PARALLEL Δ-STEPPING ALGORITHM FOR SHARED MEMORY ARCHITECTURES

M. Kranjčević, D. Palossi, S. Pintarelli

E-mail: marija.kranjcevic@inf.ethz.ch, daniele.palossi@iis.ee.ethz.ch, simon.pintarelli@sam.math.ethz.ch

ABSTRACT

We present a shared memory implementation of a parallel algorithm, called Δ-stepping, for solving the single source shortest path problem for directed and undirected graphs. In order to reduce synchronization costs we make some deviations from the algorithm and discuss the consequences. We study the behaviour of our implementation on small-world and scale-free graphs, and graphs arising from game maps. We collect performance data on multi-core CPUs and Intel Xeon Phi. When run in sequential mode, our implementation outperforms the implementation of Dijkstra's algorithm from Boost Graph Library[1] *on graphs with a small diameter. Both on the CPU and the co-processor we achieve an overall performance of at least* 50% *parallel efficiency.*

1 INTRODUCTION

We consider the single source shortest path (SSSP) problem: in a directed graph with non-negative weights, find the minimal cost path from one chosen node to all other nodes.

Motivation. The single short shortest path problem is ubiquitous in many applications related to networks and graph theory, for example in robotics [1], combinatorial optimization or the analysis of complex networks.

Related work. One of the best sequential algorithms for the SSSP problem is Dijkstra's algorithm [2]. The Δ-stepping algorithm, proposed in [3], divides Dijkstra's algorithm into a number of phases, such that each phase can be executed in parallel. This algorithm has been successfully implemented for distributed memory architectures [4, 5], but, to our knowledge, there exists only one previous shared memory implementation, written specifically for the multithreaded parallel computer Cray MTA-2 [6].

Contribution. We present a shared memory implementation of the Δ-stepping algorithm written in C/C++ and parallelized with OpenMP. We test our implementation on desktop computers, on a single node of the Euler cluster of ETH Zürich and on a Xeon Phi co-processor. We outperform Boost Dijkstra on a single core for graphs with a low diameter, and we obtain comparable execution times for large diameter lattice graphs with a moderate number of vertices. The parallel efficiency on the CPU and Xeon Phi is 50% or more for the two graph classes.

[1] Boost Graph Library, http://www.boost.org/libs/graph/

2 Δ-STEPPING ALGORITHM

In this section we will more formally define the SSSP problem and explain the Δ-stepping algorithm in more detail.

SSSP problem. The single source shortest path problem with non-negative weights can be stated as follows: given a weighted graph $G = (V, E, c)$, where

V ... set of *vertices* or *nodes*,
E ... set of *edges*, i.e., ordered pairs of nodes,
c ... *cost* or *weight function*, $c : E \to \mathbb{N}$,

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.

Δ-stepping algorithm. The pseudocodes of the Δ-stepping algorithm and its auxiliary *relax* function are given in Alg. 1 and 2, respectively. Both are obtained from [3].

Input. The input of the Δ-stepping algorithm is a graph, given by its vertices V, edges E, and the cost function c, a source node s and an additional parameter $\Delta > 0$.

Buckets. The algorithm introduces the concept of *buckets*, denoted for example by $B = (B_1, B_2, \dots)$, where each bucket B_i is a collection of vertices that are scheduled to be processed in one iteration. The distribution of vertices into the appropriate buckets is determined by the parameter Δ. More precisely, for a vertex $v \in V$, if $tent[v]$ denotes the best known cost with which v can be reached from s, then v belongs to B_i, where $i = \lfloor tent[v]/\Delta \rfloor$.

Relax function. For a vertex w and a newfound cost d, the relax function checks if d is lower than the previously best known cost $tent[w]$ (Alg. 2, line 2). If so, the cost is updated (Alg. 2, line 3), the vertex w taken out of the bucket it was previously in and inserted into the appropriate bucket according to the newfound cost (Alg. 2, line 4-5).

