Brief Announcement: Better Speedups for Parallel Max-Flow

George C. Caragea
Dept. of Computer Science
University of Maryland, College Park
george@cs.umd.edu

Uzi Vishkin
Institute for Advanced Computer Studies
University of Maryland, College Park
vishkin@umiacs.umd.edu

ABSTRACT

We present a parallel solution to the Maximum-Flow (Max-Flow) problem, suitable for a modern many-core architecture. We show that by starting from a PRAM algorithm, following an established "programmer's workflow" and targeting XMT, a PRAM-inspired many-core architecture, we achieve significantly higher speed-ups than previous approaches. Comparison with the fastest known serial max-flow implementation on a modern CPU demonstrates for the first time potential for orders-of-magnitude performance improvement for Max-Flow. Using XMT, the PRAM Max-Flow algorithm is also much easier to program than for other parallel platforms, contributing a powerful example toward dual validation of both PRAM algorithmics and XMT.

Categories and Subject Descriptors: C.1.4 Parallel Architectures C.4 Performance of Systems

General Terms: Algorithms, Performance

1. INTRODUCTION

The maximum flow (Max-Flow) problem is a fundamental graph theory problem, with applications in numerous domains. Given a graph G with arc capacities, a source s, and a sink t, the problem is to find a flow of maximum value from s to t. A flow is a function on arcs that satisfies *capacity constraints* for all arcs and *conservation constraints* for all vertices except the source and the sink (see e.g. [8]).

In the serial realm, efficient algorithms and implementations exist (e.g. [2]). In the parallel area, several PRAM algorithms have been proposed [12, 7]. However, even though implementations of the parallel algorithms abound (e.g. [1, 3, 9]), the speedups are quite low considering the parallel machines used (e.g. less than 2.5x speedup for a GPU [9]). This suggests that the Max-Flow algorithm is "difficult to parallelize" on existing parallel platforms.

In this paper we evaluate an efficient, scalable implementation of a PRAM Max-Flow algorithm for explicit multi-threading (XMT), a many-core architecture designed from the ground up to support PRAM-like programming. Using XMT, the algorithm is both easier to program and achieves higher speedups than prior work when compared to the best serial implementation. This supports the argument that the previous low speedups are not caused by inefficient algorithms or implementations, but by the mismatch between the algorithm and the underlying platform. It also strengthens the case for XMT as an efficient, general-purpose, easy-to-program many-core.

SPAA'11, June 4–6, 2011, San Jose, California, USA.
ACM 978-1-4503-0743-7/11/06.

2. MAX-FLOW ALGORITHMS

Numerous **serial Max-Flow algorithms** have been developed over the years which proposed improving complexity bounds. Some early Max-Flow algorithms worked by finding augmenting paths, using the layered network approach of Dinic. Using Karzanov's concept of preflow, Shiloach and Vishkin [12] contributed an alternative method that also introduced parallelism to Max-Flow. Goldberg and Tarjan replaced the layered network approach by introducing the concept of distance labels into the Shiloach-Vishkin (SV) algorithm. Distance labels were easier to manipulate than layered networks and led to asymptotic improvements [8]. In practice however, the Goldberg-Tarjan algorithm had poor performance. Apparently distance labels were more helpful for improving asymptotic results than implementation runtime. Indeed, Goldberg's PhD thesis noted the advantage of *global relabels* (in effect, layered networks) making the implementation closer to the original SV algorithm. The fastest serial implementation that we are aware of (Goldberg, [6]), includes other heuristics and optimizations, such as *gap relabeling* and *highest-level node selection* [5].

There have been fewer improvements when it comes to **parallel algorithms** for Max-Flow. Shiloach and Vishkin [12] proposed a first $O(n^2 \log n)$ time and $O(nm)$ space parallel algorithm for directed graphs. Goldberg and Tarjan [7] introduced an algorithm for acyclic graphs that runs in $O(n \log n)$ time and $O(nm)$ space. Vishkin [14] extended [12] to acyclic graphs and showed that an improved $O(n^2)$ space bound applies to both [12, 14]. By incorporating distance labels into the SV algorithm, Goldberg and Tarjan reduced the space requirement down to $O(m)$ [8]. The resulting algorithm, called Push-Relabel, is the one used as basis for most subsequent implementations.

