Acceleration of Dijkstra's Algorithm on Multi-core Processors

Abhay Prasad, Sukruth Kumar Krishnamurthy, and Youngsoo Kim
Department of Electrical Engineering, San Jose State University, San Jose, CA
abhay.prasad@sjsu.edu, sukruthkumar.krishnamurthy@sjsu.edu, youngsoo.kim@sjsu.edu

Abstract

The Single Source Shortest Path (SSSP) problem has been solved using various algorithms. We focus on accelerating a well known SSSP algorithm, the Dijkstra's algorithm using a multi–core CPU. We achieve acceleration by using the iParallel kernel, a hybrid kernel that runs on the sequential as well as parallel kernels intelligently. Our experimental results help find an optimal threshold to exploit parallelism for different sizes of the search terrain. We have achieved an overall acceleration of ∼48% and ∼51% on a dual- core ARM A9 processor and a 16-core Epiphany co-processor respectively.

***Keywords*—Multicore processing, OpenMP, path finding**

I. Introduction

Path finding is being used extensively in various domains such as the self driving car [1], gaming [2], route planners for the internet [3] and web searching [4]. Algorithms that are used to solve the shortest path problem are expensive in terms of computation. They require large amounts of memory and computational power as the size of the search area increases. The advanced path finding algorithms make use of heuristic approaches to reduce computational complexity [5]. The single source shortest path problem (SSSP) focuses on finding the shortest path between a single start node and the goal node.

This work targets at accelerating a commonly used algorithm to solve SSSP, the Dijkstra's algorithm. We have exploited the best of the parallel as well as the sequential algorithms to result in an efficient hybrid approach. The proposed algorithm intelligently switches between these two kernels considering the size of nodes to process at run time. A detailed description of this can be found in section II.

All our experiments have been done on a Parallella board [6] using Open Multi-Processor (OpenMP) APIs. The description of our experimental setup can be found in section III. A detailed description of the results can be found in section IV. We have achieved an overall average speed up of ∼48% on a dual ARM core A9 processor found on the Zynq Soc(ZDP) and ∼51%(approximately) on a 16-core Adeptiva Epiphany co-processor (EPP) with the proposed approach.

This work is partially supported by the U.S. Army grant W911NF-16-1-0217.

A. Single Source Shortest Path Problem (SSSP)

SSSP has been a popular topic in literature and has been addressed using various algorithms [7]. The most basic of these are the Depth first search(DFS) and Breadth first search(BFS) algorithms [8]. These algorithms start from a node of a graph and traverse the graph entirely to check if a given node is present or not. Both algorithms traverse through the graph in different patterns.

The most commonly used algorithm for path finding is the Dijkstra's algorithm. This is a modification of BFS algorithm. While BFS considers equal weights for traversal between nodes (edges) in the graph, the Dijkstra's algorithm offers the advantage of assigning weights to each edge of the graph hence, making it more versatile for real world applications. Several heuristic methods can be used to reduce the computational complexity of path finding algorithms in known terrains. A well known algorithm that uses heuristics to find the shortest path between two nodes is the A^* algorithm [2]. For each iteration, the A^* algorithm evaluates the current node with the sum of the distance it takes to reach that node from the start node and the distance estimated to reach the goal node from the current node. If the estimated distances are all 0, the algorithm is equivalent to the Dijkstra's algorithm. Hence, the Dijkstra's algorithm is a special case of the A^* algorithm. A detailed study of both these algorithms can be found in [9].

The Dijkstra's algorithm computes the shortest path between the source node and all other nodes in the graph. This algorithm serves to be a typical optimization problem [10]. The time complexity of the Dijkstra's algorithm is $O(n^2)$ [11]. The Dijkstra's algorithm is as shown in Algorithm 1. During the initialization phase, the value of tentative distances, D is all set to ∞ and all the nodes in the graph are marked as unsettled($\forall u \in U = true$). The tentative distance of the start node $D[s]$ is set to 0 and is settled ($U[s] = false$) before starting the search. A matrix of weights, $w(v, e) : e \in E$ is used to represent the weight (cost) for each of the nodes ($v \in V$) traversed in the graph, $G = (V, E)$. For each iteration of the outer

loop, the node ($n \in N$) with the least tentative distance in D is found. This vertex is marked as the frontier node, f. The tentative distance for each neighbor of f in D is updated in the inner loop and node n is considered to be settled ($U[n] = false$). The outer loop is executed until all nodes ($\forall n \in N$) in graph $G(V, E)$ are settled.

