POSTER: Toward Efficient Architecture-Independent Algorithms for Dynamic Programs

Mohammad Mahdi Javanmard[¶]

Pramod Ganapathi[§]

Rathish Das[¶]

Zafar Ahmad[¶]

Stephen Tschudi^{*}

Rezaul Chowdhury[¶]

[¶]Computer Science, Stony Brook University, NY, USA. [§]Computer Science & Engg., IIT Indore, India. ^{*}Google Inc., USA.

Abstract

Recursive divide-&-conquer algorithms are known for solving dynamic programming (DP) problems efficiently on shared-memory multicore machines. In this work, we extend them to run efficiently also on manycore GPUs and distributed-memory machines without changing their basic structure.

Our GPU algorithms work efficiently even when the data is too large to fit into the host RAM. These are external-memory algorithms based on recursive r -way divide and conquer, where r (≥ 2) varies based on the current depth of the recursion. Our distributed-memory algorithms are also based on multi-way recursive divide and conquer that extends naturally inside each shared-memory multicore/manycore compute node. We show that these algorithms are work-optimal and have low latency and bandwidth bounds.

We also report empirical results for our algorithms.

Keywords GPU, distributed, recursive, divide-&-conquer, dynamic program, I/O efficiency, communication efficiency

1 Introduction

Many of the world's current fastest supercomputers are networks of distributed-memory hybrid compute nodes where each node houses both latency optimized multicores (a.k.a. fat cores) and throughput optimized manycores (a.k.a. thin cores, e.g., GPU cores) connected through a multilevel memory hierarchy (www.top500.org) which is also what an exascale supercomputer is expected to look like in the near future [Kogge & Shalf, Computing in Science & Engineering, 15(6) 2013]. In addition to allowing various types of parallelism, e.g., distributed-memory, shared-memory, task (on multicores) and data (on manycores), a program running on these supercomputers must exploit data locality at various levels of computation for efficiency. Indeed, higher data locality leads to better intra-node I/O and cache performance and lower inter-node communication complexity, which in turn can reduce running times and energy consumption.

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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).

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/10.1145/3293883.3299109>

We argue that the recursive divide-&-conquer paradigm is highly suited for designing efficient algorithms for both shared-memory and distributed-memory architectures. The depth-first recursive decomposition of tasks and data is known to allow computations with potentially high temporal locality, and automatic adaptivity when resource availability (e.g., space in shared caches [1]) changes during runtime. We show that a class of grid-based parallel recursive divide-&-conquer algorithms for solving *dynamic programming* (DP) problems can be run with provably optimal or near-optimal performance bounds on fat cores (cache complexity), thin cores (data movements), and purely distributed-memory machines (communication complexity) without changing their basic structure. We call them $\mathcal{R}\text{-}\mathcal{DP}$ algorithms.

Our contributions. We extend 2-way $\mathcal{R}\text{-}\mathcal{DP}$ algorithms designed to solve DP problems efficiently on shared-memory multicores to run efficiently also on GPUs and distributed-memory machines. Our algorithms, called r -way $\mathcal{R}\text{-}\mathcal{DP}$ s, perform r -way ($r \geq 2$) recursive divide and conquer, where r varies with the level of recursion. Our approach works for the *fractal DP* class [1] that includes Floyd-Warshall's APSP (FW-APSP), the parenthesis problem, pairwise sequence alignment, and the gap problem among many others.

We prove theoretical performance bounds for our algorithms and present empirical results.

2 Multi-way Recursive Divide & Conquer

To obtain an r -way $\mathcal{R}\text{-}\mathcal{DP}$ from a 2-way $\mathcal{R}\text{-}\mathcal{DP}$, where $r = 2^t$ for some integer $t \geq 1$, we start with $t = 1$ and keep applying the following two steps until we can identify the pattern of recursive functions calls in the resulting unrolled $\mathcal{R}\text{-}\mathcal{DP}$.

STEP 1. Take the 2^t -way $\mathcal{R}\text{-}\mathcal{DP}$, and unroll each recursive function call by one level based on its 2-way $\mathcal{R}\text{-}\mathcal{DP}$ version.