As was the case for the sequential version, implementations of the basic parallel Push-Relabel algorithm were observed to be slow in practice. All existing parallel implementations are enhanced with global relabeling, which is effectively a periodical breadth-first search (BFS) on the residual graph. Anderson and Setubal [1] evaluated an implementation for a 14-processor SMP, and observed performance improvements of *up to two orders of magnitude due only to global relabeling.* Bader and Sachdeva [3] designed a cache-friendly version of the parallel algorithm for a multiprocessor, adding a parallel gap-relabel heuristic implementation. Both [1, 3] use locks to handle concurrent updates.

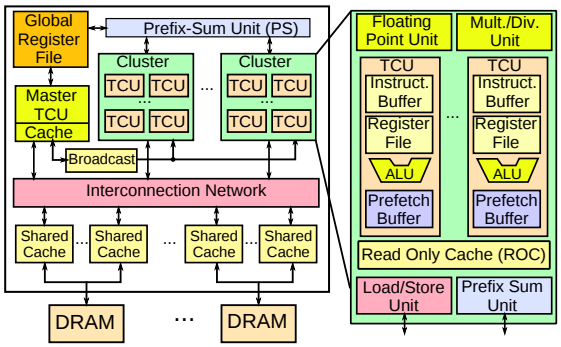

Figure 1: Block diagram of the XMT architecture

Several GPU MaxFlow implementations have also been proposed. [10, 13] present restriced MaxFlow solutions applicable only for grids. He and Hong [9] evaluate a lock-free, hybrid CPU-GPU implementation designed for general graphs, reporting speedups of up to 2.5x over the fastest sequential code. With GPU-like peak computing capacity and bandwidth, better speedups seem possible.

The XMT Max-Flow Space limitations allow very limited review of our XMT Max-Flow implementation. Its PRAM description of the Push-Relabel Max-Flow algorithm starts with a Work-Depth description (introduced in [12]) per the "XMT programmer's workflow" [15] for advancing from a high level abstraction to a program in XMTC, a modest extension of C. Our implementation is completely lock-free, allowing it to scale to many more cores; both the push and relabel operations are split into two phases, one to compute and one to propagate results. The atomic prefix-sum operation is used to coordinate accumulation of flow from multiple sources. We drastically reduced the amount of work by maintaining a list of all active nodes and using only one thread per active node. A parallel BFS is run periodically on the residual graph to perform global relabeling.

3. XMT – AN EASY TO PROGRAM MANY-CORE PLATFORM

The primary goal of the XMT on-chip general-purpose computer architecture (e.g. [15]) has been improving single-task performance through parallelism, a goal orthogonal and complementary to throughput oriented architectures such as some multi-cores. XMT was designed to capitalize on the huge on-chip resources becoming available in order to support the formidable body of knowledge, known as Parallel Random Access Model (PRAM) algorithmics, and the latent, though not widespread, familiarity with it.

The XMT architecture, depicted in Fig. 1, includes an array of lightweight cores or Thread Control Units (TCUs) and a serial core with its own cache (Master TCU). The architecture includes several clusters of TCUs connected by a high-bandwidth low-latency interconnection network. The underlying programming model of the XMT framework is arbitrary CRCW (concurrent read/write) reduced-synchrony PRAM-like model [15], with serial and parallel execution modes. Each thread progresses at its own speed without ever having to busy-wait for other threads, a methodology called "independence of order semantics (IOS)." XMT also incorporates hardware implementation of a powerful atomic prefix-sum primitive that provides constant, low overhead inter-thread coordination. The high-bandwidth interconnection network, the low-overhead creation of many threads and the low-cost synchronization primitives facilitate efficient support for fine-grained, irregular parallelism.

We have prototyped the XMT architecture using FPGA technology. *Paraleap*, a 64-core configuration running at 75MHz, has been in use since 2007 at UMD. It showed the feasibility of a PRAM-On-Chip architecture, and served to run initial performance comparisons with existing CPUs [16]. In addition to Paraleap, we used XMTSim, the XMT cycle-accurate software simulator. Cycle-accuracy of XMTSim is achieved by modeling timing after after both the FPGA and synthesized Verilog code, and it can be customized to realistically simulate larger configurations, beyond the limitations of the prototype. The XMT compiler and simulator are publicly available [11].

