Efficient Parallelization of Path Planning Workload on Single-chip Shared-memory Multicores

Masab Ahmad, Kartik Lakshminarasimhan, Omer Khan
University of Connecticut, Storrs, CT, USA
{masab.ahmad, kartik.lakshminarasimhan, khan}@uconn.edu

***Abstract*—Path planning problems greatly arise in many applications where the objective is to find the shortest path from a given source to destination. In this paper, we explore the comparison of programming languages in the context of parallel workload analysis. We characterize parallel versions of path planning algorithms, such as the Dijkstra's Algorithm, across C/C++ and Python languages. Programming language comparisons are done to analyze fine grain scalability and efficiency using a single-socket shared memory multicore processor. Architectural studies, such as understanding cache effects, are also undertaken to analyze bottlenecks for each parallelization strategy. Our results show that a right parallelization strategy for path planning yields scalability on a commercial multicore processor. However, several shortcomings exist in the parallel Python language that must be accounted for by HPC researchers.**

I. Introduction

Due to advances in big data analytics, there is a growing need for scalable parallel algorithms [24]. These algorithms encompass many domains including graph processing [14], machine learning [5], and signal processing [19]. 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 [4]. 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 [6], fall in the domain of graph 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 [9]. 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 [4]. 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, such as Galios [16], Ligra [20], and GraphLab [13]. 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. Pannotia [4] and [14] 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 multicores. The single-chip parallel implementations can be scaled up at multiple nodes or clusters granularity, which we discuss in Section V.

Furthermore, programming language variations for large scale processing also cause scalability issues that need to be analyzed effectively [16]. So far the most efficient parallel shared memory implementations for graph processing are in C/C++, such as Galios [16] and Ligra [20]. 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 [18]. However, security checks increase memory and performance overheads. Critical sections of code, such as locked data structures, now take more time to process, and hence communication and synchronization overheads exacerbate for parallel implementations [2]. 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 [6] 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 [15] is one example

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

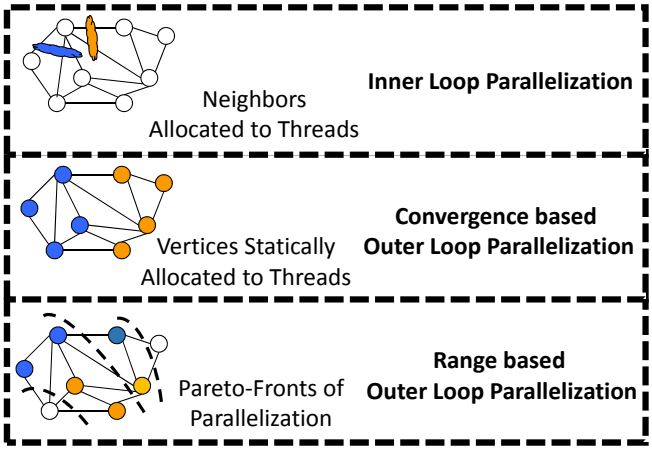

Fig. 1. Dijkstra's Algorithm Parallelizations. Vertices allocated to threads shown in different colors.

which classifies graph vertices and processes them in different stages of the algorithm. The A*/D* algorithms [22] 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 Dijkstra's algorithm utilizes a heap structure [7], and has a complexity of $O(E + VlogV)$. 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 pseudocode 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 [8]. 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 Parallelizations

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 locks 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 [9]. 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 pareto fronts on vertices in each iteration [4]. Vertices in these fronts are equally divided amongst threads to compute on, however, atomic locks are still required due to vertex sharing. As pareto 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 pareto 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 Haswell processor to analyze our workloads [1]. The machine has 4 cores with 2-way hyper-threading, an $8MB$ shared L3 cache, and a $256KB$ per-core private L2 cache.