Preprocessing. In a preprocessing stage of the Δ-stepping algorithm, the parameter Δ is used to divide all the outgoing edges of each vertex into two categories, called *light* and *heavy* edges, based on whether the cost of that edge is smaller or larger than Δ (Alg. 1, lines 3-5). The *tentative cost* from s to all nodes $v \in V$ is initially set to $tent[v] = \infty$ (Alg. 1, line 6), after which the relax function is called on

the source node s and cost 0 (Alg. 1, line 8), which results in $tent[s] = 0$ and the insertion of s into the bucket B_0.

Main loop. The algorithm explores the buckets B_i successively, starting with $i = 0$ (Alg. 1, lines 9,21). In each iteration i, it first explores all the light edges from the vertices that are currently in B_i. Since this exploration may result in reinsertions into the bucket that is being explored, it is done in a loop (Alg. 1, lines 13-18). It also requires two auxiliary sets: the *request set* Req which stores pairs of vertices that can be reached by those light edges and the corresponding cost (Alg. 1, line 14), and the set S which collects all the vertices from B_i that were just explored, so that their light edges are not explored again in the next iteration of the inner loop (Alg. 1, lines 15-16). At the end of each iteration of the inner loop, the relax function is called on the pairs from the request set (Alg. 1, line 17). After the bucket B_i has been emptied, it remains to explore the heavy edges of the vertices stored in S, i.e., all the vertices that were at some point in B_i (Alg. 1, lines 19-20).

Algorithm 1 Pseudocode of the Δ-stepping algorithm.

```
1  function Δ-Stepping(V,E,c,s,Δ):
2
3    for each vertex v in V:
4    | heavy[v] ← {(v,w) ∈ E : c(v,w) > Δ}
5    | light[v] ← {(v,w) ∈ E : c(v,w) <= Δ}
6    | tent[v] ← ∞
7    end for
8    relax(s,0)
9    i ← 0
10
11   while B ≠ ∅:
12   | S ← ∅
13   | while B[i] ≠ ∅:
14   | | Req ← {(w,tent(v)+c(v,w)) : v ∈ B[i] and
     | |                              (v,w) ∈ light[v]}
15   | | S ← S ∪ B[i]
16   | | B[i] ← ∅
17   | | for each (w,d) ∈ Req: relax(w,d)
18   | end while
19   | Req ← {(w,tent(v)+c(v,w)) : v ∈ S and
     |                             (v,w) ∈ heavy[v]}
20   | for each (w,d) ∈ Req: relax(w,d)
21   | i ← i+1
22   end while
23   return tent[]
24 end function
```

Algorithm 2 Pseudocode of the auxiliary relax function.

```
1  function relax(w,d):
2  | if d<tent[w]
3  | | tent[w] ← d
4  | | B[⌊tent[w]/Δ⌋] ← B[⌊tent[w]/Δ⌋] \ {w}
5  | | B[⌊d/Δ⌋] ← B[⌊d/Δ⌋] ∪ {w}
6  | end if
7  end function
```

Output. When there are no more elements to be explored, the algorithm returns the array $tent$ (Alg. 1, line 23), where $tent[v]$ contains the minimal cost to get from s to $v \in V$.

Furthermore, each time we have explored the edges of a vertex v and found a better cost to one its neighbours w, along with updating the tentative cost array (Alg. 2, line 3), we also note down the vertex v as the *predecessor* of w. Thus, we are also able to reconstruct the minimal cost path.

3 Implementation and Parallelization

In this section we will describe the main features of our implementation and the optimizations we employed.

In order to keep the parallel region overhead as small as possible, we decided to implement one large parallel region containing the main loop (Alg. 1, lines 11-22), and then endeavoured to reduce the number of thread synchronizations as much as possible.

