scheduling and memory allocation for data-flow task-parallel applications. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016* https://doi.org/10.1145/2851141.2851193

Published in:
Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester's Takedown Procedures [http://man.ac.uk/04Y6Bo] or contact uml.scholarlycommunications@manchester.ac.uk providing relevant details, so we can investigate your claim.

Download date:12. Dec. 2024

NUMA-aware Scheduling and Memory Allocation for data-flow task-parallel Applications

Andi Drebes Antoniu Pop
The University of Manchester
School of Computer Science
andi.drebes@manchester.ac.uk

Karine Heydemann
Nathalie Drach
Sorbonne Universités, UPMC Paris 06,
CNRS, UMR 7606, LIP6, France
karine.heydemann@lip6.fr

Albert Cohen
Inria
École Normale Supérieure
albert.cohen@inria.fr

Abstract

Dynamic task parallelism is a popular programming model on shared-memory systems. Compared to data parallel loop-based concurrency, it promises enhanced scalability, load balancing and locality. These promises, however, are undermined by non-uniform memory access (NUMA) systems. We show that it is possible to preserve the uniform hardware abstraction of contemporary task-parallel programming models, for both computing and memory resources, while achieving near-optimal data locality. Our run-time algorithms for NUMA-aware task and data placement are fully automatic, application-independent, performance-portable across NUMA machines, and adapt to dynamic changes. Placement decisions use information about inter-task data dependences and reuse. This information is readily available in the run-time systems of modern task-parallel programming frameworks, and from the operating system regarding the placement of previously allocated memory. Our algorithms take advantage of data-flow style task parallelism, where the privatization of task data enhances scalability through the elimination of false dependences and enables fine-grained dynamic control over the placement of application data. We demonstrate that the benefits of dynamically managing data placement outweigh the privatization cost, even when comparing with target-specific optimizations through static, NUMA-aware data interleaving. Our implementation and the experimental evaluation on a set of high-performance benchmarks executing on a 192-core system with 24 NUMA nodes show that the fraction of local memory accesses can be increased to more than 99%, resulting in a speedup of up to 5× compared to a NUMA-aware hierarchical work-stealing baseline.

1. Introduction

High-performance systems are composed of hundreds of general-purpose computing units and dozens of memory controllers to satisfy the ever-increasing need for computing power and memory bandwidth. Shared memory programming models with fine-grained concurrency have successfully harnessed the computational resources of such architectures (Blumofe et al. 1995; Pratikakis et al. 2011; OpenMP Architecture Review Board 2013; Planas et al. 2009; Pop and Cohen 2013; Charles et al. 2005). In these models, parallelism is exposed by the programmer through the creation of fine-grained units of work, called tasks, and the specification of synchronization that constrains the order of their execution. A run-time system manages execution of the task-parallel application and acts as an abstraction layer between the program and the underlying hardware and software environment. That is, the run-time is responsible for bookkeeping activities necessary for the correctness of the execution (e.g., the creation, destruction and synchronization of tasks), interfacing with the operating system for resource management (e.g., allocation of task data structures, scheduling tasks to cores) and efficient exploitation of the hardware.

This concept relieves the programmer from dealing with details of the target platform and thus greatly improves productivity. Yet it leaves issues related to efficient interaction with system software, efficient exploitation of the hardware, and performance portability to the run-time. On today's systems with non-uniform memory access (NUMA), in which memory latency depends on the distance between the requesting cores and the targeted memory controllers, memory accesses must be kept local in order to reduce latency and data must be distributed across memory controllers to avoid contention. That is, resource-aware task scheduling needs to go hand in hand with the optimization of memory accesses through NUMA-aware memory allocation.

The alternative of abstracting only from computing resources and leaving NUMA-specific optimization to the application is far less attractive. The programmer would have to take into account the different characteristics of all target systems, such as the number of NUMA nodes, the amount of memory associated to each node and the latencies for inter-node access. Application data would have to be partitioned properly and to be placed explicitly using operating system specific interfaces. For applications with dynamic behavior, the programmer would also have to provide mechanisms that constantly react to changes throughout the execution. For example, an initial placement with good data locality might have to be revised when data affinities between tasks and data change. Such changes would have to be coordinated with the run-time system to prevent destructive performance interference, introducing a tight and undesired coupling between the run-time and the application.

We show that it is possible to efficiently and portably exploit dynamic task parallelism on NUMA machines without exposing programmers to the complexity of these systems, preserving a simple, uniform abstract view for both memory and computations, yet achieving near-optimal locality of memory accesses.

2. NUMA-aware optimizations

To achieve the goals above, it is necessary for the run-time system to gain full control over task and data placement, as any static

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, contact the Owner/Author. Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481. Copyright 2016 held by Owner/Author. Publication Rights Licensed to ACM.

PPoPP '16 Mar d–d, 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00

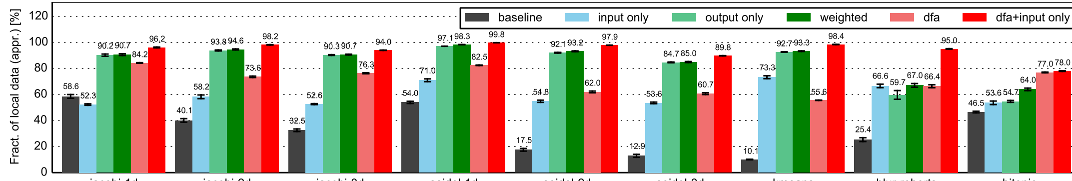

Figure 1: Locality of data managed by the run-time.