From the above description of algorithm 1, we can consider the working of the algorithm as three basic operations i.e., initialization(initialization kernel), finding the minimum(minimum kernel), and the updating of tentative distances(relax kernel). The initialization kernel can be considered to be lines 2 – 7 in algorithm 1. The minimum kernel would be the function described in algorithm 2 (line 8 of Algorithm 1) and the update kernel would be lines 11 – 14 (algorithm 1). Each of these kernels can be worked on separately to achieve an overall acceleration.

Algorithm 1: Dijkstra's algorithm

```
   Data:
   Number of nodes, N
   Tentative distances, D[N]
   Settled nodes, U[N]
   Start node, s
1  begin
2      for i ← 1 to N do
3          D[i] ⟵ ∞;
4          U[i] ⟵ true;
5      end for
6      D[s] ⟵ 0
7      U[s] ⟵ false
8      for i ← 1 to N do
9          f = min(D, U)
10         for n ← 1 to N do
11             if D[f] + w(f, n) < D[n] then
12                 D[n] ← D[f] + w(f, n)
13                 U[n] ← false
14             end if
15         end for
16     end for
17 end
```

B. Previous works

The acceleration of SSSP is a well known topic in literature [12], [13]. Acceleration of the BFS, Belman ford, Dijkstra, A^* and D^* algorithms have been done using both, algorithmic as well as hardware techniques [14]. These parallelized algorithms have been implemented in Graphic Processing Units (GPUs), multi-core CPUs and FPGAs in order to achieve faster execution time[15], [16], [17].

In literature, parallelizing the Dijkstra's algorithm has been done in two approaches. The first approach is to work on disjoint graphs parallely [12]. The second approach is to parallellize the inner loop of the sequential Dijkstra's algorithm [18], [19]. In this paper we propose a hybrid approach that is an extension of the second method to achieve acceleration.

Algorithm 2: Pseudo code of minimum index kernel

```
   Input: D, U
   Output: minInd
1
2  minVal ← ∞
3  begin
4      for n ← 1 to N do
5          if (U[n] ≠ false & D[n] < mval) then
6              minVal = D[n]
7              minInd = n
8          end if
9      end for
10 end
```

Cruiser et. al. [18] have proposed an efficient method to parallelize the Dijkstra's algorithm. In this work, the authors propose to settle nodes in the frontier set all together at once according to a pre-defined criterion in order to reach the goal faster, thus exploiting parallelism. An extension of this algorithm has been proposed by Martin et. al [19]. The criterion to select nodes to settle has been modified to best suite the architecture of GPUs. They propose to have achieved speed-ups of 32x by using Fibonacci heaps and running their algorithm on a GPU. Also, Ortega et. al. [10] propose a modification of the algorithms proposed by cruiser et al and Martin et. al. [18], [19] to further optimize the algorithm in terms of memory access.

Several hierarchical approaches have also been proposed [20], [21] to accelerate the Dijkstra's Algorithm. Approaches exploiting FPGA implementations for efficient memory access [16], [22], [23] for path finding can also be found in literature.

C. Parallel programming models

The maximum processing speed a processor can work is limited to its clock frequency. In order to maximize the speed of execution above that of the clock frequency, parallel computing is necessary. This can be found in high performance computer systems as well as on distributed computing systems. Parallel programming models work on the concept of shared memories hence, making it possible to achieve higher execution speeds. A commonly used programming models to support multi-processor systems is OpenMP [24] which is based on the creation of threads in a shared memory environment.

This work uses OpenMP to accelerate the performance of the Dijkstra's algorithm. In a For or Do loop, the OpenMP operates as a fork and join model. The program starts as a single master thread and executes sequentially until a parallel region is reached. As soon as the master

reaches the parallel region, it creates a team of threads to process data in parallel. When the team of the threads complete in the parallel regions, all the threads join and synchronize. As soon as they finish synchronizing, the code continues execution in a sequential manner until another parallel region is reached.