Bucket structure. To prevent the need for any thread synchronization during the insertions and deletions of the elements in the buckets (e.g. using a shared list), we implemented the bucket structure as an array of fixed size $|V|$. If we denote that array by B, then for each node $v \in V$, $B[v]$ stores the index of the bucket that vertex is currently in, or -1 if that vertex is not in any of the buckets. Due to the fact that each element of the bucket array can store only one value at a time, there is no longer any need for the removal procedure in the relax function (Alg. 2, line 4). Moreover, the insertions into this bucket structure (Alg. 2, line 5) require no synchronization among threads, since in the worst case more than one thread will try to write to the same memory location with the same value. Therefore, even though this approach forces us to scan the entire bucket array B at the beginning of each iteration of the inner loop in order to find all the nodes belonging to the bucket that is currently being explored (Alg. 1, line 13), it still outperforms other approaches because they all require some sort of synchronization for updating the bucket structure.

Updating the cost. We update the tentative cost array $tent$ with an auxiliary `CAS` function, shown in Alg. 3, which we base on the x86 *__sync_bool_compare_and_swap* atomic operation. In this way, a thread will write its cost c in line 6 only if the value stored in memory, $tent[w]$, is still the same value that thread has compared its cost c with in line 5, and it will not exit the loop (lines 3-9) until it has either stored its cost in the $tent$ array, or the proposed cost is no longer smaller than the one already stored in memory.

Algorithm 3 Compare-and-swap auxiliary function.

```
1  function CAS(w, c):
2    s ← false
3    while s is false:
4    | old_c ← tent[w]
5    | if c < old_c:
6    | | s ← __sync_bool_compare_and_swap(tent[w], old_c, c)
7    | else break
8    | end if
9    end while
10 end function
```

Data packing. As shown in Sec. 2, in addition to updating the tentative cost array, we also have to keep track of

the predecessor of each element so that we can reconstruct the minimal cost path. To be able to do this, we have to make sure that, when a thread is updating the predecessor id of a vertex w in memory, while that thread is reading the value $tent[w]$, comparing it with its cost and writing the predecessor to memory, no other thread can change either of those variables. For this, however, atomic operations are not enough.We have to use either a critical region, which serializes a part of the parallel execution, or an additional lock array with one lock for each element in the graph, which results in increased footprint in shared memory and additional read/write operations to manipulate the locks. To avoid these solutions, we decided to package the two pieces of information into one 64-bit element. The 32 right-most bits are used to store the vertex, or *vertex id*, while the remaining bits store the corresponding cost. Thus, we are still able to compare costs, since only the left-most bits affect this comparison, which means that we can again use the `CAS` function from the previous subsection (Alg. 3). We manipulate this packed data with fast bitwise operations.

Deviations from the algorithm. We decided to keep the auxiliary sets S and Req private to each thread, so that they can proceed independently until the end of the current iteration of the main loop. This may generate additional work, as more than one thread may come upon the same element belonging to the current bucket, put it in its private S and explore its edges, so it is not strictly in accordance with the Δ-stepping algorithm as presented in [3]. However, despite the additional work, keeping the auxiliary set S private still results in a better performance.

As a second deviation from the original Δ-stepping algorithm [3], we evaluate and update the tentative cost array $tent$ during the request set computation stage instead of in the relax function, which reduces the number of elements in the request set array Req. In fact, we do not need to store the pair (w, d) any longer, but only the id of the neighbouring node w, and only if the condition $d < tent[w]$ holds, which further reduces of the number of elements in Req.

Code reorganization. As the last step, we reorganized the code in order to reduce the number of memory accesses and avoid undesired cache effects. When computing the request set (Alg. 1 lines 14,19) in a manner described in the previous subsection, each thread fetches the cost related to its current *reference node*, and for each of its neighbours, adds the cost of the edge to that neighbour and checks if the resulting value is better than the previously best known cost to that neighbour. However, cache line ping-pong effect can occur due to false sharing, since the update of some tentative cost by one thread can invalidate the entire cache line used by some other thread trying to fetch the tentative cost to its reference node. That is why we decided to split that procedure into two steps, decoupling reads and writes to the $tent$ array. First, each thread pre-fetches the tentative costs to all the vertices it will explore in that phase and stores them in a private temporary array. Then, it uses this private array to compute the new costs to all the (light or heavy) neighbouring vertices and update them if necessary, using the `CAS` function from Alg. 3.

