Exploring single source shortest path parallelization on shared memory accelerators

Author(s):
Palossi, Daniele; Marongiu, Andrea

Publication date:
2016

Permanent link:
https://doi.org/10.3929/ethz-b-000117736

Originally published in:
https://doi.org/10.1145/2906363.2915925

Exploring Single Source Shortest Path Parallelization on Shared Memory Accelerators

Daniele Palossi
IIS, ETH Zürich
Zürich, Switzerland
dpalossi@iis.ee.ethz.ch

Andrea Marongiu
IIS, ETH Zürich
Zürich, Switzerland
amarongiu@iis.ee.ethz.ch

ABSTRACT

Single Source Shortest Path (SSSP) algorithms are widely used in embedded systems for several applications. The emerging trend towards the adoption of heterogeneous designs in embedded devices, where low-power parallel accelerators are coupled to the main processor, opens new opportunities to deliver superior performance/watt, but calls for efficient parallel SSSP implementation. In this work we provide a detailed exploration of the Δ-stepping algorithm performance on a representative heterogeneous embedded system, TI Keystone II, considering the impact of several parallelization parameters (threading, load balancing, synchronization).

Keywords

Parallel Graph Exploration; Heterogeneous Acceleration

1. INTRODUCTION

Single Source Shortest Path (SSSP) algorithms are widely used in embedded systems for several applications. The problem can be stated as follows: in a directed graph with non-negative weights, find the minimal cost path from one chosen node to all other nodes. In wireless sensor networks the routing problem is often modeled as a shortest path problem, where the cost functions to be minimized can be numerous (e.g. load-balancing, energy, etc.). The well known Dijkstra algorithm [1] has been used to maximize the energy-efficiency for multicast communication [7], searching for mininum-energy paths with maximum geographical advance towards desired destinations. In body sensor networks, where bio-medical sensor nodes can be implanted in tissues, SSSP has been used to solve a least total-route-temperature problem [8]. Edge costs are dynamically updated, accordingly with the current temperature of each node, and the route from source to destination is selected among all possible routes. SSSP is also widely used for trajectory planning for autonomous vehicles [5], where computation- and energy-constrained on-board embedded devices can make fast obstacle avoidance quite challenging.

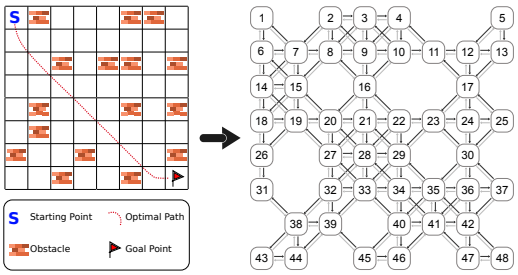

Figure 1: A 8x8 sample of Game-Map graph.

In general, a common requirement in all these applications is to rely on efficient solutions able to exploit the limited performance/watt delivered by embedded systems, while respecting the given constraints (e.g. QoS, deadlines, etc.). The emerging trend towards the adoption of low-power parallel accelerators in embedded systems[1] opens new opportunities to deliver superior performance/watt, but calls for efficient parallel implementation of the above described problems.

In this work we address the SSSP problem through the exploration of the Δ-stepping algorithm [6]. The Δ-stepping algorithm divides Dijkstra's algorithm into a number of phases, such that each phase can be executed in parallel. This algorithm has been successfully implemented and explored for distributed memory architectures [2]. The investigation presented here focuses on the *Game-Map* class of graphs (see Figure 1), that is commonly used to represent the physical environment for trajectory planning of autonomous vehicles [3, 5].

The main contribution of our work is is a detailed performance analysis of Δ-stepping on a representative heterogeneous embedded system, Texas Instrument's Keystone II[2]. We consider the impact of several parallelization parameters (threading, load balancing, synchronization) considering an OpenMP implementation of the target algorithm.