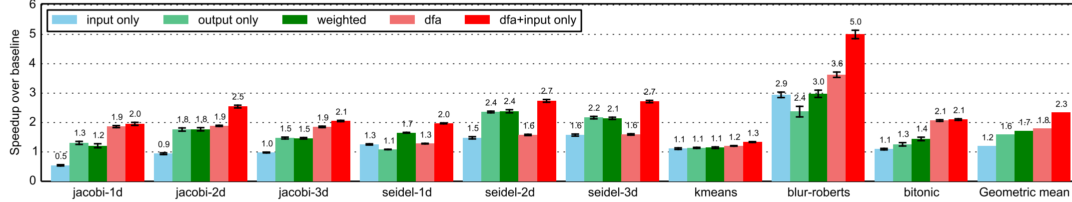

Figure 2: Speedup over the parallel baseline.

placement may conflict with the dynamic control flow of complex applications or dynamic load-balancing in the run-time itself. Such a run-time system needs a dynamic placement algorithm that ensures task input and output data are local and that interacts gracefully with work-stealing:

- The first step is to avoid making early placement decisions which could later prove sub-optimal. In particular, the memory to store task output data should not be allocated until the task placement is known. Our new technique, called *deferred allocation*, hands over the responsibility of allocating memory to the producer task, on its local NUMA node. This is made possible through the *privatization* of task output data.
- The second step is to enhance the locality of read memory accesses. Since the inputs of a task are outputs for another task, the location of input data is determined when the producer task executes. This advocates for an *enhanced work-pushing* technique, building on the algorithm proposed by Drebes et al. (Drebes et al. 2014), and revising it to together with deferred allocation: a task is placed according to the location of its input data *before* allocating memory for its outputs.

This combination of *enhanced work-pushing* and *deferred allocation* is fully automatic, application-independent, portable across NUMA machines and transparently adapts to dynamic changes at run time. These techniques require detailed information about the affinities between tasks and data, but this information is either readily available or can be obtained automatically in the run-times of recent task-parallel programming models, such as StarSs (Planas et al. 2009), OpenMP 4 (OpenMP Architecture Review Board 2013), SWAN (Pratikakis et al. 2011) and OpenStream (Pop and Cohen 2013), which allow the programmer to make inter-task data dependences explicit. While specifying the precise task-level dataflow rather than synchronization constraints alone requires more initial work for programmers, this effort is more than offset by the resulting enhanced performance and performance portability.

3. Experimental evaluation

Our test system, is an SGI UV2000 with 192 cores and 756GiB RAM, distributed over 24 NUMA nodes, and running SUSE Linux Enterprise Server 11 SP3 with kernel 3.0.101-0.46-default. The system is organized in *blades*, each of which contains two Intel Xeon E5-4640 CPUs running at 2.4GHz. Each CPU has 8 cores and has direct access to a memory controller. From a core's perspective, a memory controller can be either local if associated to the same CPU, at 1 hop if on the same blade, at 2 hops if on a different blade that is connected directly to the core's blade or at 3 hops if on a remote blade with an indirect connection.

Figures 1 and 2 show respectively the improvement in locality of memory accesses and speedup of five variants of each benchmark against a parallel baseline. The baseline is a state-of-the-art data-flow implementation that relies on NUMA-aware hierarchical work-stealing to enhance locality. The optimized variants use our work-pushing optimization with either *input only*, *output only* or a *weighted* heuristic or deferred allocation (*dfa*) alone or along with the *input only* heuristic.

Figure 1 shows that our optimizations alow reaching a near-optimal locality of memory accesses, up to 99.8%. Figure 2 shows the speedup of the optimizations over the parallel baseline, reaching up to $5\times$ speedup. The best performance is systematically achieved by combining work-pushing with deferred allocation. The significant performance improvement results from better locality.

Acknowledgements. This work is supported by EPSRC grant EP/M004880/1 and A. Pop is funded by a Royal Academy of Engineering Research Fellowship.

References

R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. of the 5th ACM SIGPLAN Symp. on Princ. and Pract. of Par. Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proc. of the 20th Annual ACM SIGPLAN Conf. on OOP, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

A. Drebes, A. Pop, K. Heydemann, A. Cohen, and N. Drach. Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages. *ACM TACO*, 11(3):30:1–30:25, Aug. 2014.

OpenMP Architecture Review Board. *OpenMP API Version 4.0*, July 2013.

J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with StarSs. *Intl. J. on HPC Arch.*, 23(3):284–299, 2009.

A. Pop and A. Cohen. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM TACO*, 9(4):53:1–53:25, Jan. 2013.

P. Pratikakis, H. Vandierendonck, S. Lyberis, and D. S. Nikolopoulos. A programming model for deterministic task parallelism. In *Proc. of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '11, pages 7–12, New York, NY, USA, 2011.