Load balancing. When iterating over the buckets in order to distribute the vertices belonging to the current bucket among the threads, we used OpenMP *#pragma omp for* with the default *static* scheduling. This leads to a load distribution that is, in general, not perfect, because it greatly depend on the properties of the graph. However, even though it can be suboptimal for some classes of graphs, due to the low computational intensity of the job of each thread, our approach performs much better than OpenMP *dynamic* scheduling, *tasks*, or manual redistribution of work.

Parallel preprocessing. In addition to parallelizing the main loop of the algorithm, we also divided the edges into light and heavy sets in parallel, using *#pragma omp for schedule(static)* to distribute the nodes of the graph among all available threads, so that each thread divides the outgoing edges of the vertices it has been given.

4 Experimental Results

In this section we will analyze the performance and scalability of our implementation of the Δ-stepping algorithm.

Experimental setup. The devices we used to test our code are reported in Table 1, as well as the corresponding compilers, used in all cases with the flags `-O3 -fopenmp`.

Intel Architecture	No. of Cores	Cache Size	Compiler
Xeon E5 2680v3	2×12 @ 2.6 GHz	30 MB	icc 15.0.0
i7-4790	4 @ 3.6 GHz	8192 KB	gcc 4.4.7
Xeon Phi 7120	60 @ 1.238 GHz	30.5 MB	icc 15.0.0

Table 1: Devices and compilers used in our experiments.

The code was executed in native mode on the Xeon Phi, thus avoiding the communication overhead due to memory transfer. We considered several classes of graphs: *small-world* graphs, *scale-free* graphs and *game maps*. For small-world and scale-free graphs we assigned integer edge weights from the discrete uniform distribution $\mathcal{U}(1, 20)$, and for game maps the weight 14 for diagonal and 10 for horizontal and vertical movement. The speedup for a particular Δ is computed with respect to the execution for the same Δ on a single core. For all the configurations we tested, we first compared that run on a single core with Dijkstra's algorithm implemented in the Boost Graph Library, version 1.59.0, which has complexity $\mathcal{O}(|V| \ln |V| + |E|)$.

Small-world graphs. We say that an undirected graph has *small-world properties*, or that it is a *small-world graph*, if most nodes are not neighbours, but the average number of steps needed to connect any two nodes grows like $\sim \ln |V|$.

The most common way to generate such graphs is the *Watts-Strogatz model*: first construct a regular ring lattice by connecting each node with its k closest neighbours; then, for each edge in that graph, with probability $p \in (0, 1]$, rewire one of its endpoints to a random node in the graph, such that it does not result in a duplicate edge. This generation can be done using Boost Graph Library.

The timings reported in Table 2 and Table 3 for small-world graphs with $p \in \{10^{-4}, 10^{-2}\}$, $k \in \{60, 100, 150\}$ and $|V| \in \{5 \times 10^5, 10^6, 2 \times 10^6, 6 \times 10^6\}$ clearly show that our implementation is efficient. Although $\Delta = 10$ involves additional work compared to Dijkstra's algorithm, our implementation is at least 2x faster on Xeon Phi and 2x-100x faster on Xeon E2680v3 than the implementation of Dijkstra's algorithm from the Boost Graph Library.

Type	p	k	No. of vertices 1M	2M	6M
Δ-stepping	$1 \cdot 10^{-4}$	60	852	1,770	5,445
Boost Dijkstra	$1 \cdot 10^{-4}$	60	2,423	5,180	16,520
Δ-stepping	$1 \cdot 10^{-4}$	150	1,402	2,849	10,421
Boost Dijkstra	$1 \cdot 10^{-4}$	150	5,860	11,724	$7 \cdot 10^5$
Δ-stepping	$1 \cdot 10^{-2}$	60	922	2,026	7,029
Boost Dijkstra	$1 \cdot 10^{-2}$	60	2,984	6,172	19,080
Δ-stepping	$1 \cdot 10^{-2}$	150	1,887	4,293	16,852
Boost Dijkstra	$1 \cdot 10^{-2}$	150	6,524	13,312	$2 \cdot 10^6$

Table 2: Timings in ms on a single core of Xeon E2680v3. Δ-stepping was run with $\Delta = 10$.

