PPTA: Prefetch-Process-Thread-Alternation to Speed Up Dijkstra's Algorithm

Konstantinos Kanellopoulos, Konstantinos Nikas, Dimitrios Siakavaras, Vasileios Karakostas, Georgios Goumas and Nectarios Koziris

National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
{konkanello,knikas,jimsiak,vkarakos,goumas,nkoziris}@cslab.ece.ntua.gr

Abstract. Many modern applications are memory-bound due to irregular memory access patterns. Dijkstra's algorithm belongs in this class of applications that suffer from this kind of accesses. In this paper, we introduce the Prefetch-Process-Thread-Alternation scheme which is based on a helper-thread prefetching technique. PPTA involves two threads that alternately switch between a prefetching and a process phase, to hide the memory latency caused by cache misses. We evaluate both schemes using two different platforms. Our experiments, on graphs with increasing densities, show that PPTA achieves performance speedup up to 1.82 for sparse and 1.62 for dense graphs.

1 Introduction

Optimizing the performance of memory-bound applications is a challenging task and becomes even harder when they do not expose parallelism in a straightforward way. These applications typically operate on large data-sets with limited temporal locality. The cost of intensive data movement and irregular memory accesses is significant since most of these accesses are served by main memory. The so called memory-wall accentuates the need to use techniques like software prefetching in order to overcome irregular memory patterns which cannot be predicted by modern hardware prefetchers.

In this paper, we implement and evaluate a software-based technique, in modern processors, using a scheme that derives from the idea of moving computation near the data it demands. The proposed scheme is called Prefetch-Process-Thread-Alternation (PPTA) and alternates the role of the involved threads between prefetching and processing. Colocating data and threads appropriately, we are able to substantially reduce cache misses and significantly increase the application's performance.

We apply our scheme to Dijkstra's algorithm [4] since it is a challenging example of memory-bound application whose performance is negatively affected by irregular memory accesses. We also evaluate it, using an AMD's Opteron and an Intel-Broadwell, on graphs with increasing densities, resulting in speedups up to 1.82.

2 Dijkstra's Algorithm

2.1 Algorithm's Basics

Dijkstra's algorithm computes single source shortest paths (SSSP) for directed graphs with non-negative edges and is used in a variety of applications, like routing protocols and VLSI design. Specifically, let $G = (V, E)$ be a directed graph with $n = |V|$ vertices, and $w : E \rightarrow \Re^+$ weight function assigning non-negative real-valued weights to the edges of G. For each vertex v, the SSSP problem computes $\delta(v)$, the value of the shortest path from a source vertex s to v. For each vertex v, Dijkstra's algorithm maintains a shortest-path estimate (or tentative distance) $d(v)$, which is an upper bound for the actual value of the shortest path from s to v, $\delta(v)$. Initially, $d(v)$ is set to $+\infty$ and through successive edge relaxations it is gradually decreased, converging to $d(v)$. The relaxation of an $edge(v, w)$ sets $d(w)$ to $min\{d(w), d(v) + w(v, w)\}$, which means that the algorithm tests whether it can decrease the weight of the shortest path from s to w by going through v. Finally, this algorithm updates an array called *previous* so that the shortest path to a vertex v can be recursively reconstructed.

Initially, all vertices v are unreached ($d[v] = +\infty$) except s ($d[s] = 0$). For each iteration, the vertex with the minimum key is extracted from a min-priority queue (**ExtractMin**: line 8) and its state is settled to *visited*. Then, its outgoing edges are relaxed (**Update**: lines 18,19) and the keys d of the neighbors, whose $state \neq visited$, are decreased if needed. It is also essential to update the *previous* array and decrease the key of each neighbor-node in the priority queue to retain its minimum property (**Decrease-Key**: line 17).

In our implementation of the algorithm we use a m-ary heap as a min-priority queue and m is set to 2, resulting in a binary heap. We also use a practical optimization of Dijkstra's algorithm called Uniform Cost Search (UCS) [5]. During the initialization phase of the default algorithm, all vertices are entered into the priority queue. However this is not mandatory since the algorithm can start with its priority queue containing a single element and continue by dynamically inserting the new vertices discovered. UCS maintains a smaller priority queue and speeds up the Decrease-Key and Extract-Min operations and reveals that the Update operation is a major bottleneck. The algorithm is presented in more detail in Alg. 1.

