Parallelism in Linnea

Henrik Barthels
AICES, RWTH Aachen University
barthels@aices.rwth-aachen.de

Paolo Bientinesi
AICES, RWTH Aachen University
pauldj@aices.rwth-aachen.de

Abstract

Linnea is an experimental tool for the automatic translation of linear algebra expressions to efficient programs consisting of a sequence of calls to BLAS and LAPACK kernels. Linnea generates programs by constructing a search graph, where each path in the graph represents one program. We introduce two problems related to parallelism that arise in Linnea. Those problems consist in 1) parallelizing the construction of the search graph and 2) generating parallel programs.

Keywords linear algebra, code generation, parallelism

1. Code Generation for Linear Algebra

Linear algebra problems appear in fields as diverse as computational biology, signal processing, communication technology, finite element methods and control theory. The evaluation of linear algebra expressions is a central part of both languages for scientific computing such as Julia and Matlab, and libraries such as Eigen, Blaze, and NumPy. However, the existing strategies are still rather primitive.

At present, the only way to achieve high performance is by handcoding algorithms using libraries such as BLAS [2] and LAPACK [1], a task that requires extensive knowledge in linear algebra, numerical linear algebra and high-performance computing.

We are developing Linnea, a tool that automates the translation of the mathematical description of a linear algebra problem to an efficient sequence of calls to library kernels. The main idea of Linnea is to construct a search graph that represents a large number of programs, taking into account knowledge about linear algebra. The algebraic nature of the domain is used to reduce the size of the search graph, without reducing the size of the search space that is explored. Experiments show that 1) the code generated by Linnea outperforms standard linear algebra languages and libraries, and 2) in contrast to the development time of human experts, the generation takes only few seconds.

2. Parallel Code Generation

Linnea generates a large number of possible algorithms by constructing a search graph. The nodes represent the current state of the computation, the edges are annotated with the kernel calls to get from one state to another. This graph is a directed acyclic graph with one source node and one or more sink nodes. Each path in the graph from the source to a sink node represents one algorithm.

The construction of the graph starts with the source node. In every step of the generation, new successors are added to existing nodes. Since there are usually many different ways to get to the same state of the computation, there is redundancy in the graph. To reduce the size of the graph and speed up the derivation, in every step nodes are merged. Experiments indicate that merging nodes can decrease the size of the search graph by one to two orders of magnitude, resulting in a similar speedup of the derivation.

For problems with sufficiently large matrices, the cost of the derivation is always amortized by the increased performance of the generated code. To ensure that the code generation also pays off for smaller problems, it would be desirable to parallelize the construction of the search graph. While the generation of new successors for existing nodes can be done in parallel, merging nodes would require synchronization between all threads. Furthermore, the time to generate successors for a node is difficult to predict and may vary significantly. To reduce synchronization between threads, one can look at the tradeoff between merging nodes and performing redundant derivations.

3. Generation of Parallel Code

The second form in which parallelism plays a role in Linnea is the the generation of parallel code. The programs generated by Linnea consist of a sequence of calls to library kernels. On the one hand, these calls are often independent from one another and can be executed in parallel; on the other hand, the kernels themselves might offer multithreading. The challenge lies in the distribution of the available resources (computing cores & threads) to the kernels. The range of possibilities is broad, as it includes static and dynamic scheduling and overbooking [3], and the optimal solution is typically found only by trial and error.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' guide*, volume 9. SIAM, 1999.

[2] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[3] E. Peise and P. Bientinesi. The ELAPS Framework - Experimental Linear Algebra Performance Studies. *CoRR*, cs.PF, 2015.