D. Parallella board

The Parallella from Adapteva is a credit card sized computer using Epiphany 16 core RISC SoC and a Zynq SoC [25]. The board also features 1GB of RAM and is highly energy efficient [6]. The Epiphany chip offers a scalable, multi-core memory sharing capability [25]. It has a 2 dimensional mesh network connected on a Network on chip with a low latency [26]. The Zynq SoC consists of Dual–core ARM cortex A9 processor which is the host to the on board Adeptiva Epiphany III co–processor.

II. Parallellizing the Dijkstra's Algorithm

As discussed in section I-C, parallel computing is achieved with the help of threads. It must be noted that the creation of a threads always causes an overhead during execution. Hence, parallelism would result to be efficient only when processing large amounts of data as the overhead for creating and destroying threads is compensated for in such cases. Hence, for relatively low number of iterations the time for executing the code sequentially would result in a better execution time than running it in parallel. Exploiting these base concepts of the working of threads on a multi-core system, we propose to extend it on the algorithm proposed by Cruiser et. al. [18].

The sequential execution kernel follows the algorithms 1 and 2 for execution. The iParallel kernel executes as described in algorithm 3. During run time, the kernel decides whether to execute in the parallel or sequential regions depending on the potential to parallelize for a given iteration of the outer loop. This is determined by algorithm 4. This function checks to see the number of nodes in D that are $\leq f$ and are in the list of unsettled nodes. These nodes are marked as potentially parallelizable nodes and stored in a list γ. The number of nodes in γ is stored as ρ. In other words γ consists of ρ number of elements. If ρ is greater than the set threshold η, the parallel regions is executed, else the code executes in the sequential region. Similar to the work done by Ortega et. al. [10], the parallel kernel is designed to execute in parallel using OpenMP in the parallel region.

Algorithm 5 shows the working of the relax kernel in the parallel region of the iParallel kernel. A thread for each value in γ_i $(0 < i < \rho)$ is required to be created to perform this operation. Before updating the value in D, an OpenMP atomic region is entered to make sure the operation is thread safe. Since ρ number of nodes are settled in parallel for a single iteration of the outer loop, the overall execution time turns out to be better than using the sequential kernel.

Algorithm 3: Overall working of the iParallel kernel

```
1  begin
2      delta ← 0
3      while δ ≠ ∞ do
4          f ← minVal(U, D)
5          ρ, γ ← iParallel_update_kernel(D, U, N)
6          if ρ > η then
7              <<< Enter parallel kernel >>>
8          end if
9          else
10             <<< Enter sequential kernel >>>
11         end if
12     end while
13 end
```

Algorithm 4: iParallel_update_kernel

```
   input : D, U, N
   output: ρ, γ
1  ρ ← 0
2  <<< PARALLEL REGION >>>
3  <<< SET OPENMP REDUCTION ON ρ >>>
4  for i ← 1 to N do
5      if U[i] = true and D[i] ≤ f then
6          ρ ← ρ + 1
7          γ_i ← true
8          U[i] ← false
9      end if
10 end for
11 return ρ
```

Algorithm 5: iParallel relax kernel

```
1  <<< PARALLEL REGION >>>
2  tid = omp_get_thread_num()
3  if (U[tid] = true) then
4      for (v ← tid to N) do
5          if U[v] = true then
6              BEGIN CRITICAL REGION
7              D[v] ← D[tid] + w(tid, v)
8              END CRITICAL REGION
9          end if
10     end for
11 end if
```

III. Experimental setup

The proposed hybrid approach has been tested for N varying from 10^1 - 10^8 number of nodes. All experiments have been done using an adjacency list to represent

the graph. This offers less memory consumption thus allowing us to execute the algorithm on larger data sizes. The Zynq ARM core A9 dual-core processor (ZDP) and the Epiphany co-processor (EPP) on the Parallella board have been used to evaluate the algorithm. For each value of varying N, η was varied in steps of 10 from 0 - 100. The code for all the computations have been written in the C programming language. A gcc compiler was used to compile the code with no optimization and enabling the OpenMP flag (-fopenMP).

Considering a 2-D plane, the cost matrices designed were based on the Pythagorean theorem. In order to eliminate floating-point computations, the cost for movement between NORTH(N)–SOUTH(S) and EAST(E)–WEST(W) was set to 10 and the cost for moving diagonally (NE–SW and NW–SE) was set to 14.