2. Δ-STEPPING ALGORITHM

The SSSP problem with non-negative weights can be stated as follows: given a weighted graph $G = (V, E, c)$, where V is the set of *vertices* or *nodes*, E the set of *edges* (i.e. pairs of

[1] http://www.pulp-platform.org, http://www.kalrayinc.com
[2] http://www.ti.com/product/66AK2H12

Final published version in ACM Digital Library available at https://dl.acm.org/citation.cfm?id=2915925.
DOI: http://dx.doi.org/10.1145/2906363.2915925

nodes) and c the *cost* ($c : E \rightarrow \mathbb{R}_+$), find a minimal weight path from one chosen node $s \in V$, called the *source node*, to all other nodes in V. We say that the nodes $v, w \in V$ are *neighbours* if $(v, w) \in E$, i.e., if there exists an edge between them.

The Δ-Stepping algorithm goes forward in the exploration of the graph dividing the problem-space in *boundaries*. A boundary represents the part of the graph that is reachable from the source at a cost multiple of Δ. The computation within each boundary is suitable for parallelization.

The main data structure to keep track of these boundaries is the *bucket*, that contains all the nodes to be explored in the corresponding step. This data structure has to be checked at every step in order to identify all the nodes to be explored in the current iteration. When a node is selected for exploration (it is said to be the *reference* node) all his neighbors are evaluated. If a smaller cost is found to reach such neighbor through the current reference node the neighbor's cost is updated using the *relax* function.

The relax function is also in charge of distributing the neighbors in the appropriate bucket to be evaluated, as reference node, in a future iteration (depending on the cost and Δ). Updating the cost of a node can happen concurrently when multiple reference nodes are being explored in parallel, and this requires a proper synchronization mechanism. The original algorithm uses also the Δ parameter, in a preprocessing phase, to divide all the edges in two groups: *heavy* and *light* edges, based on whether the cost of that edge is smaller or larger than Δ.

In this work we use a regular graph structure: each node has eight outgoing edges that represent allowed movements from the node. We associate a cost of 10 to all perpendicular movements and a cost of 15 to all diagonal movements, representing the physical topology of the map (i.e. euclidean distance). For this reason we can avoid the typical preprocessing stage to identify if an edge is light or heavy during the exploration.

The specific shared memory implementation considered here [4] implements bucket checking by scanning a monodimensional array (representing the graph) of dimension V. Choosing a linear, statically-sized data structure allows for a lightweight synchronization mechanism (compared to dynamic, pointer-based data structures [6]) and for easier parallelization (loop-level, as opposed to costlier tasking abstractions). On the downside, a full graph exploration is implied at every iteration, while for the targeted class of graphs only the neighbors of the nodes belonging to the *boundary* (i.e. the explored part of the graph, shaded in yellow in Figure 2) should be checked in the current iteration.

3. RESULTS AND DISCUSSION

As a target architecture for our exploration we use the TI Keystone II. As shown in the simplified block diagram in Fig. 3, this platform captures a widely used template where a general-purpose *host* processor is coupled to a parallel accelerator. Here, the *host* system consists of a ARM A15 quad core processor, while the parallel accelerator is made of a cluster of 8 C66x VLIW DSPs. The two systems leverage private memory hierarchy but share the main DRAM controller and memory. Table 1 provides additional architectural details.

For our experiments we consider different graph dimensions, identified in the plot labels as the size of the edge of

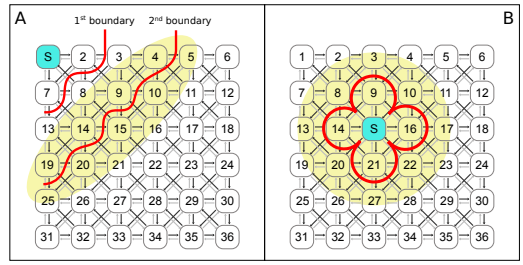

Figure 2: Δ-Stepping *boundaries* (red lines), for a given source node (S) and the explored part of the graph (area in yellow).

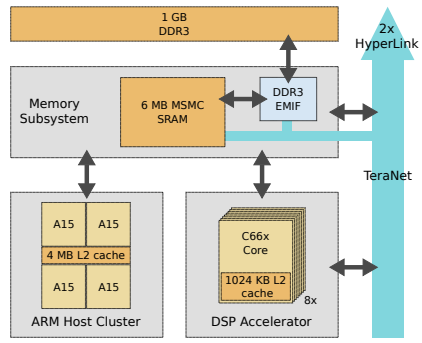

Figure 3: TI Keystone II block diagram.

a squared map (thus a graph labeled "1000" has 1 M nodes). As a main performance metric we report speedup of the parallel implementation compared to the sequential one. Execution cycles for each experiment are the average of several runs.