An intuitive choice for parallelizing Dijkstra's algorithm is to exploit parallelism at the inner loop by relaxing all outgoing edges of vertex u in parallel. Though, this approach leads to significant slowdowns because we need to impose excessive synchronization to the threads trying to concurrently access the priority queue and the *distance* & *previous* arrays [2].

2.2 Performance analysis

We profiled the different operations of Dijkstra's Algorithm in a fine-grained way and we present in Figure 1, the distribution of the execution times, in graphs with 10M nodes and increasing number of edges. For this experiment we have

Algorithm 1 : Dijkstra with UCS optimization

Require: Graph $G(V, E)$, Source vertex S
Ensure: Predecessors array *previous*
Shortest distance array d

```
1: for all v ∈ V[G] do
2:    d[v] ← +∞
3:    previous[v] ← undefined
4: d[s] ← 0 // Initialization
5: S ← empty set
6: Q ← s
7: while Q is not empty do
8:    u ← Extract-Min(Q) // Extract-Min
9:    S ← S ∪ {u}
10:   for all edge (u, v) outgoing from u do
11:      if v ∉ S then
12:         sum ← d[u] + w(u, v)
13:         if v ∉ Q then
14:            Insert(Q,v,sum)
15:         else
16:            if sum < d[v] then // Update
17:               u ← DecreaseKey(Q,v,sum)
18:               d[v] ← d[u] + w(u, v)
19:               previous[v] ← u
```

used an Intel Broadwell-EP, whose characteristics are described in Section V. The Update operation includes lines 18,19 of Alg. 1 excluding line 17, i.e. the Decrease-Key operation. It is evident that, the **Update** operation becomes the main part of the execution as graphs become denser and the **Initialization** phase's piece of execution time is minimal compared to the other ones.

As shown in Figure 1, the Extract-Min and the Decrease-Key operations affect the execution time less than the Update does, as graphs become denser. On the one hand, both operations consist of an upward traversal of the minimum heap that involves a small number of hops, since the height of the heap is logarithmic to the number of vertices. Specifically the Decrease-Key operation is performed only if we need to relax an edge. On the other hand, the Update operation consists of four read operations that occur for every edge of the extracted vertex. The four memory locations that need to be read are: the *distance* array at indexes v and u, the cost of the edge $w(u, v)$ and the *previous* array at index v. Reading the *distance* and the *previous* array, whose size is equal to the number of vertices (10M), at a random index v, generates memory accesses resulting in cache misses. In other words, there is no temporal locality between the data needed to relax the edges of u. In dense graphs, this phenomenon is more intense since the number of edges per vertex is large. As irregular memory accesses are increased and our data-set does not fit in the lower levels of the cache hierarchy, performance becomes worse.

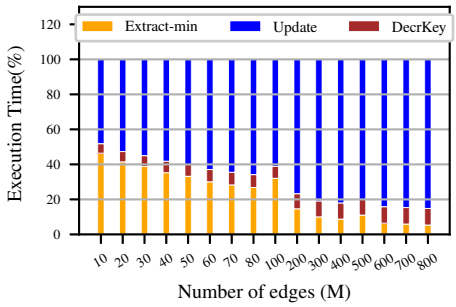


Fig. 1: Breakdown of execution time percentage. The Update operation dominates in dense graphs.

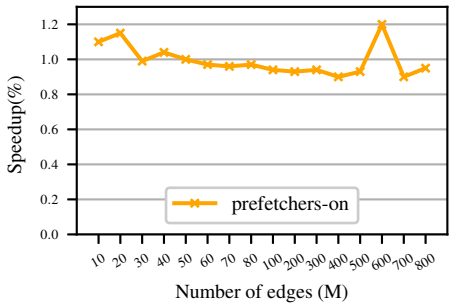


Fig. 2: The effect of enabling hardware prefetchers on Broadwell-EP. A slowdown is observed in most cases.

In addition, the hardware prefetchers used in modern systems fail to prefetch the correct piece of data in these irregular cases. In Fig. 2 we present the achieved speedup when hardware prefetchers are enabled compared to when they are disabled. Given graphs in range of 40M-500M, we notice a drop in performance, at about 0.9 on average. We observe that prefetchers provide no performance boost to this irregular memory accesses algorithm. In this paper, we will apply software prefetching to accelerate such memory accesses.

