Multithreaded Algorithms for Graph Coloring

Assefaw Gebremedhin [*] Umit Catalyurek [†] John Feo [‡]
Mahantesh Halappanavar [§] Alex Pothen [¶]

Extended Abstract

We present a set of efficient multicore and massively multithreaded algorithms for a prototypical graph problem, graph coloring. The algorithms are implemented—and shown to perform and scale well—on a collection of platforms with varying degrees of multithreading capabilities. The platforms considered include a 128-processor Cray XMT, a 16-core Sun Niagara 2, and an 8-core Intel Nehalem system. We find that obtaining good performance on these machines involves designing algorithms that pay careful attention to and take advantage of the programming abstractions and hardware features the machines provide. The resultant algorithms are different from algorithms that do well on earlier machines that support shared memory and distributed memory programming models.

Graph coloring is an abstraction for partitioning a set of binary-related objects into subsets of independent objects. A need for such a partitioning arises in situations where there is a scarce resource that needs to be utilized optimally. One example is in discovering concurrency in parallel computing. Graph coloring is known to be NP-hard to solve optimally; in fact it is known to be NP-hard to approximate to within $O(n^{1-\epsilon})$ for all $\epsilon > 0$, where n is the number of vertices in the graph [3]. Despite such inapproximability results, however, greedy coloring algorithms that employ good ordering techniques yield near optimal solutions on graphs that arise in practice [2].

Graph algorithms have a number of well recognized features that make them particularly challenging to parallelize with emphasis on performance and scalability: Runtime is dominated by memory latency rather than processor speed; there exist little computation to hide memory access costs; data locality is poor; and available concurrency is low. For these reasons, there are few graph algorithms that perform and scale well on distributed memory machines.

Researchers have had more success on shared-memory platforms, and interest in these platforms is growing with the increasing abundance and popularity of multicore architectures. The primary mechanism for tolerating memory latencies on most shared memory systems is the use of *caches*, but caches have been found to be rather ineffective for many graph algorithms. A more effective mechanism is *multithreading*. By maintaining multiple threads per core and switching among them in the event of a long latency operation, a multithreaded processor uses parallelism to hide latencies. Unlike caches, which "hide" only memory latencies, thread parallelism can hide both memory and synchronization overheads. Thus, multithreaded, shared-memory systems are more suitable platforms for graph algorithms than either distributed memory machines or single-threaded, multicore shared-memory systems.

[*]Department of Computer Science, Purdue University
[†]Departments of Biomedical Informatics and Electrical & Computer Engineering, The Ohio State University
[‡]Pacific Northwest National Laboratory
[§]Pacific Northwest National Laboratory
[¶]Department of Computer Science, Purdue University

In this presentation, we discuss primarily two different parallel distance-1 coloring algorithms we developed for shared-memory, multithreaded systems. The first algorithm relies on *speculation* and *iteration*, and is suitable for any shared-memory system, including multicore platforms. The algorithm is derived from the parallelization framework for coloring on distributed-memory architectures developed in [1]. We benchmarked the algorithm on the Cray XMT, Intel Nehalem, and Sun Niagara 2 systems mentioned earlier. These systems represent a broad spectrum of multithreading capabilities and memory structure: the Cray XMT has a flat, cache-less memory system and utilizes massive multithreading (128 threads per processor) as the sole mechanism for tolerating latencies in data access, the Intel Nehalem relies primarily on a cache-based hierarchical memory system as a means for hiding latencies and supports only two threads per processor, and the Sun Niagara 2 offers a middle path by utilizing a moderate number of hardware-threads along with a hierarchical memory system. We found that the limited parallelism and coarse synchronization of the iterative algorithm fit well with the limited multithreading capabilities of the Sun and Intel processors.

The iterative algorithm ran equally well on the Cray XMT, but it does not take advantage of the system's massively multithreaded processor and hardware support for fast synchronization. To better exploit the XMT's unique hardware features, we developed a fine-grained, *dataflow* algorithm requiring single word synchronization, which is the second algorithm we discuss in this presentation. This XMT-tailored algorithm achieves shorter runtime and uses fewer colors than (the generic) iterative algorithm when run on the XMT. The iterative algorithm has the attractive feature of being portable on different architectures.

We assess the scalability and performance of both algorithms using a set of massive synthetic graphs carefully designed to include instances that test-stress the algorithms. We show that the dataflow algorithm scales well (nearly ideally for certain classes of graphs) on the XMT. The iterative algorithm scales in a similar fashion on all three platforms considered, with increasing relative performance on platforms with greater thread concurrency. Further, the number of colors used by the parallel algorithms is fairly close to what the sequential algorithm uses. In turn, the number of colors the sequential algorithm uses is only a small factor of the optimal (we were able to determine the factory by computing an appropriate lower bound). Hence, there is negligible loss in quality of solution due to parallelization.

In addition to the results on distance-1 coloring, we will briefly discuss results from ongoing work on related topics: parallelization of ordering techniques for reducing the number of colors required; and parallelization of algorithms for other coloring problems needed in the context of automatic differentiation.

References

[1] Doruk Bozdağ, Assefaw Gebremedhin, Fredrik Manne, Erik Boman, and Umit Catalyurek. A framework for scalable greedy coloring on distributed-memory parallel computers. *Journal of Parallel and Distributed Computing*, 68(4):515–535, 2008.

[2] Assefaw Gebremedhin, Duc Nguyen, Alex Pothen, and Mostofa Patwary. ColPack: Graph coloring software for derivative computation and beyond. Submitted to ACM TOMS, 2010.

[3] David Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. *Theory of Computing*, 3:103–128, 2007.