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

B. Metrics and Programming Language Variations

We use C/C++ to create efficient implementations of our parallel path planning algorithms. We use the `pthread` parallel library, and enforce gcc/g++ [21] compiler `-O3` optimizations to ensure maximum performance. The `pthread` library is preferred over `OpenMP` 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 [17]. We use these two parallelization paradigms to show the limitations and 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 [1].

$$Speedup = \frac{SequentialTime}{ParallelTime} \quad (1)$$

C. Graph Input Data Sets and Structures

Synthetic graphs and datasets are generated using a modified version of the GTGraph generator [3], which uses RMAT graphs from Graph500 [23]. We also use real world graphs form the Stanford Large Network Dataset Collection (SNAP), such as road networks [12]. 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.

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

IV. CHARACTERIZATION

In this section we show results for input graph variations and safe language usages for parallel Dijkstra. All results use C/C++ implementations unless otherwise stated.

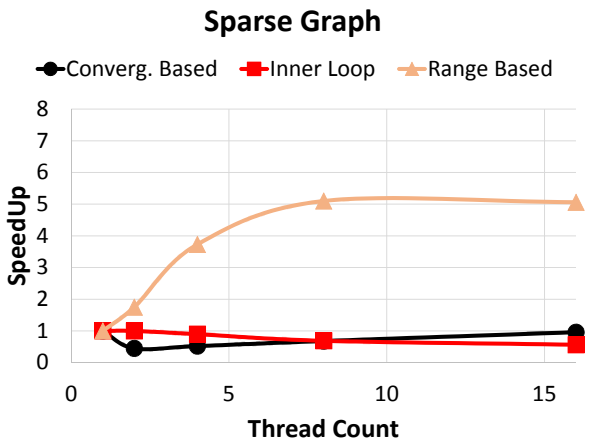

Fig. 2. Input Dependence in Dijkstra's Parallelizations for Sparse Graph (1M vertices, 16M edges).

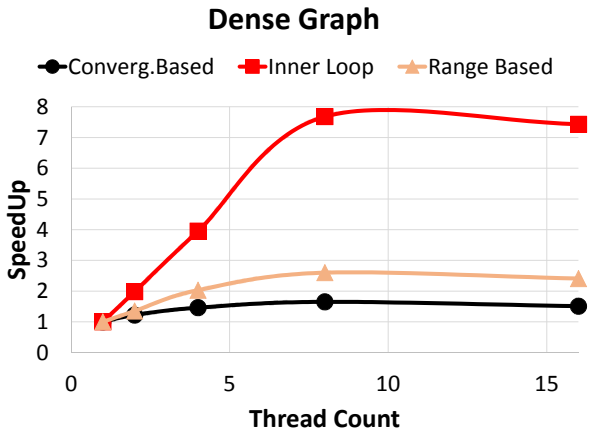

Fig. 3. Input Dependence in Dijkstra's Parallelizations for Dense Graph (16K vertices, 134M edges).

A. Input dependence and Scalability

We run all parallelizations for various input graph types (sparse, dense, and road networks). Figures 2 and 3 depict the speedup/slowdown for all three parallelization strategies for both sparse and dense graph types, while speedup/slowdown for the road network inputs are depicted in Table II. Graphs shown in Table I are used.

For sparse input graphs, we observe that the range based parallelization performs best, giving a maximum speedup of around $5\times$ at 8 threads. Range based outer loop parallelization scales because the outer loop has more available parallelism due to large vertex count. The convergence based outer loop parallelization does not perform well due to more redundant work in extra edge relaxations. The inner loop parallelization also does not scale because it cannot exploit much parallelism in the inner loop of sparse graphs, where communication overheads due to barrier usages overcome computation. Similar trends are seen in the cases of road networks.

In the case of dense graphs, as seen in Figure 3, the inner loop parallelization performs optimally. This occurs due to high amount of parallelism available in the inner loop for dense graphs. Convergence based outer loop and range

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

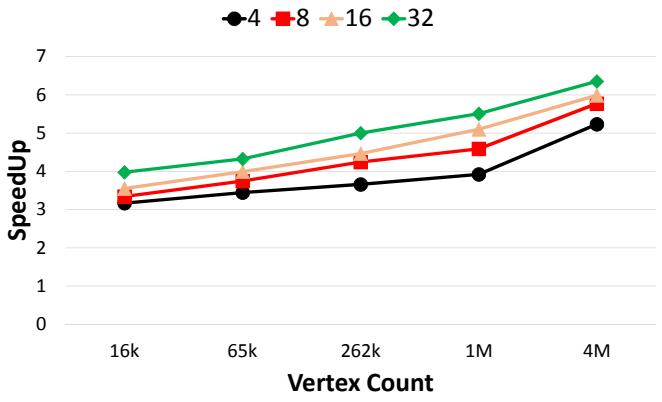

Fig. 4. Vertex and Edge Scalability for Sparse Graphs. Ranged based Parallelization used due to its better performance. Only speedups at best thread counts shown.