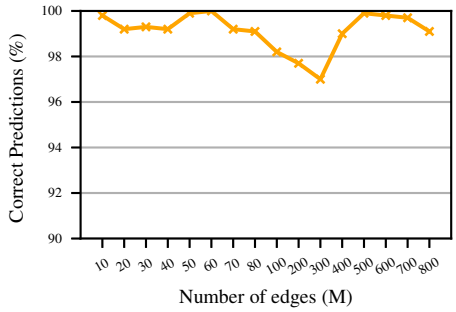


Fig. 3: Correct Predictions (%) of the forthcoming Extracted minimum vertex. The prediction is 99.1% accurate on average.

3 Mitigating memory latency with software prefetching

Our goal is to hide the latency of cache misses that originate from the Update operation. The idea of software prefetching is to predict irregular memory accesses, so that the piece of data needed is loaded from the main memory to the cache hierarchy before it is accessed by the application [1], [6]. Prefetching the correct chunk of data relies on the prediction of the forthcoming extracted minimum-key vertex u. Using the information stored in the priority queue, we

are able to make a reliable prediction about the next extracted vertex. The piece of data we prefetch are the neighbors of the minimum key-vertex of the priority queue after the extract-min operation of the current phase of the algorithm but before its Update Operation. Our guess is that there is a high probability this vertex will be extracted next. In Fig.3 we present the ratio of correct predictions and we observe that it is almost perfect in every graph. This way, the probability that the data we prefetch will be utilized is much higher.

Based on the previous motivation, we came up with the Prefetch-Process-Thread-Alternation scheme which is based on a common and less complex prefetching helper-thread scheme.

3.1 Prefetching Helper-Thread scheme

Firstly, we implemented a simple Prefetching Helper Thread (PHT) scheme similar to [7]. This scheme consists of two threads. The first one constantly executes the algorithm (main) and the second works in parallel with the main thread and aggresively prefetches data to the nearest shared cache memory, helping the main thread avoid misses from upcoming irregular memory accesses. This scheme depends on the shared cache memory between the two threads and on the shared pipeline resources, in case we employ simultaneous multithreading. As shown in Section V. this scheme is not capable of reaching increasing speedups as graphs become denser. Therefore, we came up with the PPTA scheme whose performance scales better as graphs become denser.

3.2 Prefetch-Process-Thread-Alternation scheme

In this section we present a more sophisticated scheme called Prefetch-Process-Thread-Alternation (PPTA). It is also a two-thread scheme that employs software prefetching. However, this technique derives from the idea of moving computation near data. In fact, we try to hide cache misses using two different phases for each thread. Compared to the original algorithm, we expect a significant reduction of cache misses, created by the Update operation, that results in improved performance.

In our implementation one of the two threads starts by initializing all the flags needed for synchronization of both threads and the data structures Q, d and *previous*. Our scheme consists of two phases in each round of the algorithm: the prefetching and the process phase. One of the threads is responsible for executing the algorithm while the prefetch-thread prefetches the data that will be required during the next round. More specifically, the process-thread extracts the minimum vertex while the prefetching-thread is waiting for the prefetch signal. After the Extract-Min operation, the process-thread notifies the prefetching-thread to read the new minimum key of the pq and start prefetching the neighbors of the speculatively selected minimum key-vertex. Then, the process-thread begins the Update operation, whose completion signals the end of the round and informs the prefetching-thread to stop prefetching. As shown in Fig. 4, at the start of round

$i+1$ the two threads change roles, to exploit the fact that the prefetching-thread brought the data it needs, for the upcoming Update, to the L1 cache memory.

