Automatic Discovery of Multi-level Parallelism in MATLAB

Arun Chauhan
Indiana University and Google Inc.
achauhan@cs.indiana.edu

Pushkar Ratnalikar
Indiana University
pratnali@cs.indiana.edu

Abstract

The popularity of MATLAB in scientific and engineering domains is tempered by its performance. Highly optimized libraries, automatic thread-level parallelism for large computations within libraries, and loop-level parallel constructs in the language are some of the ways in which the language implementers have tried to recoup the performance. Greater potential for parallelism exists in typical MATLAB programs that remains unexploited. We discuss our MathWorks-sponsored effort in automatically exploiting parallelism in MATLAB programs using a combination of compile-time and run-time techniques. Our approach is inspired by data-flow-style computation and makes use of some modern C++ libraries for generating highly readable code with support for data parallelism and GPUs.

1 Motivation and Design

Computing on modern high performance machines afford parallelism at multiple levels, from the vector instructions on a single core to multiple multi-core nodes connected through fast interconnects. Graphical Processing Units (GPUs) add heterogeneity and scheduling complexity to the mix.

In order to shield non-expert users from the complexities and interactions of the various forms of parallelism it is often wrapped inside libraries. The libraries are carefully optimized to make use of the vector instructions of the underlying hardware, to use multiple threads when the amount of computation makes it worthwhile, and to provide versions that might utilize accelerators, such as GPUs. Indeed, this is the dominant approach to parallelism in MATLAB. With a few exceptions, such as having to specify the computations to be performed on GPUs, the process is largely automatic for the users and, hence, highly attractive from the perspective of programmability. However, it suffers from two major inefficiencies: the decisions about parallelism must be made inside libraries, which are only locally optimal, at best; and parallelism across library functions is hard to exploit.

Our primary motivation behind this work is to eliminate these inefficiencies without burdening the user with additional code annotations, such as those required for using MATLAB parallel constructs. The high-level nature of MATLAB makes code analysis sufficiently accurate in the common cases that the compiler is able to expose the parallelism that MATLAB libraries would be unable to exploit and which would be non-trivial to express with the repertoire of MATLAB's parallel constructs. In order to make full use of the parallelism we emit C++ code, instead of MATLAB, which lets us use a custom run-time system combined with modern C++ libraries, such as Intel Threading Building Blocks (TBB) for task scheduling, Armadillo for data-parallel matrix operations, and Thrust or ArrayFire for GPUs[1]. The MATLAB libraries continue to be available to the translated code. However, their lack of reentrance property prevents us from making concurrent calls to any single MATLAB library function. Figure 1 shows the overall system.

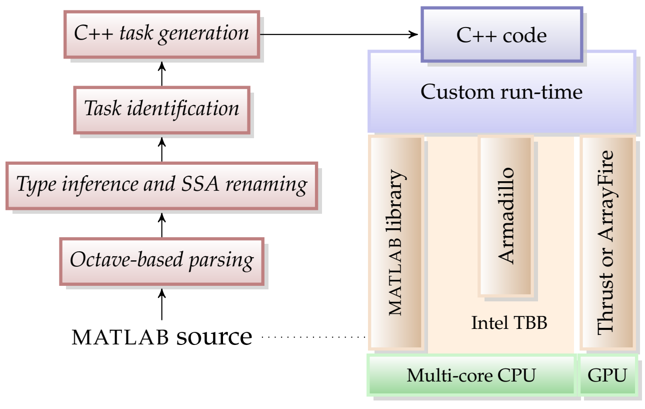

Figure 1: System components.

Our computation model is inspired by coarse-grained data-flow computation [3], which is also implicit in streaming applications and has been used in large practical systems, such as MillWheel at Google [1]. The wide applicability of the model makes it a powerful mechanism to exploit parallelism at multiple levels and scales, including heterogenous parallelism with GPUs.

[1] http://www.threadingbuildingblocks.org, http://arma.sourceforge.net, http://docs.nvidia.com/cuda/thrust, http://arrayfire.com/

2 Approach

The compiler takes as input a MATLAB function to be parallelized. It performs static type inference, primarily to determine the array variables and their sizes, if possible. No additional input is required from the user, except type hints in cases where it is impossible to infer types, e.g., for certain input arguments. The compiler then generates *task specification* in which each task consists of some statements, in the program dependence order, from the original code. A task contains at least one vector statement (i.e., a statement involving arrays) and each vector statement in the original code belongs to exactly one task. These tasks represent the *computational nodes* in the data-flow graph. Each task itself could be data parallel and could be, potentially, scheduled on the GPU [6].

