A Review on Effective Cache-Supported Path Planning on Roads

Sailaja Yandrapati, M.Tech 2nd Year,Dept.Of CSE, VRS & YRN College of Engineering and Technology , Chirala,India
Damarla Sree Latha,Associate professor,Dept of CSE, VRS & YRN College of Engineering and Technology , Chirala,India

Abstract: In mobile navigation services, on-road path planning is a basic function that finds a route between a queried start location and a destination. While on roads, a path planning query may be issued due to dynamic factors in various scenarios, such as a sudden change in driving direction, unexpected traffic conditions, or lost ofGPS signals. In these scenarios, path planning needs to be delivered in a timely fashion. The requirement of timeliness is even more challenging when an overwhelming number of path planning queries is submitted to the server, e.g.,during peak hours. As the response time is critical to user satisfaction with personal navigation services, it is a mandate for the server to efficiently handle the heavy workload of path planning requests. To meet this need, we propose a system, namely, Path Planning by Caching (PPC), that aims to answer a new path planning query efficiently by caching and reusing historically queried paths (queried-paths in short). Unlike conventional cache-based path planning systems where a cached query is returned only when it matches completely with a new query, PPC leverages partially matched queried-paths in cache to answer part(s) of the new query. As a result, the server only needs to compute the unmatched path segments, thus significantly reducing the overall system workload.

Keywords: Spatial Database, Path Planning, Cache.

I. INTRODUCTION

Due to advances in big data analytics, there is a growing need for scalable parallel algorithms. These algorithms encompass many domains including graph processing, machine learning, and signal processing. However, one of the most challenging algorithms lie in graph processing. Graph algorithms are known to exhibit low locality, data dependence memory accesses, and high memory requirements. Even their parallel versions do not scale seamlessly, with bottlenecks stemming from architectural constraints, such as cache effects and on-chip network traffic. Path Planning algorithms, such as the famous Dijkstra's algorithm, fall in the domain of graph

p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 04 Issue 07
June 2017

analytics, and exhibit similar issues. These algorithms are given a graph containing many vertices, with some neighboring vertices to ensure connectivity, and are tasked with finding the shortest path from a given source vertex to a destination vertex. Parallel implementations assign a set of vertices or neighboring vertices to threads, depending on the parallelization strategy. These strategies naturally introduce input dependence. Uncertainty in selecting the subsequent vertex to jump to, results in low locality for data accesses. Moreover, threads converging onto the same neighboring vertex sequentialize procedures due to synchronization and communication. Partitioned data structures and shared variables ping-pong within on-chip caches, causing coherence bottlenecks. All these mentioned issues make parallel path planning a challenge. Prior works have explored parallel path planning problems from various architectural angles. Path planning algorithms have been implemented in graph frameworks. These distributed settings mostly involve large clusters, and in some cases smaller clusters of CPUs. However, these works mostly optimize workloads across multiple sockets and nodes, and mostly constitute either complete shared memory or message passing (MPI) implementations. In the case of single node (or single-chip) setup, a great deal of work has been done for GPUs are a few examples to name a few. These works analyze sources of bottlenecks and discuss ways to mitigate them. Summing up these works, we devise that most challenges remain in the fine-grain inner loops of path planning algorithms. We believe that analyzing and scaling path planning on single-chip setup can minimize the fine-grain bottlenecks. Since shared memory is efficient at the hardware level, we proceed with parallelization of the path planning workload for single-chip multi-cores. The single-chip parallel implementations can be scaled up at multiple nodes or clusters granularity, which we discuss.

Furthermore, programming language variations for large scale processing also cause scalability issues that need to be analyzed effectively so far the most efficient parallel shared memory implementations for graph processing are in C/C++. However, due to security exploits and other potential vulnerabilities, other safe languages are commonly used in mission-deployed applications. Safe languages guarantee dynamic security checks that mitigate vulnerabilities, and provide ease of programming. However, security checks increase memory and performance overheads. Critical sections of code, such as locked data structures,

p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 04 Issue 07
June 2017

now take more time to process, and hence communication and

synchronization overheads exacerbate for parallel implementations. Python is a subtle example of a safe language, and hence we analyze it's overheads in the context of our parallel path planning workloads. This paper makes the following contributions:

- We study sources of bottlenecks arising in parallel path planning workloads, such as input dependence and scalability, in the context of a single node, single chip setup.