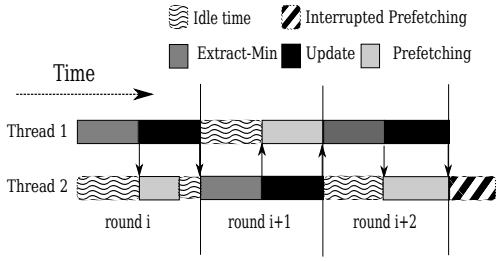


Fig. 4: Execution pattern of PPTA scheme

This process continues alternately until Q is not empty and all distances have settled. There are two cases we need to examine regarding the prefetching phase. They are illustrated in Fig. 4. In rounds i and $i+1$, thread 2 has finished its prefetching phase before thread 1 completes the Update operation. In contrast, during round i+2, thread 2 is interrupted during its prefetching phase and begins the Extract-min operation without having prefetched the whole set of data indespensable for the relaxation phase. This kind of interrupt is necessary considering that, if thread 2 waits for all chunks of data to be prefetched, the upcoming Extract-Min operation will not be executed until the prefetching is completed and therefore our algorithm's execution will slow down. The code for both threads is presented in Alg. 2 and Alg. 3.

To coordinate the two threads we are using four synchronization flags. Two flags are needed to signal the start of each thread's prefetching phase and two more to notify them for the completion of the relaxation phase and the start of the Extract-Min.

4 Experimental Evaluation

4.1 Experimental setup

For our experiments we used two dual socket servers, equipped with AMD's Opteron and Intel's Broadwell-EP processors. The main characteristics of the two systems are shown in Table I and their cache memory hierarchies are respectively presented in Fig. 5. We used the configurations presented below:

Algorithm 2 : Thread 1 code for PPTA scheme

```
Initialize Q, d, previous
prefetch_1 ← 1 extract_1 ← 1
prefetch_2 ← 0 extract_2 ← 0
while Q is not empty do
  while(prefetch_1=0);
  prefetch_1 ← 0
  min ← ReadMin(Q)
  for all neighbors of min and while extract_1=0 do
    prefetch neighbors' data
  while(extract_1=0);
  extract_1 ← 0
  u ← Extract-Min(Q)
  prefetch_2 ← 1
  for all edge (u, v) outgoing from u do
    relaxation phase
  extract_2 ← 1
```

Algorithm 3 : Thread 2 code for the PPTA scheme

```
while Q is not empty do
  while(prefetch_2=0);
  prefetch_2 ← 0
  min ← ReadMin(Q)
  for all neighbors of min and while extract_2=0 do
    prefetch neighbors' data
  while(extract_2=0);
  extract_2 ← 0
  u ← Extract-Min(Q)
  prefetch_1 ← 1
  for all edge (u, v) outgoing from u do
    relaxation phase
  extract_1 ← 1
```

- **Opteron**:
  1. Threads pinned to cores which share the L2 cache.
- **Broadwell-EP**:
  1. Threads pinned to cores which share the L1 cache (SMT).
  2. Threads pinned to cores which share the L3 cache.

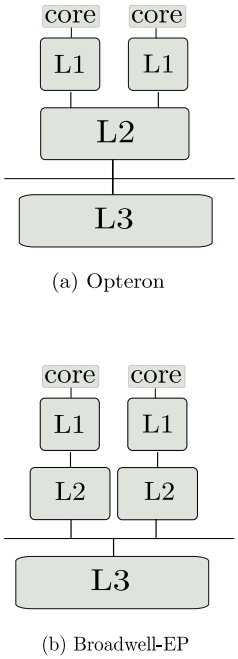
Both schemes were developped using C++. More specifically, the coordination flags used in both schemes were implemented using C++ atomic operations and the acquire/release memory order. We also disabled the hardware prefetchers, based on the results in Section III.

4.2 Reference graphs

In order to experiment on graphs with variety of density and structure, we used the GTgraph generator [3]. We constructed graphs with 10M vertices and varying number of edges from the *Random*, *R-MAT* and *SSCA families*.

Table 1: Systems' Configurations

Name	**AMD Opteron**	**Intel Broadwell-EP**
#Cores	4x8	2x22
#Threads	32	88 (SMT)
Core clock	2.4 GHz	2.2 GHz
L1(Data)	16 KB, 4-way, 64B block size	32 KB, 8-way, 64B block size
L2	256 KB, 8-way, 64B block size (**shared per 2 cores**)	2 MB, 16-way, 64B block size (**private**)
L3	16 MB, 128-way, 64B block size (shared per NUMA node)	56 MB, 20-way, 64B block size (shared per die)
Memory	250 GB	64 GB
OS	Debian 8.8	Debian 8.3
Linux Kernel	3.2.0	4.7.0
GCC	4.9.2 with -O3 optimization	4.9.2 with -O3 optimization