Name	Nodes/Edges	Type
AD-1K	1K/500K	acyclic-dense
Random-10K	10K/30K	uniform random
Random-64K	64K/200K	uniform random
RLG-Long	64K/195K	long layered
RLG-Wide	64K/196K	wide layered
RMF-Long	8K/45K	long, parallel grids
RMF-Wide	8K/46K	wide, parallel grids
Rome-99	3.3K/9K	road network

Table 1: Input datasets used.

4. EXPERIMENTAL RESULTS

We evaluated the speedups of our parallel Max-Flow implementation by comparing to the running time of the fastest known serial code **hi_pr** [6] that incorporated an insight from [5]. To collect the serial running time, we ran **hi_pr** on two platforms: (i) a modern 64-bit x86 CPU: a quadcore AMD Phenom 9600B at 2.3GHz, and (ii) the Master TCU of the Paraleap 64-core XMT prototype. Each input was ran 100 times, and the average cycle count was read and recorded. On the x86 platform, the *rdtsc* instruction was used to get accurate timing information.

The parallel implementation ran on two different embodiments of the XMT architecture: (i) the 64-core Paraleap prototype, built using FPGA technology, whose cycle count has been shown to reflect a much faster ASIC; e.g. 800MHz per [16] and (ii) a forward-looking 1024-core configuration simulated using XMTSim. Previous work [4] has shown that this configuration, if built using same technology as today's GPUs, would use approximately the same area as an NVIDIA GTX280 chip, and could run at similar clock speed.

The input datasets we used are listed in Table 1. We used input graphs that represent a broad variety: the acyclic-dense (AD), Random-Level Graph (RLG) and RMF were proposed in the DIMACS challenge (e.g. [2]), and have been widely used to evaluate Max-Flow; in the Random graphs, edges are added between pairs of nodes uniformly at random, resulting in a graph with relatively small diameter; and Rome-99 is a city road network.

Fig. 2 shows our performance results. The figure shows the running times of the implementations, measured in cycles, and normalized to the running time of the x86 serial time (hi_pr.x86). The hi_pr.MTCU shows the running time of the same serial implementation on the Paraleap Master TCU, the serial processor included in the XMT architecture. Since the XMT MTCU is not as optimized as an industry-grade CPU, slow-down was observed compared to the x86 version. We see no reason why, given the same amount of engineering

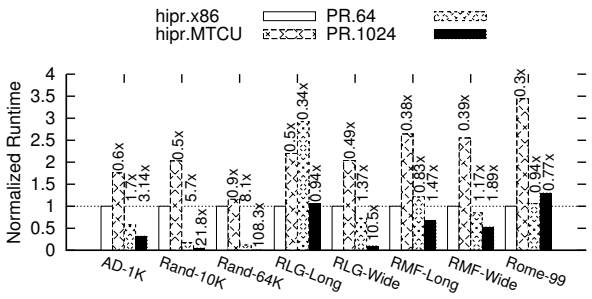

Figure 2: Performance of XMT Parallel Max-Flow. Numbers on bars represent speed-up relative to hipr.x86

effort, the performance of the MTCU would not match a same-generation CPU.

The PR.64 and PR.1024 results show the run time of our parallel implementation on the 64-TCU FPGA XMT, and on the 1024-TCU XMTSim simulator respectively. We observe large performance improvements for the AD and Random graphs, especially the larger Rand-64K. In these graphs, the degree of parallelism is high, keeping the hardware occupancy high, resulting in good performance: speedups range from 3.14x to 108.3x relative to the hi_pr.x86 running time, or 5.53x to 125.73x when compared to hi_pr.MTCU. The RLG-Wide has a large degree of parallelism as well, resulting in good speedups. There is less parallelism in RLG-Long, causing performance to be lower. The 1024-configuration is still significantly faster than the 64-TCU one, since some parts of the algorithm (e.g. the BFS pass) can take take advantage of the more TCUs. The level of parallelism in the Rome99 graph is lower, causing a small slowdown. Note that the speedups reported in [9] were in the $[0.59 \ldots 1.65]$ range for a CUDA-only implementation and $[1.0 \ldots 2.5]$ for a hybrid CPU-GPU solution; these exhibit the same large variability between input graph types that we observed, albeit with lower overall performance.