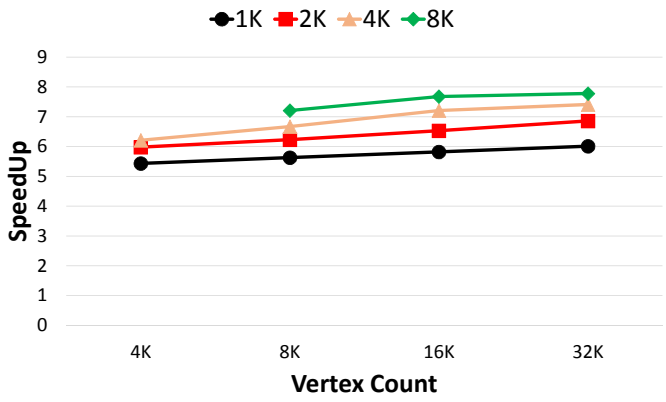

Fig. 5. Vertex and Edge Scalability for Dense Graphs. Inner Loop Parallelization used due to its better performance. Only speedups at best thread counts shown.

based outer loop parallelizations do not scale on dense graphs. This is because atomic locks over vertices are accessed much more often, and increasing graph density exacerbates associated communication costs, even more to produce bottlenecks. Intuitively, in dense graphs, single node setup can easily exploit parallelism over shared memory due to close proximity of the input and read/write data using inner loop parallelization.

TABLE II. ROAD NETWORK RESULTS. ALL SPEEDUPS RELATIVE TO DIJKSTRA SEQUENTIAL (AT 8 THREADS). SLOWDOWN INDICATED BY VALUES LESS THAN 1

Algorithm	Texas	Pennsylvania	California
Inner Loop	0.53	0.55	0.51
Converg. based Outer	0.24	0.21	0.24
Range based Outer	3.14	3.30	3.17

Insights: Path planning algorithms also exhibit input dependence. Such dependencies lead to different parallelization strategies performing optimally for different input graph types.

B. Edge and Vertex scalability

We also varied both vertex and edge counts in input graphs to determine whether shared memory parallelizations change performance characteristics. We found that the range based outer loop parallelization remains best for sparse graphs, while the inner loop parallelization performs optimally for dense graphs. Figure 4 shows the vertex and edge variation results for the range based parallelization for sparse graphs. The Best speedup for each combination is reported, with the thread count for this best speedup remaining at 8. Scalability is seen for both vertex and edge augmentations. Similar scalability results are seen for dense graphs using the inner loop parallelization in Figure 5.

Insights: Given the optimal parallelization strategy, parallel path planning algorithms scale for larger graphs.

C. Understanding Cache Effects

To understand cache effects in the three C/C++ path planning parallelization strategies, we use a sparse graph with $1M$ vertices and 16 edges per vertex (the size of this graph is much larger than the $8MB$ on-chip last-level L3 cache). Cache

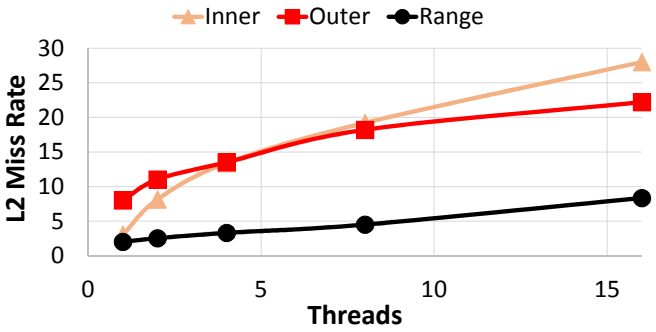

Fig. 6. L2 Cache Miss Rates for the three Parallelization Strategies.