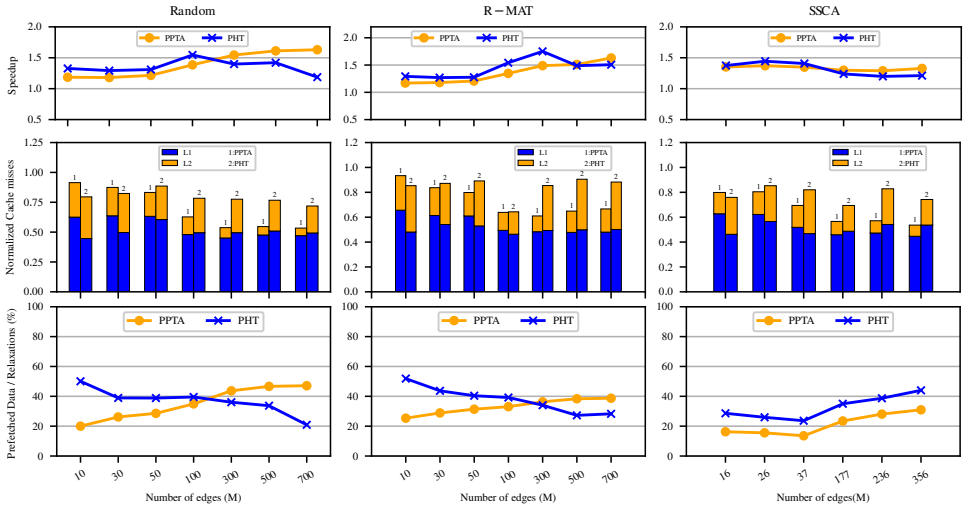
(a) Opteron

(b) Broadwell-EP

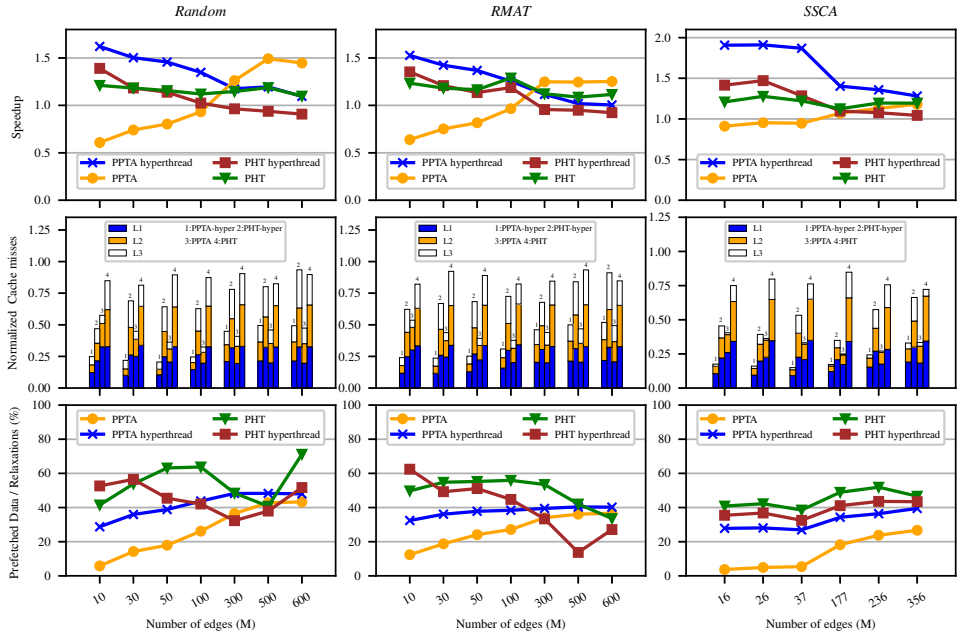
Fig. 5: Cache Hierarchy of both platforms

4.3 Performance Results

We evaluate both the PHT and also the PPTA schemes. Figure 6 presents speedup, sum of data cache misses and the ratio of prefetched data per chunk of relaxations, achieved by the PPTA and the PHT scheme. The results concerning cache misses are normalized using the ones measured from serial execution and refer to the process phase of both schemes (main thread in PHT). In the case of Opteron, we were not able to measure L3 data misses due to the lack of appropriate performance monitoring events. We focus our study on one family of graphs, the *Random,* as the other families exhibit similar behavior. Figure 6 presents the results on AMD Opteron and Intel-Broadwell, respectively. Below, we analyse the results of both schemes and correlate them with their characteristics.



