Hardware and Software Implementations of Prim's Algorithm for Efficient Minimum Spanning Tree Computation

Artur Mariano, Dongwook Lee, Andreas Gerstlauer, Derek Chiou

Hardware and Software Implementations of Prim's Algorithm for Efficient Minimum Spanning Tree Computation

Artur Mariano[1⋆], Dongwook Lee[2], Andreas Gerstlauer[2], and Derek Chiou[2]

[1] Institute for Scientific Computing
Technische Universität Darmstadt
Darmstadt, Germany
[2] Electrical and Computer Engineering
University of Texas
Austin, Texas, USA
artur.mariano@sc.tu-darmstadt.de,
{dongwook.lee@mail,gerstl@ece,derek@ece}.utexas.edu

Abstract. Minimum spanning tree (MST) problems play an important role in many networking applications, such as routing and network planning. In many cases, such as wireless ad-hoc networks, this requires efficient high-performance and low-power implementations that can run at regular intervals in real time on embedded platforms. In this paper, we study custom software and hardware realizations of one common algorithm for MST computations, Prim's algorithm. We specifically investigate a performance-optimized realization of this algorithm on reconfigurable hardware, which is increasingly present in such platforms.
Prim's algorithm is based on graph traversals, which are inherently hard to parallelize. We study two algorithmic variants and compare their performance against implementations on desktop-class and embedded CPUs. Results show that the raw execution time of an optimized implementation of Prim's algorithm on a Spartan-class Xilinx FPGA running at 24 MHz and utilizing less than 2.5% of its logic resources is 20% faster than an embedded ARM9 processor. When scaled to moderate clock frequencies of 150 and 250 MHz in more advanced FPGA technology, speedups of 7x and 12x are possible (at 56% and 94% of the ARM9 clock frequency, respectively).

Keywords: Prim's algorithm, FPGA, Hardware acceleration, MST

1 Introduction

The minimum spanning tree (MST) problem is as an important application within the class of combinatorial optimizations. It has important applications in computer and communication networks, playing a major role in network reliability, classification and routing [1]. In many application domains, such as

⋆ This work was performed while Artur Mariano was at UT Austin.

wireless and mobile ad-hoc networks (WANETS and MANETS), MST solvers have to be run online, demanding efficient, low-power, real-time implementations on embedded platforms.

In this paper, we focus on hardware implementation of one particular, common MST solver: Prim's algorithm [2], which is used in several ad-hoc networks [3] for topology calculation and other maintenance tasks, such as broadcasts, at both initialization and run time. In particular, several maximum broadcast lifetime (MBL) algorithms have been proposed in the past, all as derivatives of Prim's algorithm, running on a general directed graph [4]. In such applications, Prim's algorithm is typically applied to medium to large network models with significant complexity, e.g. in terms of the number of nodes. In order to adapt to changing network conditions, the algorithm has to be executed in a distributed fashion at regular intervals on each node. In mobile and battery-operated nodes, cost, computational power and energy consumption are often critical resources, and high performance and low power realizations are required. For this purpose, platforms increasingly include reconfigurable logic to support hardware acceleration of (dynamically varying) tasks. This motivates an FPGA implementation of Prim's algorithm, where our primary focus is initially on improved real-time performance. To the best of our knowledge, there are currently no other studies of custom hardware realizations of this algorithm.

The rest of the paper is organized as follows: Section 2 provides a brief review of the theory behind Prim's algorithm including related work. Section 3 discusses the FPGA realization of the algorithm, and Sections 4 and 5 present the experimental setup and results. Finally, the paper concludes with a summary and outlook in Section 6.

2 Prim's Algorithm

Prim is a greedy algorithm that solves the MST problem for a connected and weighted undirected graph. A minimum spanning tree is a set of edges that connect every vertex contained in the original graph, such that the total weight of the edges in the tree is minimized.