- We analyze issues arising from safe languages, in our case Python, and discuss what safe languages need to ensure for seamless scalability.

- We plan to open source all characterized programs with the publication of this paper.

II.PATH PLANNING ALGORITHMS AND PARALLELIZATIONS

Dijkstra that is an optimal algorithm, is the de facto baseline used in path planning applications. However, several heuristic based variations exist that trade-off parameters such as parallelism and accuracy. Δ-stepping is one example

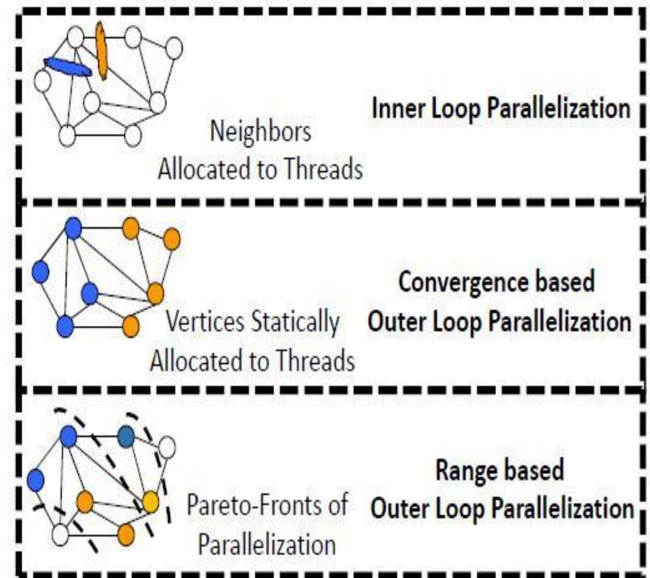

Fig.1. Dijkstra's Algorithm Parallelization's. Vertices Allocated to Threads Shown in Different Colors.

which classifies graph vertices and processes them in different stages of the algorithm. The A*/D* algorithms are another example that use aggressive heuristics to prune out computational work (graph vertices), and only visit vertices that occur in the shortest path. In order to maintain optimality and a suitable baseline, we focus on Dijkstra's algorithm in this paper.

A. Dijkstra's Algorithm and Structure

Dijkstra's algorithm consists of two main loops, an outer loop that traverses each graph vertex once, and an inner loop that traverses the neighboring vertices of the vertex selected by the outer loop. The most efficient generic implementation of

p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 04 Issue 07
June 2017

Dijkstra's algorithm utilizes a heap structure, and has a complexity of $O(E + V \log V)$. However, in parallel implementations, queues are used instead of heaps, to reduce overheads associated with re-balancing the heap after each parallel iteration. Algorithm 1 shows the generic pseudo-code skeleton for Dijkstra's algorithm. For each vertex, each neighboring vertex is visited and compared with other neighboring vertices in the context of distance from the source vertex (the starting vertex). The neighboring vertex with the minimum distance cost is selected as the next best vertex for the next outer loop iteration. The distances from the source vertex to the neighboring vertices are then updated in the program data structures, after which the algorithm repeats for the next selected vertex. A larger graph size means more outer loop iterations, while a large graph density means more inner loop iterations. Consequently, these iterations translate into parallelism, with the graph's size and density dictating how much parallelism is exploitable. We discuss the parallelizations in subsequent subsections and show examples in Fig 1.