Type	p	k	No. of vertices 0.5M	1M	2M
Δ-stepping	$1 \cdot 10^{-4}$	60	4,117	8,739	16,902
Boost Dijkstra	$1 \cdot 10^{-4}$	60	9,798	19,144	39,911

Table 3: Timings in ms on a single core of Xeon Phi. The Δ-stepping algorithm was run with $\Delta = 10$. In our case a single core on Xeon Phi is 8-10x slower than on E2680v3.

Delta. In general, the performance of Δ-stepping crucially depends on the choice of the parameter Δ. Fig. 1 shows timings depending on Δ for small-world graphs with size $|V| = 10^6$, $k = 60$ and different rewiring probabilities p. On 24 cores of Xeon E2680v3, for $p = 10^{-2}$, $\Delta = 1$ performs best, for $p = 10^{-4}$ the best is $\Delta = 2$ and for $p = 10^{-5}$, $\Delta = 3$ is optimal. In the extreme case of $p = 0$, the runtime monotonically decreases with increasing Δ.

We have also found that in order to be able to exploit the additional parallelism that can be gained by using $\Delta > 1$ there must not be an *omp barrier* inside the light relaxation phase. During the relaxation of light edges, vertices might be inserted into the current bucket in locations belonging to another thread. Therefore, the parallelization strategy described in Sec. 3 forces us to synchronize the threads before entering the while loop in Alg. 1, lines 13-18. However, by introducing a thread private bucket array in the inner loop, this synchronization can be removed and traded for additional work, since now it is no longer guaranteed that a particular vertex is inserted into the current bucket by only one thread. If a sufficient number of shortcuts to random locations is present, e.g. for $p > 10^{-3}$, there is already enough parallelism available due to the inherent graph structure and we find that $\Delta = 1$ is the best choice.

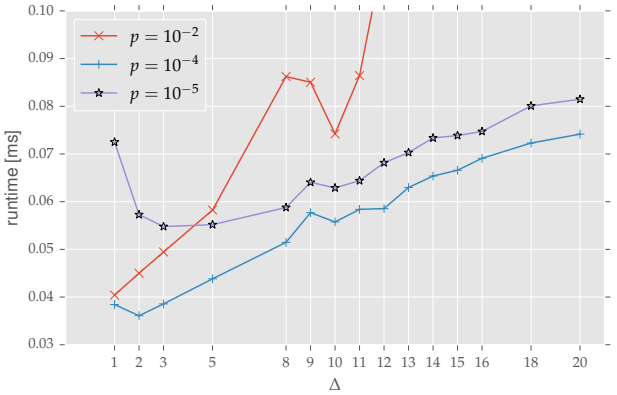

Fig. 1: Runtime with respect to Δ for a small-world graph with 10^6 vertices and k=60. Experiments were carried out using 24 cores on Xeon E2680v3. The 5, 95-percentiles deviate less than 1% of the mean (100 repetitions).

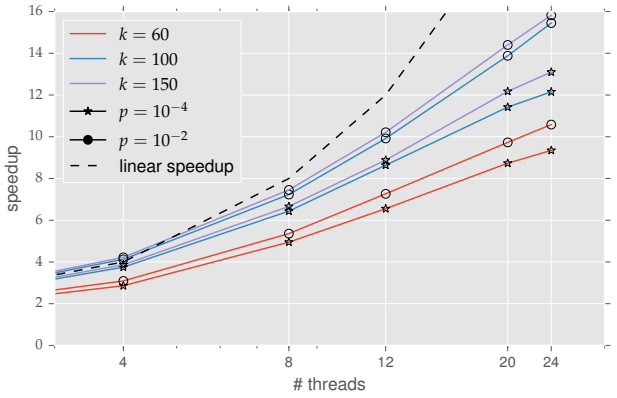

Fig. 2: Xeon: small-world graphs with $6 \cdot 10^6$ vertices, generated with the rewiring probability p and the nearest neighbour parameter k. The 5, 95-percentiles deviate less than 1% from the mean (100 repetitions).