IV. Results

Figure 1 shows the execution time for different sizes of N. We observe the algorithm to perform better as the number of nodes(N) increases for ZDP and EPP. The execution time for EPP is slightly better than that of ZDP as N increases. This is due to the large overhead involved in copying the data on to the co-processor. We predict the execution time of ZDP to be significantly better than EPP for even higher values of N. Although, it must be noted that the execution time on ZDP is comparable to that of EPP for lower data sizes.

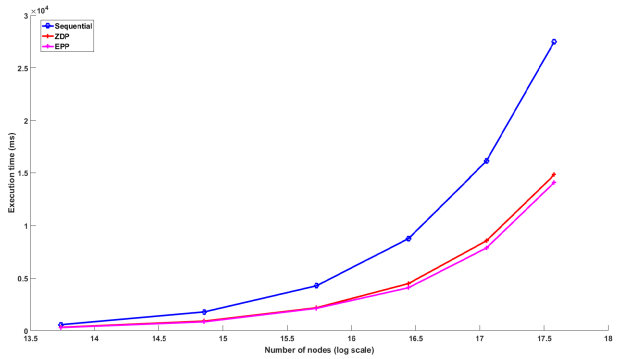

Fig. 1. Execution results

As the threshold(η) varies for different test cases, the number of times the parallel region is entered varies accordingly, thus effecting the *SpeedUp*. Figure 2 shows the speedups as η increases. We can see the speed up to be maximum for $\eta = 50$ for the test case considered. We can clearly see *SpeedUp* to reduce for values of $\eta > 50$. This is because the code enters the sequential regions more than the optimal number of times. For higher values of η, the code always executes in the sequential region, thus resulting in a constant *SpeedUp*.

V. Conclusion

We have proposed a hybrid methodology to accelerate a rigidly sequential algorithm. An overall acceleration

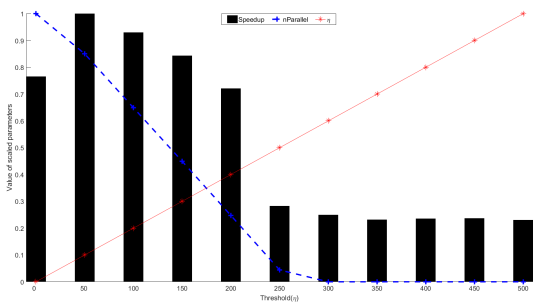

Fig. 2. Relation between η and speed up

of ~51% was obtained for an input size of 10^8 nodes. We can clearly see the acceleration to increase as the number of nodes increases. EPP proves to perform better than ZDP for relatively large number of nodes. This is due to the fact that the co-processor has an overhead to copy the data from the host processor. Although, it is interesting to note that EPP performs similar to ZDP for lesser number of nodes.

As a part of our future work, we propose to make a table of optimal thresholds for exploiting the best of the hybrid approach. A more optimal data structure can be used to evaluate the algorithm for higher number of nodes than the ones we have used. Also, it would be interesting to further accelerate the kernel using the FPGA on the Zynq processor.

References

[1] Assia Belbachir, Rémi Boutteau, Pierre Merriaux, Jean-Marc Blosseville, and Xavier Savatier, "From autonomous robotics toward autonomous cars," in *Intelligent Vehicles Symposium (IV), 2013 IEEE*. IEEE, 2013, pp. 1362–1367.

[2] Khammapun Khantanapoka and Krisana Chinnasarn, "Pathfinding of 2d & 3d game real-time strategy with depth direction a algorithm for multi-layer," in *Natural Language Processing, 2009. SNLP'09. Eighth International Symposium on*. IEEE, 2009, pp. 184–188.

[3] Gábor Rétvári, József J Bíró, and Tibor Cinkler, "On shortest path representation," *IEEE/ACM Transactions on Networking*, vol. 15, no. 6, pp. 1293–1306, 2007.

[4] Pablo Rodriguez-Mier, Manuel Mucientes, and Manuel Lama, "Automatic web service composition with a heuristic-based search algorithm," in *Web Services (ICWS), 2011 IEEE International Conference on*. IEEE, 2011, pp. 81–88.