3.1 Host: Cortex A15

The first part of our exploration focuses on the *host* processor. The ARM A15 is a quad-core SMP, capable of multithreading, thus we explore the parallelization scalability with the number of threads. Initially we consider the most straightforward static loop scheduling strategy (`#pragma omp for schedule(static)`). Here the whole graph is evenly divided in N contiguous stripes (N being the number of threads).

Fig. 4.A shows the results for this experiment, considering four different graph sizes. The plot shows that the parallelization scheme works best for graph sizes up to 1000×1000.

It is also evident that hyperthreading does not help, as considering more threads than processors does not increase the speedup. This indicates that the computation is never memory bound. As a consequence, for graph sizes until 1000×1000 the overheads are dominating when hyperthreading is used, which leads to slowdowns. For larger graphs these overheads are amortized, but no additional speedups are achieved. The parallelization scheme proposed in [4] always checks every node in the graph. This exposes more parallelism and simplifies synchronization, as it avoids costly pointer-based data structures (e.g., lists, such as in [6]). However, it potentially introduces load balancing issues, as the actual computation is applied only to neighboring nodes to the reference one. All the remaining nodes are quickly pruned out, thus making execution time among different

	ARM Cortex A15	DSP C66x
# Cores	4	8 (VLIW)
Core Frequency	1.4 GHz	1.2 GHz
L2 Cache	4096 KB (cluster)	1024 KB (core)
System DDR3 Memory	1 GB	

Table 1: Devices for the experiments.

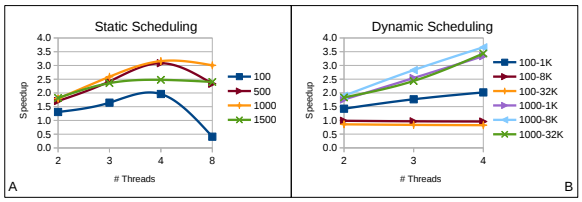

Figure 4: Host scaling with static (A) and dynamic (B) scheduling.

threads non uniform. To study the impact of load imbalance we use a second scheduling approach, namely a dynamic one with variable iteration chunk size (`#pragma omp for schedule(dynamic, C)`, where $C = 2^K$ and $K \in [0, 7]$).

Fig. 4.B shows the effect of dynamic scheduling for two graph sizes and for several chunk sizes. For very small graph sizes a scheduling chunk of 1K iterations represents the best trade-off between load balancing and overhead, while the same is true for chunks of 8K iterations for larger graphs. Below these values the overheads are predominant, and above these values the capability of better balancing the workloads is lost as the scheduling unit is too coarse. The same effects can be better appreciated from Fig. 5, which shows an exploration of dynamic chunking as compared to static scheduling in the configuration that exhibits the best speedup in Fig. 4.A: 1000×1000 graph size, 4 threads.

In the following we will focus on chunk sizes of 8 K for dynamic scheduling.

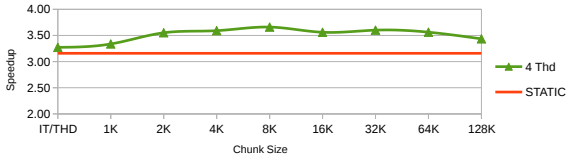

Figure 5: Dynamic chunk size exploration for 1000x1000 graph.

3.2 Accelerator: DSP C66x

The second part of our evaluation focuses on parallelization efficiency on the accelerator (8 DSPs). We study here the scalability of the parallelization with the number of threads, focusing on the configuration with the best speedup observed in the previous section (1000×1000 graph size, dynamic scheduling with chunk size 8K).

As discussed previously, updating the cost of a path that goes through a particular reference node requires mutual exclusion due to race conditions. The availability of efficient hardware primitives for synchronization is very important to minimize the effect of serialization implied by such constructs. On the ARM we leverage support for hardware-assisted *compare-and-swap* (CAS), which enables near-to-ideal speedups. Simplified pseudo-code for the usage of the atomic CAS is given in Fig. 6.A.