Algorithm 1 Dijkstra's Algorithm Skeleton

```
1: <<< Initialize D, Q >>>
2: for (Each vertex u) do                          ▷ Outer Loop
3:     for (Each Edge of u) do                     ▷ Inner Loop
4:
5:         1. Calc. dist. from Current vertex to each neighbor
6:         2. Check for next best vertex u among neighbors
```

B. Inner Loop Parallelization

The inner loop in Algorithm 1 parallelizes the neighboring vertex checking. Each thread is given a set of neighboring vertices of the current vertex, and it computes a local minimum and updates that neighboring vertex's distance. A master thread is then called to take all the local minimums, and reduce to find a global minimum, which becomes the next best vertex to jump to in the next outer loop iteration. Barriers are required between local minimum and global minimum reduction steps as the global minimums can only be calculated when the master thread has access to all the local minimums. Parallelism is therefore dependent on the graph density, i.e. the number of neighboring vertices per vertex. Sparse graphs constitute low density, and therefore cannot scale with this type of parallelization. Dense graphs having high densities are expected to scale in this case.

C. Outer Loop Parallelization

The outer loop parallelization strategy partitions the graph vertices among threads, depicted in Algorithm 1. Each thread runs inner loop iterations over its vertices, and updates the distance arrays in the process. However, atomic clocks over shared memory are required to update vertex distances, as vertices may be sharing neighbors in different threads.

1. Convergence Outer Loop Parallelization: The convergence based outer loop statically partitions the graph vertices to threads. Threads work on their allocated chunks independently, update tentative distance arrays, and update the final distance array once each thread completes work on its allocated vertices. The algorithm then repeats, until the final distance arrays stabilize, where the stabilization sets the convergence condition. Significant redundant work is involved as each vertex is computed upon multiple times during the course of this algorithm's execution.

2. **Ranged based Outer Loop Parallelization:** The range based outer loop parallelization opens pare to fronts on vertices in each iteration. Vertices in these fronts are equally divided amongst threads to compute on, however, atomic clocks are still required due to vertex sharing. As pare to fronts are intelligently opened using the graph connectivity, a vertex can be safely relaxed just once during the course of the algorithm. Redundant work is therefore mitigated, while maintaining significant parallelism. However, as initial and final pare to fronts contain less vertices, limited parallelism is available during the initial and final phases of the algorithm. Higher parallelism is available during the middle phases of the algorithm. This algorithm's available parallelism hereto follows a normal distribution, with time on the x-axis.

III. METHODS

This section outlines multicore machine configuration and programming methods used for analysis. We also explain the graph structures used for the various path planning workloads.

A. Many-core Real Machine Setups

We use Intel's Core i7-4790 has well processor to analyze our workloads. The machine has 4 cores with 2-way hyper threading; an 8MB shared L3 cache, and a 256KB per-core private L2 cache.

B. Metrics and Programming Language Variations

We use C/C++ to create efficient implementations of our parallel path planning algorithms. We use the p thread parallel library, and enforce gcc/g++ compiler -O3 optimizations to ensure maximum performance. The p thread library is preferred over Open MP to allow for the use of lower level synchronization primitives and optimizations. For Python implementations, we use both threading and multiprocessing libraries to parallelize programs, with Python3 as the language version. We use these two parallelization paradigms to show the limitations and

International Journal of Research
Available at https://edupediapublications.org/journals
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 04 Issue 07
June 2017

shortcomings in parallel safe language paradigms. For each simulation run, we measure the Completion Time, i.e., the time in parallel region of the benchmark. The time is measured just before threads/ processes are spawned/forked, and also after they are joined, after which the time difference is measured as the Completion Time. To ensure an unbiased comparison to sequential runs, we measure the Completion Time for only the parallelized code regions. These parallel completion times are compared with the best sequential implementations to compute speedups, as given by Eq (1). Values greater than 1 show speedups, while values between 0 and 1 depict slowdowns in addition to performance, memory effects in a specific parallelization strategy also affect scalability. To evaluate cache effects, the cache accesses are therefore measured using hardware performance counters.