(a) Opteron



(b) Broadwell-EP

Fig. 6: Rows present speedup, cache misses of the process phase, and ratio of the successfully prefetched data. Columns represent the three graph families.

Regarding the PHT scheme, although it's performance on sparse graphs is better than the PPTA, we witness a significant slowdown as graphs become denser. This can be attributed to the fact that while in PPTA the process-thread finds its data prefetched in its L1 cache, in the PHT scheme the same data is prefeteched in the L2 (Opteron) or L3 (Broadwell) and the process-thread has to fetch it in the L1 itself.

We observe that speedup, concerning the PPTA scheme in Opteron, is proportional to the density of the graph. As graphs become denser more data are prefetched, cache misses of the process phase are reduced, and greater speedup is achieved. PPTA results in noticeable reduction of cache misses, with maximum being 45% as shown in Figure 6a, leading to a maximum speedup 1.62 using the densest graph consisting of 700M edges. This pattern is also present in Broadwells' metrics, where maximum speedup is 1.5.

However, there is a major difference regarding sparse graphs. In Broadwell's case, we observe a significant slowdown in sparse graphs in contradiction with Opteron's case where there is a slight speedup. This can be attributed to the fact that using the PPTA scheme we introduce synchronization and coherency overhead. Both threads frequently update the priority queue and the *distance*,*previous* arrays and when each of them tries to read or write in these memory locations, cache coherency mechanisms trigger cache-to-cache transfers. Regarding Broadwell's shared L3 hierarchy, sparse graphs expose the overhead of these coherence mechanisms which is higher compared to Opteron's case where data transfers occur between the L2 caches.

We deduce that the most beneficial option for the PPTA scheme is to employ cores that share as many levels of the cache hierarchy as possible. In this way, we are able to reduce coherency overhead since data travel for shorter distances inside the cache hierachy.

By using Broadwell-EP, we can employ hyperthreads that share the whole memory hierarchy including the highest level of cache (L1). The PPTA-hyperthread scheme shows appreciable speedups in sparse graphs. However, as graphs become denser performance worsens. This may be attributed to the fact that as

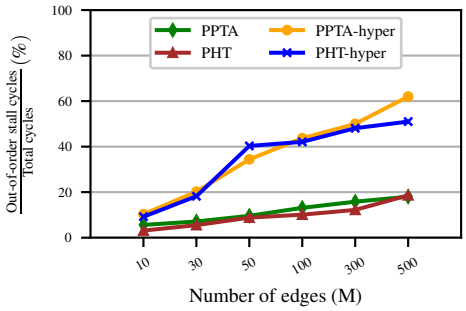


Fig. 7: Portion of process phase core is stalled due to pipeline's resource contention, as reported by performance counter(Event A2H, Umask 01H)

the prefetching phase becomes more memory-intensive, the hyperthread reserves an increasing number of pipeline's resources to successfully prefetch the data needed from our application and subsequently slows down the Update operation performed by the process-thread.

In Fig. 7 we measure, for both schemes, the portion of cycles each thread remains idle during the process phase because of out-of-order resources contention. We witness that, by employing the hyperthreads, and as graphs become denser, the main part of the execution is spent on waiting to reserve out-of-order resources. The contention is also increased because of the larger data-set used. More specifically, in the case of the 500M edges *Random* graph, despite the similar reduction of cache misses that we achieve by using PPTA and PPTA-hyperthread, speedup is smaller in the second case because of the contention overhead.

In sparse graphs, the hyperthreads contest less for the resources of the pipeline resulting in noticeable speedups with the maximum of them being 1.61 (*Random*). It concerns the sparsest of our graphs with 10M edges.

Regarding Broadwell, there is an important trade-off concerning the use of the hyperthread. Although, we reduce coherency overhead by using a shared L1 hierarchy and benefit in sparse graphs, denser ones expose the increasing overhead of pipeline contention and lead to slowdowns.