[5] Ira Pohl, "Heuristic search viewed as path finding in a graph," *Artificial intelligence*, vol. 1, no. 3-4, pp. 193–204, 1970.

[6] Spiros N Agathos, Alexandros Papadogiannakis, and Vassilios V Dimakopoulos, "Targeting the parallella," in *European Conference on Parallel Processing*. Springer, 2015, pp. 662–674.

[7] Donald B Johnson, "Efficient algorithms for shortest paths in sparse networks," *Journal of the ACM (JACM)*, vol. 24, no. 1, pp. 1–13, 1977.

[8] Shimon Even and R Endre Tarjan, "Network flow and testing graph connectivity," *SIAM journal on computing*, vol. 4, no. 4, pp. 507–518, 1975.

[9] Zhanying Zhang and Ziping Zhao, "A multiple mobile robots path planning algorithm based on a-star and dijkstra algorithm," *International Journal of Smart Home*, vol. 8, no. 3, pp. 75–86, 2014.

[10] Hector Ortega-Arranz, Yuri Torres, Diego R Llanos, and Arturo Gonzalez-Escribano, "A new gpu-based approach to the shortest path problem," in *High performance computing and simulation (HPCS), 2013 international Conference on*. IEEE, 2013, pp. 505–511.

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

[12] Dhirendra Pratap Singh and Nilay Khare, "A study of different parallel implementations of single source shortest path algorithms," *International Journal of Computer Applications*, vol. 54, no. 10, 2012.

[13] Avi Bleiweiss, "Gpu accelerated pathfinding," in *Proceedings of the 23rd ACM SIGGRAPH / EUROGRAPHICS symposium on Graphics hardware*. Eurographics Association, 2008, pp. 65–74.

[14] David Šišlák, Přemysl Volf, and Michal Pěchouček, "Accelerated a* path planning," in *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*. International Foundation for Autonomous Agents and Multiagent Systems, 2009, pp. 1133–1134.

[15] In-Kyu Jeong, Jia Uddin, Myeongsu Kang, Cheol-Hong Kim, and Jong-Myon Kim, "Accelerating a bellman–ford routing algorithm using gpu," in *Frontier and Innovation in Future Computing and Communications*, pp. 153–160. Springer, 2014.

[16] Shijie Zhou, Charalampos Chelmis, and Viktor K Prasanna, "Accelerating large-scale single-source shortest path on fpga," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*. IEEE, 2015, pp. 129–136.

[17] Sandy Brand and Rafael Bidarra, "Multi-core scalable and efficient pathfinding with parallel ripple search," *computer animation and virtual worlds*, vol. 23, no. 2, pp. 73–85, 2012.

[18] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders, "A parallelization of dijkstra's shortest path algorithm," *Mathematical foundations of computer science 1998*, pp. 722–731, 1998.

[19] Pedro Martín, Roberto Torres, and Antonio Gavilanes, "Cuda solutions for the sssp problem," *Computational Science–ICCS 2009*, pp. 904–913, 2009.

[20] Han Cao, Fei Wang, Xin Fang, Hong-lei Tu, and Jun Shi, "Openmp parallel optimal path algorithm and its performance analysis," in *Software Engineering, 2009. WCSE'09. WRI World Congress on*. IEEE, 2009, vol. 1, pp. 61–66.

[21] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang, "Foregraph: Exploring large-scale graph processing on multi-fpga architecture," *Vertex*, vol. 1, no. 4, pp. 5, 2017.

[22] Jialiang Zhang, Soroosh Khoram, and Jing Li, "Boosting the performance of fpga-based graph processor using hybrid memory cube: A case for breadth first search," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 207–216.

[23] Xiaoyu Ma, Dan Zhang, and Derek Chiou, "Fpga-accelerated transactional execution of graph workloads," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 227–236.

[24] Leonardo Dagum and Ramesh Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[25] James A Ross, David A Richie, Song J Park, and Dale R Shires, "Parallel programming model for the epiphany many-core coprocessor using threaded mpi," *Microprocessors and Microsystems*, vol. 43, pp. 95–103, 2016.

[26] Linley Gwennap, "Adapteva: More flops, less watts," *Microprocessor Report*, vol. 6, no. 13, pp. 11–02, 2011.