miss rates are measured for both L2 and L3 caches, dividing the total number of misses with the total number of cache accesses. Figure 6 and Figure 7 show the miss rates for the L2 and L3 caches respectively, with L2 mises signifying mainly sharing misses, and L3 misses showing mainly size and reuse constraints in the algorithm. The inner loop parallelization does not scale in the case of a sparse graph, and thus it's miss rate increases with the increase in thread count due to higher sharing of cache lines in the L2 cache. Although both convergence based outer loop and the range based outer loop scale, the convergence based strategy has a higher L3 cache miss rate due to its higher memory requirements and a higher L2 miss rate due to data sharing. Miss rates for the range

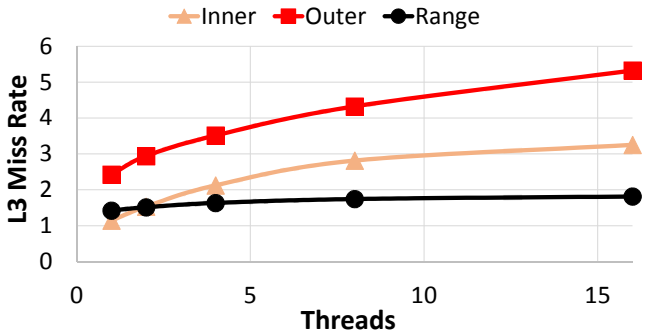

Fig. 7. L3 Cache Miss Rates for the three Parallelization Strategies.

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

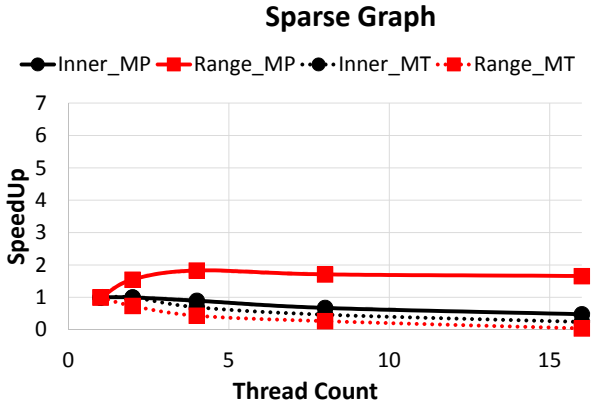

Fig. 8. Scalability results for Python Implementations on Sparse Graphs. MT depicts multithreading, and MP depicts multiprocessing. Speedups are relative to sequential Python implementations.

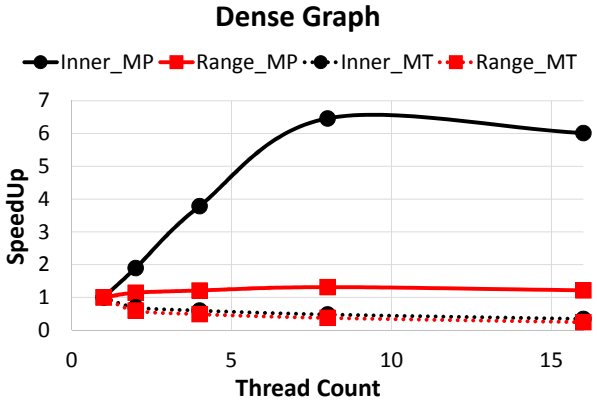

Fig. 9. Scalability results for Python Implementations on Dense Graphs. MT depicts multithreading, and MP depicts multiprocessing. Speedups are relative to sequential Python implementations.

based version increase at a smaller rate, mainly because it does less redundant work, and thus requiring less overall cache accesses. Furthermore, higher locality in the ranged based strategy reduces data sharing, and hence has a lower L2 cache miss rate.

Insights: Parallelization strategies need to exploit the on-chip cache hierarchy to scale.