The algorithm starts at a random node of the graph and, in each iteration, examines all available edges from visited to non-visited nodes in order to choose the one with the lowest cost. The destination of the chosen edge is then added to the visited nodes set and the edge added to the MST. The pseudo-code of the algorithm is presented as Algorithm 1.

2.1 Performance analysis

As with a majority of graph traversals, Prim's algorithm has irregular memory access patterns. In CPUs, this limits cache use and thus overall performance. As such, the algorithm is memory-bound with low computational requirements, and its performance is highly dependent on the organization of memory storage and memory access patterns.

Algorithm 1: Standard Prim's algorithm.

Input: A non-empty connected weighted graph G composed of vertexes V_G and edges E_G, possibly with null weights;
Result: The minimal spanning tree in the $finalPath$ array;
Initialization: $V_T = \{r\}$, where r is a random starting node from V;

```
while V_T ≠ V_G do
    minimum ← ∞;
    for Visited nodes s ∈ V_T do
        for all edges E(s,v) and v ∉ V_T do
            if Weight(E) ≤ minimum then
                minimum ← Weight(E);
                edge ← E;
                newVisited ← v;
    finalPath ← finalPath ∪ {edge};
    V_T ← V_T ∪ {newVisited};
```

Depending on the used data structures, Prim's algorithm can have different asymptotic complexities. For common implementations using an adjacency matrix, Prim's complexity is $\mathcal{O}(V^2)$. For other implementations using adjacency lists with binary or Fibonacci heaps, the complexity reduces down to $\mathcal{O}((V+E)\ log\ V) = \mathcal{O}(E\ log\ V))$ and $\mathcal{O}(E+V\ log\ V)$, respectively. This comes at a higher fixed complexity per step with reduced regularity and exploitable parallelism. Hence, we focus in our work on the most common form using adjacency matrix based realizations.

2.2 Parallelism Analysis

Depending on the implementation of Prim's algorithm, it can exhibit some parallelism. For realizations that use an adjacency matrix to represent the graph, an improved implementation has been reported in [5], which is shown in Algorithm 2. This second version uses a supplemental array d to cache every value if it represents a cheaper solution than the ones seen so far, which allows lookups for minimum paths to be done in parallel.

In [5], the authors have pointed out that the outer while loop in this second implementation is hard to parallelize due to its inherent dependencies. However, the other operations in the body of the loop, namely minimum edge cost lookups (i.e. min-reduction steps) and updates of the candidate set can be processed in parallel over different elements of the supplementary array d. However, min-reduction operations have to take into account that values representing edges $E(s, d)$ in which $s \in V_T$ and $d \in V_T$ can not be considered.

Algorithm 2: Second implementation of Prim's algorithm.

Input: A non-empty connected weighted graph G composed of vertexes V_G and edges E_G, possibly with null weights;
Result: The minimal spanning tree in the d array;
Initialization: $V_T = \{r\}$ and $d[r] = 0$, where r is a random node from V;

```
for v ∈ (V-V_T) do
    if E(r,v) then
        d[v] ← Weight(E);
    else
        d[v] ← ∞
while V_T ≠ V_G do
    Find a vertex u such that:
        d[u] =min{d(v) | v ∈ (V − V_T)}
    V_T ← V_T ∪ {u};
    for all v ∈ (V − V_T) do
        d[v] ←min{d[v], Weight(u, v)};
```

2.3 Related Work

There are a number of implementations of Prim's method on CPUs. Some work has been done on parallel realizations targeting SMP architectures with shared address space, growing multiple trees in parallel and achieving a reported speedup of 2.64x for dense graphs [6]. Additionally, in [7] a distributed memory implementation, which supports adding multiple vertexes per iteration was demonstrated using MPI. Next to CPU implementations, GPUs were also used to compute Prim's algorithm [8]. Such GPU implementations achieve only limited speedups of around 3x, highlighting the difficulties in implementing Prim's algorithm in an efficient and real-time manner. In [8], the authors argued that the difficulty in parallelizing Prim's algorithm is very similar to other SSSP (single source shortest path) problems, like Dijkstra's algorithm.