Speedup = Sequential Time/Parallel Time (1)

C. Graph Input Data Sets and Structures

TABLE I. Synthetic Graphs Used for Evaluation

Graph Dataset	Vertices	Edges
Sparse Graph	1,048,576	16,777,216
California Road Network	1,965,206	2,766,607
Pennsylvania Road Network	1,088,092	1,541,898
Texas Road Network	1,379,917	1,921,660
Dense Graph	16,384	134,217,728

Synthetic graphs and datasets are generated using a modified version of the GT Graph generator, which uses RMAT graphs from Graph500. We also use real world graphs from the Stanford Large Network Dataset Collection (SNAP), such as road networks. These are undirected graphs, with a degree irregularly varying from 1 to 4. Generated graphs have random edge weights and connectivity. All graphs are represented in the form of adjacency lists, with one data structure containing the edge weights, and another for edge connectivity, and all values represented by integers. Both sparse and dense graphs are used to analyze parallelizations across different input types, as shown in Table I. We also scale synthetic graphs from 16K vertices to 1M vertices, and the graph density from 16 up to 8K connections per vertex.

IV. EXPERIMENTS

A. Dataset

We conduct a comprehensive performance evaluation of the proposed PPC system using the road network dataset of Seattle obtained from ACM SIGSPATIAL Cup 2012. The dataset has 25,604 nodes and 74,276 edges. For the query log, we obtain the Points-of-interest (POIs) in Seattle from. Next, we randomly select pairs of nodes from these POIs as the source and destination nodes for path

International Journal of Research
Available at https://edupediapublications.org/journals
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 04 Issue 07
June 2017

planning queries. Four sets of query logs with different distributions are used in the experiments: QL_{normal} and $QL_{uniform}$ are query logs with normal and uniform distributions, respectively. $QL_{central}$ is used to simulate a large-scale event (e.g., the Olympics or the World Cup) held in a city. $QL_{direction}$ is to simulate possible driving behavior (e.g., changing direction) based on a random walk method described as follows. We firstly randomly generate a query to be the initial navigational route. Next, we randomly draw a probability to determine the chance for a driver to change direction. The point of direction change is treated as a new source. This process is repeated until the anticipated numbers of queries are generated. The parameters used in our experiments are shown in Table 2.

B. Cache-Supported System Performance

1. Cache versus Non-Cache

The main idea of a cache-supported system is to leverage the cached query results to answer a new query. Thus, we are interested in finding how much improvement our path planning system achieves over a conventional non-cache system. We generate query sets of various sizes to compare the paths generated by our PPC and A* algorithm. The performance is evaluated by two metrics: a) Total number of visited nodes: it counts the number of nodes visited by an algorithm under comparison in computing a path, and b) Total query time: it is the total time an algorithm takes to compute the path. By default, we apply 3,000 randomly generated queries to warm up the cache before proceeding to measure experimental results. Table 5 summarizes the statistics of the above two metrics with five different sized query sets. From the statistics we find that our cache-supported algorithm greatly reduces both the total visited nodes and the total query time. On average, PPC saves 23 percent of visiting nodes and 30.22 percent of response time compared with a non-cache system.

TABLE II. Experimental Parameters

Parameter	Default	Value
Grid size	2 km	0.5 ~ 5 km
Cache size	5k	1k ~ 10k
#Quries	5k	0.5k ~ 5k
Data sets	QL_{normal}	QL_{normal}, $QL_{uniform}$, $QL_{central}$, $QL_{direction}$

#Query	#Nodes		Time (ms)	
	PPC	A*	PPC	A*
1k	80,087	107,856	14,190,670	19,973,996
2k	157,162	215,459	27,869,212	39,889,166
3k	230,185	319,231	41,493,092	59,983,844
4k	328,879	419,345	55,139,411	79,937,684
5k	437,362	501,312	69,843,232	100,037,461