The sizes of the request sets grow with k and p, which results in increased capability for parallelization, and, at the same time, the number of required iterations decreases. This observation agrees well with the speedups for Xeon E2680v3 and Xeon Phi which are shown in Fig. 2 and 3. Be-

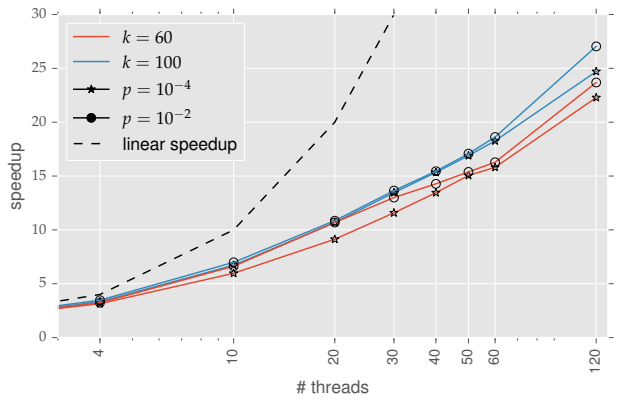

Fig. 3: Xeon Phi: small-world graphs with $2 \cdot 10^6$ vertices, generated with the rewiring probability p and the nearest neighbour parameter k. The $5, 95$-percentiles deviate less than 1% from the mean (100 repetitions).

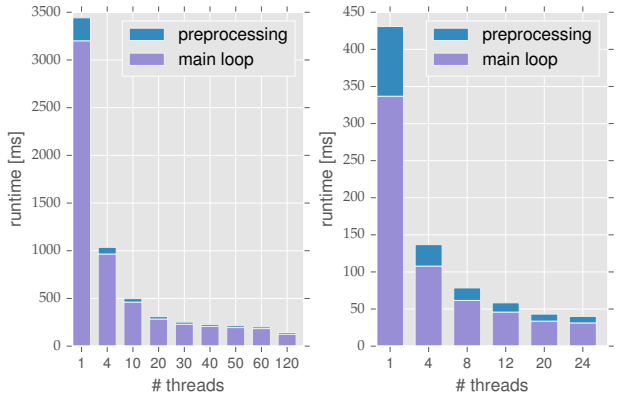

Fig. 4: Timings for preprocessing and the main loop for a small-world graph with k=60, p=10^{-2}, $|V| = 5 \cdot 10^5$. **Left:** Xeon Phi, **Right:** Xeon E2680v3 .

cause of the low *diameter*, i.e. the longest path in the graph, and the sharply concentrated out-degree distribution around k, load-balancing for small-world graphs is not an issue. We get peak speedups of almost 16x and 18x on Xeon E2680v3 (24 threads) and Xeon Phi (60 threads), respectively. The results for other configurations we have used are essentially the same; the worst was obtained for a graph with $5 \cdot 10^5$ vertices, $k = 60$, and $p = 10^{-4}$, where we observed a speedup of 10x with 24 threads on Xeon E2680v3, and a speedup of 14x using 60 threads on the Xeon Phi. Fig. 4 shows the timings for preprocessing and the main loop, on a graph with $5 \cdot 10^5$ vertices, both on Xeon E2680v3 and the Xeon Phi.

Scale-free graphs. We used the Recursive Matrix (RMat) [7] generator implementation from Boost Graph Library to construct scale-free graphs comprised of a given number of vertices and edges. All RMat graphs were generated using the probabilites $a = 0.5$, $b = 0.25$, $c = 0.1$, $d = 0.15$, cf. [7, Sec. 3]. Results for a graph with $2 \cdot 10^6$ vertices and $40 \cdot 10^6$ edges are given in Fig. 5. For each thread configuration 40 successive runtime measurements were taken. These timings exhibit a high variance compared to the results for small-world graphs. A possible explanation could be a large number of concurrent writes of the atomic operation in the `CAS` function (Sec. 3, Alg. 3) to identical locations. This could also be a reason for the sudden improvement in efficiency when hyper-threading is used. The behaviour for scale-free graphs was not explored in detail. Nevertheless, we suspect that the results in Fig. 5 are close to the best that can be achieved with our implementation. We expect that the highly skewed out-degree distribution of scale-free graphs might create load-imbalance, although we did not observe it in our experiments.