STEP 2. Reorder the recursive function calls in the unrolled version from step 1 to call each of them as early as possible without violating dependency constraints.

The resulting 2^{t+1} -way $\mathcal{R}\text{-}\mathcal{DP}$ will have potentially higher parallelism than its 2^t -way version.

Figure 1 shows an r -way $\mathcal{R}\text{-}\mathcal{DP}$ for the parenthesis problem. The initial function call is $\mathcal{A}_{par}(C, C, C, 1)$, where C is the original $n \times n$ DP table for the problem. The precomputed value of r at each recursion level d is stored in $tileSize[d]$.

3 External-memory GPU Algorithms

We consider a memory hierarchy that includes the GPU shared memory, the GPU global memory, the CPU RAM and the disk. We assume that the input DP table is stored either in

Figure 1. An r -way $\mathcal{R}$ -DP for parenthesis problem [2]. Here, X, U and V are $m \times m$ tables. Figure 2. Speedup of cpu-rdp (2-way parallel $\mathcal{R}$ -DP on CPU) and gpu-rdp (r -way parallel $\mathcal{R}$ -DP on GPU) programs over cpu-tdp (iterative serial program on CPU). Also, included are three tiled iterative GPU programs: Harish and Narayanan's [HiPC, 2007] (gpu-tidp-harish), Lund and Smith's [arXiv:1001.4108, 2010] (gpu-tidp-lund), and Katz and Kider's [EUROGRAPHICS, 2008] (gpu-tidp-katz). Buluc et al.'s GPU implementation of the 2-way R-Kleene algorithm [Parallel Computing, 36(5), 2010] (gpu-rec-buluc) and r -way $\mathcal{R}$ -DP with Buluc et al.'s optimization for matrix-multiplication-like functions are also included.

Figure 3. Strong and weak scaling of our distributed-memory algorithm for FW-APSP.

the RAM or in the disk. In our r -way $\mathcal{R}$ -DP algorithms, we use $r = 2$ at every level of recursion until the input fits in the GPU global memory. At that level we choose a large enough r to decompose the input into chunks of small enough size that fit in the GPU shared memory. We do not use the knowledge of the CPU RAM size as STXXL (stxxl.sourceforge.net) can maintain the RAM as a fully associative memory with an automatic LRU page replacement policy. Also, through the choice of r we resort to iterative tiling once we reach inside the GPU as the device provides limited support for recursion.

We prove that our algorithms are I/O-optimal between every two adjacent levels of the memory hierarchy.

We have implemented our algorithms for FW-APSP, parenthesis problem, gap problem, and Gaussian elimination w/o pivoting. Our programs run significantly faster than all internal-memory multicore CPU implementations and almost all existing tiled GPU implementations (see, e.g., Figure 2).

4 Distributed-memory Algorithms

We modify the top level of the recursion of our r -way $\mathcal{R}$ -DP algorithms by choosing r based on the number of compute

nodes available and adding appropriate communication instructions. Thus only the first level of recursion under function $\mathcal{A}_{par}$ has a distributed-memory implementation. Every other function and every other level of recursion of function $\mathcal{A}$ run completely inside a compute node and thus has either a multicore or a manycore (GPU) implementation.

Our distributed-memory algorithms are work-optimal and have latency and bandwidth bounds that are within $\log p$ factor of optimal, where p is the number of compute nodes.

We have implemented our algorithms for Floyd-Warshall's APSP, the parenthesis problem and the gap problem. All of them show good scalability properties (see, e.g., Figure 3).

Acknowledgments

NSF grants CCF-1439084, CNS-1553510, CCF-1725428.

References

- [1] R. Chowdhury, P. Ganapathi, J. J. Tithi, C. Bachmeier, B. Kuszmaul, C. Leiserson, A. Solar-Lezama, and Y. Tang. 2016. AutoGen: automatic discovery of cache-oblivious parallel recursive algorithms for solving dynamic programs. In *PPoPP*. 10.
- [2] R. Chowdhury and V. Ramachandran. 2008. Cache-efficient dynamic programming algorithms for multicores. In *SPAA*. 207–216.