5 Related Work

To deal with the impact of indirect memory accesses, other schemes propose the use of specialized hardware prefetchers such as the Indirect Memory Prefetcher [10] or the VLDP [9], at the cost of increased hardware. In [8] helper-threads are spawned to accelerate the computational part of Dijkstra's Algorithm. However, one drawback of this scheme is the cache pollution, caused by the helper threads, that affects the performance of the main thread (max speedup 1.84 using 14 threads) . Finally, in [6] one or multiple threads prefetch data while the main thread runs the computation and the main thread migrates to the other core. In contrast, the PPTA scheme does not involve thread migration and prefetches data by exploiting the algorithm's characteristics.

6 Conclusions

In this paper, we introduced the Prefetch-Process-Thread-Alternation scheme that employs software prefetching to deal with irregular memory accesses and speeds up Dijkstra's algorithm. We conclude that our scheme introduces coherency overhead which can be diminished by employing cores that share as many levels of the cache hierarchy as possible. Moreover, PPTA achieves increasing speedups as graphs become denser. AMD Opteron achieves speedups up to 1.62 for the densest of our graphs and Intel Broadwell-EP reaches 1.82 for a sparse graph, with the hyperthread in use. Considering the serial nature of

Dijkstra's algorithm and the inherent difficulties in its parallelization, this is a significant performance gain.

The fact that the PPTA scheme achieves an increasing rate of speedups as graphs become denser, is a key result since extremely large scale graphs have emerged in various modern applications, such as, the graph of the Twitter social network and the neuronal network of the Human Brain Project.

As future work, we intend to explore the potential of a Near-Data Processing scheme which provides specialized hardware and appropriate software API to deal with memory-intensive applications such as graph processing. We would also like to expand our research towards different kinds of memory intensive algorithms related to data analytics and machine learning.

References

1. Ainsworth, S., Jones, T.M.: Software prefetching for indirect memory accesses. In: Proceedings of the 2017 International Symposium on Code Generation and Optimization. pp. 305–317. CGO '17, IEEE Press, Piscataway, NJ, USA (2017), http://dl.acm.org/citation.cfm?id=3049832.3049865
2. Anastopoulos, N., Nikas, K., Goumas, G.I., Koziris, N.: Early experiences on accelerating dijkstra's algorithm using transactional memory. In: 23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009. pp. 1–8. IEEE (2009), https://doi.org/10.1109/IPDPS.2009.5161103
3. Bader, D., Madduri, K.: gtgraph: A suite of synthetic graph generators. http://www.cc.gatech.edu/kamesh/GTgraph/ (2006)
4. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms. McGraw-Hill Higher Education, 2nd edn. (2001)
5. Felner, A.: Position paper: Dijkstra's algorithm versus uniform cost search or a case against dijkstra's algorithm. In: Borrajo, D., Likhachev, M., Lpez, C.L. (eds.) SOCS. AAAI Press (2011)
6. Kamruzzaman, M., Swanson, S., Tullsen, D.M.: Inter-core prefetching for multicore processors using migrating helper threads. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 393–404. ASPLOS XVI, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1950365.1950411
7. Kim, D., Liao, S.S.w., Wang, P.H., Cuvillo, J.d., Tian, X., Zou, X., Wang, H., Yeung, D., Girkar, M., Shen, J.P.: Physical experimentation with prefetching helper threads on intel's hyper-threaded processors. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization. pp. 27–. CGO '04, IEEE Computer Society, Washington, DC, USA (2004), http://dl.acm.org/citation.cfm?id=977395.977665
8. Nikas, K., Anastopoulos, N., Goumas, G.I., Koziris, N.: Employing transactional memory and helper threads to speedup dijkstra's algorithm. In: ICPP 2009, International Conference on Parallel Processing, Vienna, Austria, 22-25 September 2009. pp. 388–395. IEEE Computer Society (2009), https://doi.org/10.1109/ICPP.2009.60
9. Shevgoor, M., Koladiya, S., Balasubramonian, R., Wilkerson, C., Pugsley, S.H., Chishti, Z.: Efficiently prefetching complex address patterns. In: Proceedings of

the 48th International Symposium on Microarchitecture. pp. 141–152. MICRO-48, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2830772.2830793
10. Yu, X., Hughes, C.J., Satish, N., Devadas, S.: Imp: Indirect memory prefetcher. In: Proceedings of the 48th International Symposium on Microarchitecture. pp. 178–190. MICRO-48, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2830772.2830807