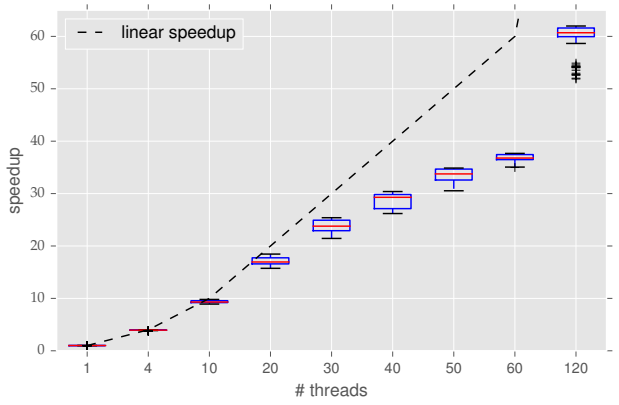

Fig. 5: Xeon Phi: RMat graph with $|V| = 2 \cdot 10^6$, $40 \cdot 10^6$ edges, and $\Delta = 10$. Δ-stepping finished after 4 iterations.

Game Maps. The graph used for the Game Maps scenario is a topological representation of an environment discretized with an occupancy grid. Such a grid defines if a cell location is empty, i.e. accessible, or occupied by some obstacle. Each cell is mapped to a node in the graph, which has as many edges as there are possible movements from that cell. In general, for each node, we have four nodes connected with horizontal and vertical edges and as much nodes connected with diagonal edges. As already mentioned, we chose the cost of 10 for horizontal and vertical movement, 14 for diagonal movement, and a threshold of $\Delta = 13$.

Due to this regular graph structure, we can avoid preprocessing and easily identify if an edge is light or heavy during the request set computation. When dealing with a graph with only two different edge weights, and as a consequence of the code reorganization from Sec. 3, we can also use the SIMD execution, i.e. vectorization. However, because of the low computational intensity of the algorithm this does not result in a significant improvement.

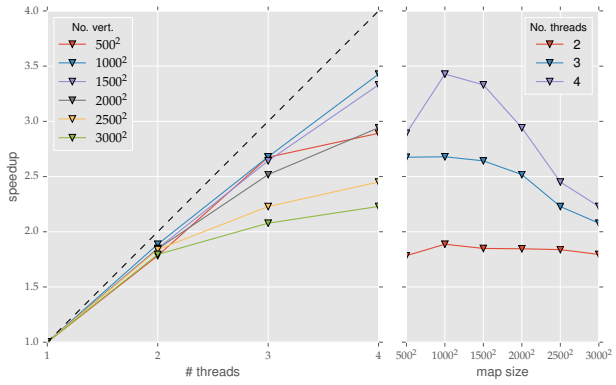

Fig. 6: i7: game maps with 10% of uniformly distributed obstacles. The standard deviation is in the range 0.4-6% from the mean value (10 runs per configuration).

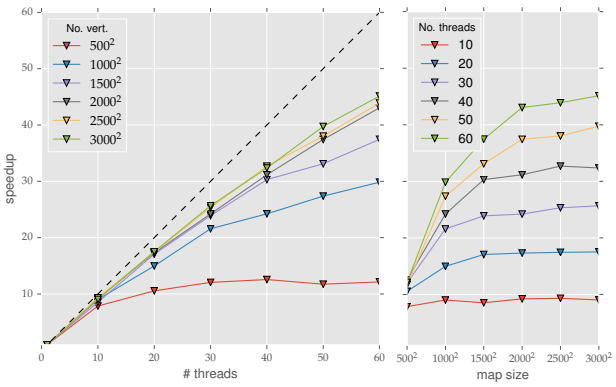

Fig. 7: Xeon Phi: game maps with 10% of uniformly distributed obstacles. The standard deviation is in the range 0.05-6% from the mean value (10 runs per configuration).