The speedups reported above are already much higher than the ones reported for GPUs [9]. Moreover, there is potential for even higher speedups, if we assume that the XMT architecture undergoes industry-grade optimizations: if the running time of the hi_pr.MTCU is brought down to match hi_pr.x86, it is reasonable to infer that the execution time of the parallel code will also be reduced, although by a smaller factor. In this case, it is likely that speedups will be accomplished on all inputs, since the upgraded MTCU will ensure that there will never be a slow-down.

5. CONCLUSION

We reviewed an XMT many-core implementation of the Max-Flow algorithm and its evaluation. Although other implementations could not achieve speedups in excess of 2.5x versus a best serial algorithm (**hi_pr**) on current many-cores, we demonstrated a potential for much better performance on XMT. This example provides powerful new evidence that XMT is better suited to handle general-purpose, irregular applications. It provides performance and increased programmer's productivity by directly relying on a simple and well-established algorithmic model coupled with a programmer's workflow.

Voiding PRAM criticism and addressing asymptotic analysis criticism. Criticism of the PRAM model has sometime been confused with criticism of constant factors suppressed by asymptotic analysis. In our opinion the XMT platform and the performance it facilitated have voided much of the criticism on the PRAM model. However, one has to be a bit more careful with understanding the issue of constant factors. In the same way that theoretical papers on serial algorithms and their asymptotic analysis were often followed by separate efforts minimizing constant factors, the current work complements the original theory PRAM papers by reducing them to practice with respect to XMT, accounting for constant factors and concrete speedups. This often amounts to first modifying a published PRAM algorithm to another PRAM algorithm, or using alternative data structures with better constant factors, and only then program it for the XMT platform. The former is where the intellectual merit of this work lies. The good news is that the latter turns out to be a rather simple task.

6. REFERENCES

[1] R. J. Anderson and J. Setubal. On the parallel implementation of goldberg's maximum flow algorithm. In *Proc. 4th ACM SPAA*, 168–177, 1992.

[2] R. J. Anderson and J. Setubal. Goldberg's algorithm for maximum flow in perspective: a computational study. *In D. Johnson and C. McGeoch, eds., Network Flows and Matching: First DIMACS Implementation Challenge*, 1993.

[3] D. Bader and V. Sachdeva. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. In *Proc. of the 18th ISCA International Conference on Parallel and Distributed Computing Systems*, 2005.

[4] G. C. Caragea, F. Keceli, A. Tzannes, and U. Vishkin. General-purpose vs. gpu: Comparison of many-cores on irregular workloads. In *HotPar '10: Proceedings of the 2nd Workshop on Hot Topics in Parallelism*. USENIX,2010.

[5] J. Cheriyan and K. Mehlhorn. An analysis of the highest-level selection rule in the preflow-push max-flow algorithm. *Information Processing Letters*, 69:69–239, 1998.

[6] A. Goldberg. Network optimization library. http://www.avglab.com/andrew/soft.html, 2006.

[7] A. Goldberg and R. Tarjan. A parallel algorithm for finding a blocking flow in an acyclic network. *Information Processing Letters*, 31:265–271, 1989.

[8] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

[9] Z. He and B. Hong. Dynamically tuned push-relabel algorithm for the maximum flow problem on cpu-gpu-hybrid platforms. In *The 24th IEEE International Parallel and Distributed Processing Symposium*, 2010.

[10] M. Hussein, A. Varshney, and L. Davis. On implementing graph cuts on cuda. In *First Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, 2007.

[11] F. Keceli, A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin. Toolchain for programming, simulating and studying the xmt many-core architecture. In *Proc. International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2011.

[12] Y. Shiloach and U. Vishkin. An $O(n^2 \log n)$ parallel max-flow algorithm. *J. Algorithms*, 3(2):128–146, 1982.

[13] V. Vineet and P. Narayanan. Cuda cuts: Fast graph cuts on the gpu. In *Computer Vision and Pattern Recognition Workshops (CVPR), 2008.*

[14] U. Vishkin. A parallel blocking flow algorithm for acyclic networks. *J. Algorithms*, 13(3):489–501, 1992.

[15] U. Vishkin. Using simple abstraction to reinvent computing for parallelism. *Comm. ACM*, 54:75–85, Jan. 2011.

[16] X. Wen and U. Vishkin. Fpga-based prototype of a pram-on-chip processor. In *CF '08: Proc. of the Conference on Computing frontiers*, 2008. ACM.