Instead of adhering to strict data-flow semantics we use a hybrid approach that includes two important optimizations for contemporary off-the-shelf hardware, aimed at reducing inter-task communication and memory copies. Scalar statements are liberally replicated to be with the tasks that consume their outputs. In this way, scalar values produced from scalar computations never need to travel across tasks. Arrays are considered mutable to enable partial in-place updating. This also allows arrays to be communicated by reference across tasks in shared-memory (e.g., multi-core) environments.

We rely on the underlying TBB scheduler for scheduling tasks across the cores on a single node. The compiler generates code to create tasks as early as possible to maximize parallelism, while respecting control- and data-dependencies. Arbitrary control dependencies make it challenging to generate data-flow-style code without using extra *gate tasks* or special SSA analysis [2, 4]. We achieve that by using a combination of a smart run-time system, indexing tasks with iteration-vectors, and a simplifying assumption of the dependence distance of one for all loop-carried dependencies, which turns out to not have any significant impact on parallelism in practice [5].

Thanks to the modern libraries—Armadillo for CPU and Thrust or ArrayFire for GPUs—the output code is highly readable and amenable to manual tweaking, if desired.

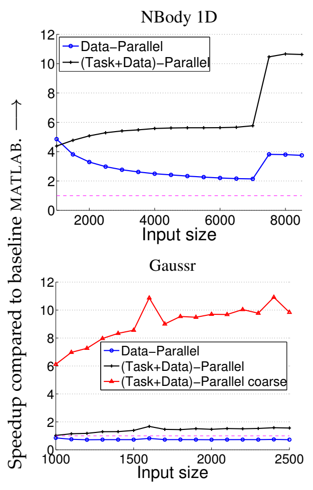

Figure 2: Evaluation.

We have evaluated our current system on several benchmarks and found promising results. Figure 2 shows two examples. In both cases "data-parallel" refers to the C++ code that uses data-parallel libraries but only a single task, and "(task+data)-parallel" refers to the full C++ data-flow-style code. In some cases, such as `Gaussr`, the task granularity must be adjusted before we can get full performance benefits, which is the subject of our ongoing research.

3 Conclusion and Outstanding Issues

Our approach exposes the parallelism in MATLAB code that is often hidden. It unifies multiple types of parallelism, including data parallelism, task parallelism, and accelerator-based heterogenous parallelism. It promises to be scalable with applicability to heterogeneous and distributed address-space tasks. The source code builds on open-source Armadillo and Thrust or ArrayFire libraries and is being prepared for release in open source.

We are currently developing a model to determine optimal task granularity for a given environment, to balance parallelism against communication costs. Enhanced with sufficient flexibility, the model can also help in scheduling decisions, especially on distributed address spaces and GPUs [6]. We are developing static analyses that inform the run-time system for dynamic scheduling of tasks to account for the unpredictability of communication latencies.

References

[1] Tyler Akidau, Alex Balikov, Kaya Bekirouglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, August 2013. DOI: 10.14778/2536222.2536229.

[2] Micah Beck, Richard Johnson, and Keshav Pingali. From control flow to dataflow. *Journal of Parallel and Distributed Computing*, 12(2):118–129, June 1991. DOI: 10.1016/0743-7315(91)90016-3.

[3] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, March 2004. DOI: 10.1145/1013208.1013209.

[4] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. In *Languages and Compilers for Parallel Computing (LCPC)*, volume 1366 of *Lecture Notes in Computer Science*, pages 114–130. Springer Berlin Heidelberg, 1998. DOI: 10.1007/BFb0032687.

[5] Pushkar Ratnalikar and Arun Chauhan. Automatic parallelism through macro dataflow in MATLAB. In *Proceedings of the 27th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2014.

[6] Chun-Yu Shei, Pushkar Ratnalikar, and Arun Chauhan. Automating GPU computing in MATLAB. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 245–254, 2011. DOI: 10.1145/1995896.1995936.