3 FPGA Implementation

We used high-level synthesis (HLS) to synthesize C code for the different Prim variants down to RTL. We employed Calypto Catapult-C [9] to generate RTL, which was further synthesized using Mentor Precision (for logic synthesis) and Xilinx ISE (for place & route). Table 1 summarizes LUT, CLB, DFF and BRAM utilizations over different graph sizes for both implementations. We have realized the algorithm for graph sizes up to $N = 160$ nodes, where graphs are stored as adjacency matrices with $N \times N$ float values representing edge weights (and with negative values indicating edge absence).

Within Catapult-C, we exploited loop unrolling and chaining only at a coarse granularity, i.e. for the bigger outer loops. This allows a fair comparison with

Table 1. FGPA synthesis results.

Graphs size	40	70	100	130	160
Algorithm 1					
LUTs	1047	1044	1109	1425	1448
CLBs	523	522	554	712	724
DFFs	563	623	628	635	624
BRAMs	6	12	21	33	48
Algorithm 2					
LUTs	1029	1145	1198	1368	1425
CLBs	514	572	599	684	712
DFFs	592	622	635	658	653
BRAMs	7	13	22	34	49

Table 2. Test platform specifications.

Device	CPUs		FPGA
Manufacturer	Intel	ARM	Xilinx
Brand	Pentium M	ARM 9	Spartan
Model	T2080	926EJ-S	3
Max clock	1.73 GHz	266 MHz	400 MHz
Cores	2	1	-
System mem	2 Gbytes	32 Mbytes	1.8 Mbit
L1 Cache	32kB	16kB	-
L2 Cache	4MB	-	-
Pipeline	12 stage	5 stage	-
Year	2006	2001	2003
Launch price	$134	$15.5	$3.5

CPUs, for which we did not realize manually optimized implementations. Exploited optimizations of the outer loops did not provide us with substantial performance gains, while unrolling of loops did increase total area. Area increases by 3 times for a 8x unrolling degree. We were not able to pipeline the loops due to dependencies. The middle inner loop in Algorithm 1 has shown some small performance improvements when unrolled, with no significant difference between unrolling by 2, 4 or 8 times. Overall, the code did not exhibit significant benefits when applying loop optimizations.

4 Experimental Setup

We tested the performance of Prim's algorithm on 3 devices: a desktop-class CPU, an embedded processor and a Xilinx FPGA. Characteristics of tested platforms are summarized in Table 2. We evaluated algorithm performance on all platforms in order to measure possible speedups when moving to the FPGA.

For FPGA prototyping, we utilized a development board that includes a Freescale i.MX21 applications processor (MCU), which communicates with a Xilinx Spartan 3 FPGA over Freescale's proprietary EIM bus. The MCU contains an integrated ARM9 processor running at 266 MHz, an AMBA AHB bus and an EIM module that bridges between the AHB and the EIM bus. The ARM9 runs an embedded Linux distribution with kernel version 2.6.16 on top of which we implemented a testbench that feeds the FPGA with data and reads its output. We utilized both polling and interrupt-based synchronization between the ARM9 and the FPGA. On the FPGA side, we manually integrated Catapult-generated RTL with a bus slave interface connection to the EIM bus, using a custom developed EIM slave RTL IP module to receive and send data from/to the CPU. Designs were synthesized to run at 24 MHz on the FPGA. The bus interface clock, on the other hand, was set to run at 65 MHz.

Measurements of FPGA execution times have been made both including and excluding communication overhead. To obtain total FPGA execution times, we measured the time stamps between sending the first data item and receiving the last result on the CPU side. This includes overhead for OS calls, interrupt handling and EIM bus communication. In addition, we developed a VHDL/Verilog

testbench and performed simulations to determine the raw FPGA computation time without any such communication overhead.