On TI DSPs such a primitive is not available, so we explore the effect of three alternatives (pseudo-code in also given in Fig. 6):

1. **Critical section**: `#pragma omp critical` to protect a critical section with coarse-grained locking, Fig. 6.B;
2. **Lock**: A fine-grained locking scheme, Fig. 6.C;
3. **Race condition**: The update is not protected. This configuration is aimed at estimating what could be achieved with CAS or other HW support, Fig. 6.B without `#pragma omp critical`.

```
while f is false
      cost_old ← cost[n]
      if cost_old > cost_new
            cost[n] ← cost_new
            f ← __sync_bool_compare_and_swap(cost[n], cost_old, cost_new)
      else
            break
      end if
end while
```
A

```
#pragma omp critical
if cost[n] > cost_new
      cost[n] ← cost_new
end if
```
B

```
omp_set_lock(locks+n)
if cost[n] > cost_new
      cost[n] ← cost_new
end if
omp_unset_lock(locks+n)
```
C

Figure 6: Pseudo-code for the synchronization mechanisms.

Fig. 7 shows the parallelization speedup scalability using two different baselines. The top plots (A, B, C) show speedups versus the sequential algorithms, while the bottom plots (D, E, F) show speedups versus the parallel algorithm executed with only one thread. The comparison between the two sets of plots shows that the cost for software-managed synchronization significantly limits the parallelization potential.

3.3 Obstacles Evaluation

As a final experiment we evaluate the impact of obstacles in the graph nodes on parallelization scalability. As shown in Fig. 1, the presence of an obstacle implies the deletion of the occupied node from the graph, thus, the more obstacles the smaller the graph. The percentages of obstacles taken into account are: 0% (i.e no obstacles), 25% and 50% of the total graph nodes.

Fig. 8 shows results for both the *host* and the accelerator, considering the best corresponding configurations (4 threads for the *host* and 8 for the accelerator, dynamic scheduling with a chunk size of 8 K). The main difference among the three cases is represented by the synchronization mechanism adopted to protect the update of the cost array. Where available (i.e. on the ARM A-15) we take advantage of the efficient CAS operation (Fig. 8.A); on the accelerator we use the most fine-grained primitive available: OpenMP locks (Fig. 8.B). For the sake of completeness we also show the behavior of unprotected updates (Fig. 8.C).

The plots show that when efficient synchronization is supported in hardware the introduction of obstacles in the graph has a positive effect on overall speedup. This is due to the fact that a smaller number of nodes translates in a larger portion of the node check phase being useful parallel workload (i.e., some of the non-necessary checks introduced to eliminate the costly pointer-based data structures are eliminated). The same conclusion does not apply to *locks*, as