D. Python vs. C/C++

1) Scalability in Python: Safe languages, such as Python, are generally preferred over C/C++ implementation in application deployment. The reason being that C/C++ is vulnerable to security exploits, and does not provide ease-of-programming model. We therefore study parallel implementations in Python to analyze associated overheads. As we already know that the convergence based outer loop parallelization is inefficient compared to others, we only compare inner loop and the range based outer loop parallelizations in Python. Parallel versions of Dijkstra were programmed in Python using both `threading` as well as `multiprocessing` libraries. Figures 8 and 9 show the speedup/slowdown for both sparse and dense graphs.

The `threading` library in Python does not provide any speedup over sequential implementations. The reason for this

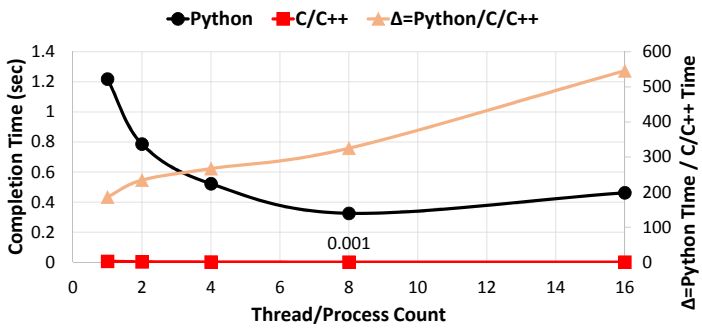

Fig. 10. Python vs. C/C++ Range based Parallelization. Sparse Graph with 16K vertices and 262K edges used. Δ depicts slowdown of Python vs. C/C++

is that Python's interpreter sequentializes threaded programs through the Global Interpreter Lock (GIL). According to Python's manual [17], the GIL Lock is necessary because CPython's memory management is not thread-safe. Variations of Python, such as IronPython, do exist that do not have the GIL Lock problem. However these are specialized interpreters written in other languages such as Java, and have other additional overheads and performance variations.

Multiprocessing implementations, however, do show some scalability, as shown in Figures 8 and 9. For sparse graphs, the range based parallelization, using `multiprocessing`, gives up to 2× speedup for sparse, while the inner loop gives up to 6.5× speedup for dense graphs. Note that `multiprocessing` implementations communicate through the operating system whenever a process needs to read/write shared data, incurring additional overheads.

2) Python Slowdown vs. C/C++: We compared completion times of Python multiprocessing implementations with C/C++ implementations for sparse graphs. Various trends were seen, with seemingly no correlation with each other, showing that slowdowns of Python are highly input and application dependent. Slowdowns ranged between 20x to 700x for different graphs. We take a small sparse graph as an example having $16K$ vertices and $262K$ edges and compare the range based parallelizations for both languages, Figure 10 shows the results. The best completion time of C/C++ is seen to be 0.001 seconds, while that of Python is 0.325 seconds, and the slowdown (Δ) is seen to be 325x. The Δ values increase as C/C++ is able to exploit more parallelism than Python, mainly due to operating system overheads in multiprocessing.

Insights: Safe languages, such as Python, do exhibit scalability with reference to their own sequential version. However, they do have overheads compared to more efficient languages, such as C/C++. Parallel programming paradigms such as multi-threading are not fully supported in Python, and therefore users have to resort to multi-processing, which incurs additional operating system overheads for shared data.

V. Discussion

Here we discuss what improvements are needed in architectures and programming languages for path planning workloads.

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

A. Python for Parallel Programming

As evident from our characterization, the GIL lock in Python is a primary concern for scalability. We argue that the Python interpreter must have a choice for the programmer who can choose to release the GIL lock for ensuring high performance. More aggressive means to improve performance include options to control security features as well. For example, turning off dynamic bound checks will improve performance, since it reduces overheads associated with critical sections of code.

B. Scalability on Multiple Nodes and Clusters

From our analysis, we show that shared memory parallelizations scale in a single-chip/single-node setup, for both Python and C/C++ language implementations. We therefore argue that large cluster based setups involving large numbers of nodes need to improve fine grain parallelization strategies. Instead of MPI only implementations, a hybrid MPI–Shared-memory approach should be used. Cores connected over single sockets can benefit from shared memory, exploiting both ease-of-programming and low latency data accesses for communication. For cores connected over multiple sockets, such as multiple nodes, the MPI approach can be used to communicate data. In the case of path planning, this can be thought of as allocating outer loop iterations (e.g., partitioning the graph) across multiple nodes. This can potentially solve both load imbalance and scalability problems in not only path planning workloads, but other workloads such as graph analytics and machine learning. Prior works such as [11], and [10] have proposed such models for both single and multi node setups. However these have not fully been validated with graph algorithms. In summary, hybrid MPI – Shared memory models need to be explored to improve scalability in distributed workloads.