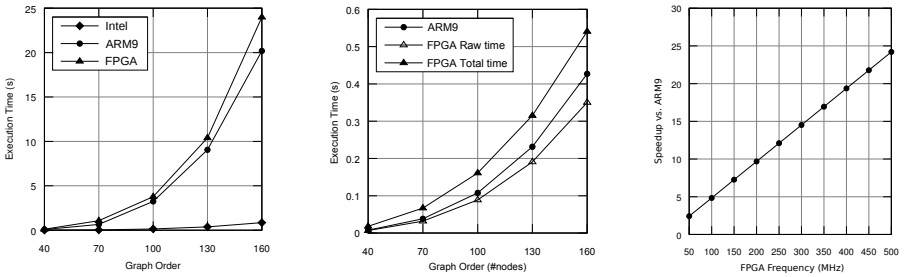
For our experiments, the algorithm ran on all platforms with fully-connected[3] random graphs with orders (sizes) of up to 160 nodes (as determined by FPGA memory limitations). When compiling the code for the CPUs, we did not perform any manual tuning, solely relying on standard compiler optimizations using GCC 4.2.3 and GCC 4.5.2 for ARM9 and Pentium processors, respectively.

5 Results

Figure 1(a) shows total execution times of running Algorithm 1 for graphs with up to 160 nodes on the Intel Pentium, the ARM9 and the FPGA. We report execution times as the median of 5 measurements.

The Intel CPU clearly outperforms other devices. While a Pentium CPU is neither common in embedded domains nor comparable to other platforms in terms of price, we include its results as a baseline for reference.

Compared to the ARM9 processor, execution times on the FPGA are slightly larger across all graph orders. However, in this setup, measured execution times include communication overhead, which may limit overall FPGA performance. On the other hand, computation and communication are overlapped in the FPGA and computational complexities grow with the square of the graph size whereas communication overhead only grows linearly. Coupled with the fact that execution time differences between the ARM and the FPGA increase with growing graph sizes, this indicates that communication latencies are effectively hidden and clock frequency and/or hardware resources are limiting performance.



(a) Algorithm 1 (b) Algorithm 2 (c) Scaled Algorithm 2

Fig. 1. Total runtime and speedup of algorithms for graphs with up to 160 nodes.

FPGA-internal storage sizes limit possible graph sizes to 160 nodes in our case. This limitation could be overcome by taking advantage of matrix symmetries: the same matrix could be represented with half of the data, enabling the

[3] Graphs with $(k-1)$-connectivity with $k =$ order.

study of bigger inputs, e.g. to see if FPGAs could overcome the ARM9. However, as previously mentioned, execution times are computation bound and relative differences of both devices are growing for larger inputs. As such, extrapolating would indicate that the FPGA will not be able to outperform the ARM9 even for larger inputs.

To take advantage of FPGA strengths with increasing benefits for any additional parallelism available in the algorithm [10, 11], we have tested the second implementation of Prim's algorithm on dedicated hardware. Figure 1(b) shows the results of these experiments as measured on the FPGA and on the ARM9, where FPGA performance is reported both with and without communication overhead.

Algorithm 2 clearly performs better on both devices, with speedups of around 37.5x and 68.5x compared to Algorithm 1 for the ARM9 and FPGA, respectively. Even though the FPGA can take more advantage of the second implementation's parallelism than the single-core ARM9, the ARM9 still outperforms the FPGA in total execution time. However, when considering raw execution times without communication overhead, the FPGA performs better.

In addition, overall FPGA performance of designs on our board is limited to a maximum frequency of 24 MHz. To extrapolate possible performance, we scaled raw execution times for a graph with 160 nodes (requiring around 8.8 million clock cycles) to other clock frequencies, as shown in Figure 1(c). Assuming the bus interface is not a limiting factor, running the developed design on an ASIC or better FPGA in a more advanced technology would result in theoretical speedups of 5x and 10x for moderate clock frequencies of 100 MHz and 200 MHz, respectively, as also shown by Figure 1(c). However, this would most likely also come at increased cost, i.e. decreased price/efficiency ratios.