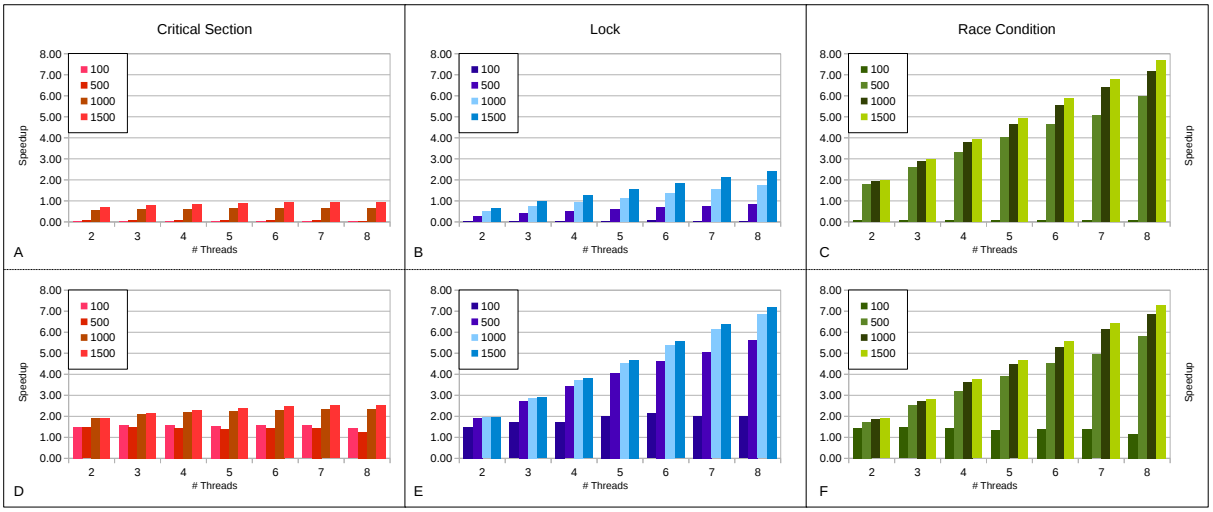

Figure 7: DSP scalability with 3 methods of synchronization. In A, B, C the speedup is vs. the sequential algorithm, in D, E, F the speedup is vs. 1 thread parallel algorithm.

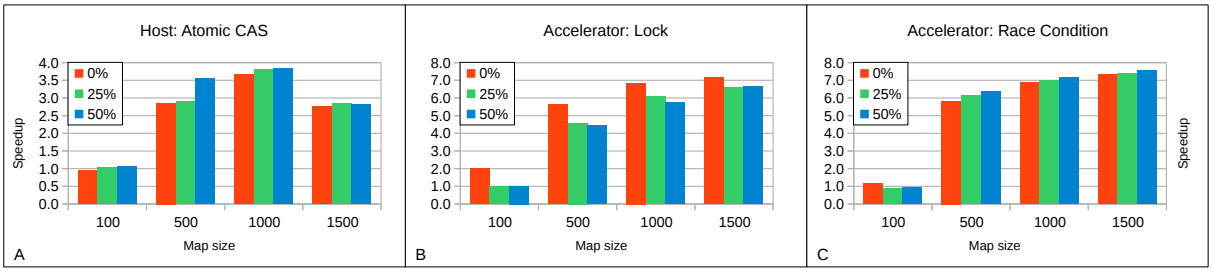

Figure 8: Impact of obstacles on parallelization scalability.

the predominant effect here is the reduction of the available parallelization due to the serialization of the critical regions. Finally, it is important to remark that all the conclusions also apply to very small graph sizes (from 500 and above), which basically covers all the range that is representative of real-world embedded applications.

4. CONCLUSIONS AND FUTURE WORK

In this work we have presented a detailed exploration of several parallelization parameters for the Δ-Stepping algorithm on a representative heterogeneous embedded device: the TI Keystone II, where a quad-core ARM A15 *host* processor is coupled to 8 VLIW DSPs. Experiments performed on game-map class of graphs highlight the beneficial exploitation of fine-grain synchronization mechanisms and the effectiveness of dynamically scheduling loop iterations in chunks of medium granularity. The analysis also highlights the limitations on the maximum parallelism achievable on *game-map* graphs. This opens the way to algorithmic enhancements to reduce dynamically the area of the graph evaluated during bucket checking.

5. ACKNOWLEDGMENTS

This work has been supported by the EU H2020 project HERCULES (688860).

6. REFERENCES

[1] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[2] N. Edmonds, J. Willcock, and A. Lumsdaine. Expressing graph algorithms using generalized active messages. *SIGPLAN Not.*, 48(8):289–290, Feb. 2013.

[3] S. Hrabar. Reactive obstacle avoidance for rotorcraft uavs. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 4967–4974, Sept 2011.

[4] M. Kranjčević, D. Palossi, and S. Pintarelli. Parallel delta-stepping algorithm for shared memory architectures, arXiv:1604.02113v1 [cs.DC].

[5] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige. The office marathon: Robust navigation in an indoor office environment. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 300–307, May 2010.

[6] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114 – 152, 2003. 1998 European Symposium on Algorithms.

[7] B. Musznicki, M. Tomczak, and P. Zwierzykowski. Dijkstra-based localized multicast routing in wireless sensor networks. In *Communication Systems, Networks Digital Signal Processing (CSNDSP), 2012 8th International Symposium on*, pages 1–6, July 2012.

[8] D. Takahashi, Y. Xiao, F. Hu, J. Chen, and Y. Sun. Temperature-aware routing for telemedicine applications in embedded biomedical sensor networks. *EURASIP J. Wirel. Commun. Netw.*, 2008:26:1–26:26, Jan. 2008.