VI. Conclusion

Path planning is an important graph workload, and is used ubiquitously in various real world applications. While many studies have been done on distributed systems, limited comparative studies have been done on its parallelizations in single node setups. In this paper, we study different parallelizations of Dijkstra's algorithm for single node machines, and analyze algorithmic and architectural bottlenecks for each parallelization strategy. We show that cache sizes and algorithmic data sharing contributes greatly to scalability. We also compare safe languages, such as Python, to more efficient implementations in C/C++. Our results show that shared memory parallelization of path planning workload scales on single node setup. We also discuss what limitations safe languages have and what should be done to improve them.

References

[1] "Intel core i7-4790, http://ark.intel.com/products/80806/intel-core-i7-4790-processor-8m-cache-up-to-4_00-ghz," 2014.

[2] M. Ahmad, S. K. Haider, F. Hijaz, M. van Dijk, and O. Khan, "Exploring the performance implications of memory safety primitives in many-core processors executing multi-threaded workloads," 2015.

[3] D. A. Bader and K. Madduri, "Gtgraph: A synthetic graph generator suite," 2006.

[4] S. Che, B. Beckmann, S. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, Sept 2013, pp. 185–195.

[5] T. Condie, P. Mineiro, N. Polyzotis, and M. Weimer, "Machine learning for big data," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 939–942.

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

[7] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM*, vol. 34, no. 3, pp. 596–615, Jul. 1987.

[8] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, Dec 2001, pp. 3–14.

[9] P. Harish, V. Vineet, and P. J. Narayanan, "Large graph algorithms for massively multithreaded architectures," *Technical Report Number IIIT/TR/2009/74*, 2009.

[10] F. Hijaz, B. Kahne, P. Wilson, and O. Khan, "Accelerating communication in single-chip shared memory many-core processors," in *Sixth Annual Boston Area Architecture Workshop*, 2015.

[11] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "Mpi + mpi: a new hybrid approach to parallel programming with mpi plus shared memory," *Computing*, vol. 95, no. 12, pp. 1121–1136, 2013.

[12] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," 2008.

[13] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[14] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 117–128.

[15] U. Meyer and P. Sanders, "δ-stepping: A parallel single source shortest path algorithm," in *Proceedings of the 6th Annual European Symposium on Algorithms*, ser. ESA '98. London, UK, UK: Springer-Verlag, 1998, pp. 393–404.

[16] D. Nguyen, A. Lenharth, and K. Pingali, "Deterministic galois: On-demand, portable and parameterless," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 499–512.

[17] G. Rossum, "Python reference manual," Amsterdam, The Netherlands, The Netherlands, Tech. Rep., 1995.

[18] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J.Sel. A. Commun.*, vol. 21, no. 1, pp. 5–19, Sep. 2006.

[19] S. Sakr, A. Liu, and A. G. Fayoumi, "The family of mapreduce and large-scale data processing systems," *ACM Comput. Surv.*, vol. 46, no. 1, pp. 11:1–11:44, Jul. 2013.

[20] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: ACM, 2013, pp. 135–146.

[21] R. M. Stallman and G. DeveloperCommunity, *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. Paramount, CA: CreateSpace, 2009.

[22] A. Stentz, "Optimal and efficient path planning for partially-known environments," in *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, May 1994, pp. 3310–3317 vol.4.

[23] K. Ueno and T. Suzumura, "Highly scalable graph search for the graph500 benchmark," in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '12. New York, NY, USA: ACM, 2012, pp. 149–160.

[24] X. Wu, X. Zhu, G.-Q. Wu, and W. Ding, "Data mining with big data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 1, pp. 97–107, 2014.

978-1-4673-9286-0/15/$31.00 ©2015 IEEE