TABLE III. Performance Comparison between PPC and the Non-Cache Algorithm

V. CONCLUSION

We propose a system, namely, Path Planning by Caching (PPC), that aims to answer a new path planning query efficiently by caching and reusing historically queried paths (queried-paths in short).The proposed system consists of three main components: (i) PPattern Detection, (ii) Shortest Path Estimation, and (iii) Cache Management. Given a path planning query, which contains a source location and a destination location, PPC firstly determines and retrieves a number of historical paths in cache, called PPatterns, that may match this new query with high probability.The idea of PPatterns is based on an observation that similar starting and destination nodes of two queries may result in similar shortest paths (known as the path coherence property). In the component PPatern Detection, we propose a novel probabilistic model to estimate the likelihood for a cached queried-path to be useful for answering the new query by exploring their geospatial characteristics. To facilitate quick detection of PPatterns, instead of exhaustively scanning all the queried paths in cache, we design a grid-based index for the PPattern Detection module. Based on these detected PPatterns, the Shortest Path Estimation module constructs candidate paths for the new query and chooses the best (shortest) one. In this component, if a PPattern perfectly matches the query, we immediately return it to the user; otherwise, the server is asked to compute the unmatched path segments between the PPattern and the query. Because the unmatched segments are usually only a smaller part of the original query, the server only processes a "smaller subquery", with a reduced workload. Once we return the estimated path to the user, the Cache Management module is triggered to determine which queried-paths in cache should be evicted if the cache is full. An important part of this module is a new cache replacement policy which takes into account the unique characteristics of road networks.In this paper, we provide a new framework for reusing the previously cached query results as well as an effective algorithm for improving the query evaluation on the server.

VI. REFERENCES

[1]Ying Zhang, Member, IEEE, Yu-Ling Hsueh, Member, IEEE, Wang-Chien Lee, Member, IEEE, and Yi-Hao Jhang, "Efficient Cache-Supported Path Planning on Roads", IEEE Transactions on Knowledge and Data Engineering, Vol. 28, No. 4, April 2016.

[2]H. Mahmud, A. M. Amin, M. E. Ali, and T. Hashem, "Shared execution of path queries on road networks," Clinical Orthopaedics Related Res., vol. abs/1210.6746, 2012.

[3]L.Zammit, M.Attard, and K. Scerri, "Bayesian hierarchical modelling of traffic flow - With application to Malta's road network," in Proc. Int. IEEE Conf. Intell. Transp. Syst., 2013, pp. 1376–1381.

[4]S. Jung and S. Pramanik, "An efficient path computation model for hierarchically structured topographical road maps," IEEE Trans. Knowl. Data Eng., vol. 14, no. 5, pp. 1029–1046, Sep. 2002.

[5]E. W. Dijkstra, "A note on two problems in connation with graphs," Num. Math., vol. 1, no. 1, pp. 269–271, 1959.

[6]U. Zwick, "Exact and approximate distances in graphs – a survey," in Proc. 9th Annu. Eur. Symp. Algorithms, 2001, vol. 2161, pp. 33–48.

[7]A. V. Goldberg and C. Silverstein, "Implementations of Dijkstra's algorithm based on multi-level buckets," Network Optimization, vol. 450, pp. 292–327, 1997.

[8]P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," IEEE Trans. Syst. Sci. Cybern., vol. SSC-4, no. 2, pp. 100–107, Jul. 1968.

[9]A. V. Goldberg and C. Harrelson, "Computing the shortest path: A search meets graph theory," in Proc. ACM Symp. Discr. Algorithms, 2005, pp. 156–165.

[10]R. Gutman, "Reach-based routing: A new approach to shortest path algorithms optimized for road networks," in Proc. Workshop Algorithm Eng. Experiments, 2004, pp. 100– 111.

[11]A. V. Goldberg, H. Kaplan, and R. F. Werneck, "Reach for A*: Efficient point-to-point shortest path algorithms," in Proc. Workshop Algorithm Eng. Experiments, 2006, pp. 129– 143.

[12]S. Jung and S. Pramanik, "An efficient path computation model for hierarchically structured topographical road maps," IEEE Trans. Knowl. Data Eng., vol. 14, no. 5, pp. 1029–1046, Sep. 2002.

Author's Profile:

Sailaja Yandrapati received B.Tech degree in Computer Science and Engineering and pursuing M.Tech in Computer Science and Engineering from VRS & YRN College of Engineering and Technology ,Dept of CSE,Chirala,Prakasam,India.

Damarla Sree Latha working as Associate professor in VRS & YRN College of Engineering and Technology ,Dept of CSE,Chirala,Prakasam,India.