6 Summary and Conclusions

In this paper, we presented an FPGA implementation of Prim's algorithm for minimum spanning tree computation. To the best of our knowledge, this represents the first study of realizing this algorithm on reconfigurable hardware. Prim's algorithm plays a major role in embedded and mobile computing, such as wireless ad hoc networks, where FPGAs may be present to support hardware acceleration of performance-critical tasks. We followed a state-of-the-art C-to-RTL methodology using HLS tools to synthesize a high-level C description of two algorithmic variants down to the FPGA, with high performance being the primary goal. On our Spartan 3 FPGA with 66,560 LUTs, 33,280 CLB slices and 68,027 DFFs, our designs utilizes less than 2.5% of each type of logic resource.

Our results show that, considering total wall-time for any of the tried implementations, our unoptimized FPGA implementation running at a low clock frequency dictated by the bus interface reaches about the same performance as an implementation running on an embedded ARM core. However, in terms of raw computation cycles without any communication or OS overhead, the FPGA design achieves a speedup of $\approx$1.21. Using more advanced FPGA technology,

with a different device, running at a moderate frequency of 150 MHz (55% of the ARM9's 266 MHz frequency), speedups of around 7.5x should be achievable. Compared to gains of 2.5-3x achieved on multi-core CPUs or GPUs, such an FPGA implementation can achieve better performance at lower cost and power consumption.

Prim's algorithm in its default implementations is limited in the available parallelism. In future work, we plan to investigate opportunities for further algorithmic enhancements specifically targeted at FPGA and hardware implementation, e.g. by speculative execution or by sacrificing optimality of results for better performance.

Acknowledgments. Authors want to thank UT Austin|Portugal 2011 for enabling this research collaboration (www.utaustinportugal.org).

References

1. Graham, R.L., Hell, P.: On the history of the minimum spanning tree problem. IEEE Ann. Hist. Comput. **7**(1) (January 1985) 43–57
2. Prim, R.C.: Shortest connection networks and some generalizations. Bell System Technology Journal **36** (1957) 1389–1401
3. Benkic, K., Planinsic, P., Cucej, Z.: Custom wireless sensor network based on zigbee. 49^{th} Elmar, Croatia (September, 2007) 259–262
4. Song, G., Yang, O.: Energy-aware multicasting in wireless ad hoc networks: A survey and discussion. Computer Communications **30**(9) (2007) 2129 – 2148
5. Grama, A., Karypis, G., Kumar, V., Gupta, A.: Introduction to parallel computing: design and analysis of algorithms. 2^{nd} edn. Addison-Wesley (2003)
6. Setia, R., Nedunchezhian, A., Balachandran, S.: A new parallel algorithm for minimum spanning tree problem. International Conference on High Performance Computing (HiPC) (2009) 1–5
7. Gonina, E., Kale, L.: Parallel Prim's Algorithm with a novel extension. PPL Technical Report (October, 2007)
8. Wang, W., Huang, Y., Guo, S.: Design and Implementation of GPU-Based Prims Algorithm. International Journal of Modern Education and Computer Science (IJMECS) **3**(4) (2011) 55
9. Bollaert, T.: Catapult Synthesis: A Practical Introduction to Interactive C Synthesis High-Level Synthesis. In Coussy, P., Morawiec, A., eds.: High-Level Synthesis. Springer Ned., Dordrecht (2008) 29–52
10. Singleterry, R., Sobieszczanski-Sobieski, J., Brown, S.: Field-Programmable Gate Array Computer in Structural Analysis: An Initial Exploration. 43^{rd} AIAA/AMSE/ASCE/AHS Structures, Structural Dynamics, and Materials Conference (April, 2002) 1–5
11. Ornl, W.Y., Strenski, D., Maltby, J.: Performance Evaluation of FPGA-Based Biological Applications Olaf. (Cray Users Group 2007, Seattle, USA, 2007.)