In Fig. 6 and 7 we show the speedup on i7 and Xeon Phi, respectively, using square game maps with 10% of uniformly distributed obstacles and an increasing resolution from 500×500 to 3000×3000, with a step of 500. We show the average results of 10 runs per map configuration. For each of these configurations, except for the smallest map, the standard deviation is in the range between 0.05-6% from the mean value, for both architectures. If executed sequentially, our implementation is 3-5 times slower than Dijkstra's algorithm implemented in Boost Graph Library on the i7 and 3-7 times slower on the Xeon Phi. Results show peak speedups of 3.4x and 45x respectively for the i7 and the Xeon Phi. In Fig. 6 we can see how the best performance on the i7 is obtained with map sizes 1000×1000 and 1500×1500 and that, on this architecture, our implementation does not scale well with the dimension of the graph. This is due to the load-imbalance obtained by the OpenMP static scheduling. However, alternative scheduling, e.g. dynamic scheduling or tasking, has negative effects on the data locality (e.g. cache trashing), which results in a decrease in performance. On the Xeon Phi, increasing the number of computational resources improves load balancing (Fig. 7), although the smallest graph, i.e. 500x500, does not exhibit enough parallelism to efficiently exploit the architecture.

5 Conclusion

In this report we have presented an efficient implementation of the Δ-stepping algorithm on shared memory architectures. The performance was analyzed for various graph types on Intel multi-core CPUs and Xeon Phi.

Our choice of the data structure for the buckets has the advantage that it does not require synchronization. However, the implementation presented in this work has to be used with care when applied to large diameter graphs. For these types of graphs the bucket array imposes a substantial overhead, since there is typically only a small fraction of all vertices located in the current bucket, but all of its $|V|$ entries have to be checked in every iteration. For example, on graphs representing d-dimensional square lattices Δ-stepping requires $\mathcal{O}(|V|^{\frac{1}{d}})$ iterations, so the checking of the buckets costs $\mathcal{O}(|V|^{1+\frac{1}{d}})$ operations in total. On the other hand, scale-free and small-world graphs are known to have a diameter at most $\sim \ln|V|$, [8], so the overhead of checking the bucket array is negligible, since the number of iterations is $\sim \frac{\ln|V|}{\Delta}$. Therefore, for those classes of graphs we benefit from such a bucket structure. Attempts to manually redistribute the nodes which are scheduled to be explored in the current phase in order achieve load-balancing were not successful. The inevitable counting of those nodes followed by a reduction creates a too high overhead. Nevertheless, we have observed that a static decomposition of vertices among threads gives good results for the types of graphs and configurations considered.

We have outperformed Boost Dijkstra on all configurations for scale-free and small-world graphs running in sequential mode and, on all shown graph classes, we reach an average parallel efficiency of at least 50%.

6 References

[1] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige, "The office marathon: Robust navigation in an indoor office environment," in *International Conference on Robotics and Automation*, 2010.

[2] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[3] U. Meyer and P. Sanders, "Δ-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114 – 152, 2003, 1998 European Symposium on Algorithms.

[4] N. Edmonds, T. Hoefler, and A. Lumsdaine, "A Space-Efficient Parallel Algorithm for Computing Betweenness Centrality in Distributed Memory," in *Interna-*

tional Conference on High Performance Computing, Dec. 2010, pp. 1 – 10.

[5] N. Edmonds, J. Willcock, and A. Lumsdaine, "Expressing graph algorithms using generalized active messages," *SIGPLAN Not.*, vol. 48, no. 8, pp. 289–290, Feb. 2013.

[6] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak, "An experimental study of a parallel shortest path algorithm for solving large-scale graph instances," in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, Philadelphia, PA, USA, 2007, pp. 23–35, Society for Industrial and Applied Mathematics.

[7] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*, Proceedings, pp. 442–446. Society for Industrial and Applied Mathematics, Apr. 2004.

[8] R. Cohen and S. Havlin, "Scale-Free Networks Are Ultrasmall," *Physical Review Letters*, vol. 90, no. 5, pp. 058701, Feb. 2003.