Parallel A* Graph Search

Ariana Weinstock[1] and Rachel Holladay[1]

Abstract— **Given the ubiquity of graph search and the rise of parallel computing, we implemented and explored a parallel A* graph search. Taking inspiration from Hash Distributed A* (HDA*) [1], we implemented the search technique with message passing and shared address space memory models. We present a detailed comparison and analysis on performance and quality along with a discussion of how we leveraged and investigated the subtleties of parallelism.**

I. Introduction

Graph search is a powerful tool that permeates across a variety of applications. Graph search is ubiquitous, used in everything from robotic motion planning [2], [3] to flight trajectory generation [4] to biological signal tracking [5]. Given its widespread use, we are constantly pushing to make search faster and more efficient. One of the most popular informed graph search algorithms is A* [6]. With the increasing advent of parallel computing, our goal was to implement a parallel A* graph search algorithm.

A* is an informed, best-first search for finding the minimum cost path on weighted graphs. The search is informed via it's heuristic, a problem specific function that estimates the distance to the goal from a particular vertex. The search maintains an open and closed set of vertices to search from and those who have been searched, respectively.

Parallelism is usually injected into A* by parallel handling of the open and closed sets, and we detailed the variety of methods explored in our related work, Sec. II. A key difficulty is managing contention and distribution of work along with efficient termination conditions.

For our parallelization, we dove into HDA* (Hash Distributed A*), a distributed search approach that schedules work among processors based on a hash function on the search space [1]. Every time a vertex is expanded, we hash its index to determine which processor will evaluate it. For the simple graph shown in Fig.1, each set of colored vertices would be evaluated by a different processor.

Using Boost Graph as our graphing library [7], we implemented HDA* under a message passing model, powered by Open MPI [8], and a shared memory model, powered by OpenMP [9]. We performed a

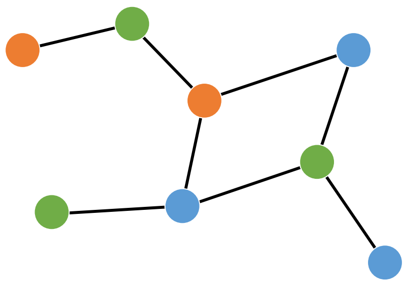


Fig. 1: HDA* distributes work by hashing vertices to different processors. Hence this simple graph could distribute all vertices of a color to a particular processor.

detailed analysis comparing our implementations, exploring the effects of algorithmic decisions and problem size.

Our work shows that the message passing implementation is better suited to HDA*, achieving better overall results. We were able to observe interesting effects with respect to memory implications and data coherence.

We begin by reviewing related work (Sec. II). Next we outline sequential A*, HDA* and our methods of parallelization (Sec. III). We then outline our experimental design (Sec. IV) and a summary of our most interesting subsequent results (Sec. V). A full report of all of our results are given at the end, as an Appendix. We conclude with a discussion and look towards the future (Sec. VI).

II. Related Work

Interest in parallel search was sparked early on in the rise of parallel computing, dating back to the mid-1990s [10]. PLA* (Parallel Local A*) presented a distributed? A* search, where each processor had a local closed and open set [11], while PRA* (Parallel Retracting A*) utilized SIMD processing and introduced a retracting algorithm [12].

Moving into the more recent age of parallel algorithms we have seen an increased interest in parallelizing A* within the last six or seven years [13]. PBNF (Parallel Best NBlock First) builds from PRA*, combining it with insights from work on parallel structured duplicate detection and powering this parallelism through a pthreads implementation [14]. Meanwhile PNBA* (Parallel New Bidirectional A*) presents a bidi-

[1]Ariana Weinstock and Rachel Holladay are with the School of Computer Science, Carnegie Mellon University. `ayw@andrew.cmu.edu, rmh@andrew.cmu.edu`

rectional search; however this is limited to only utilizing two processors, since the two sides of the search are handled by unique processors [15].

Leveraging the newer hardware, A* has been parallelized on CUDA, where path computation was spread across multiple threads [16]. There has been very recent developments on A* for GPUs, where the GPU allows for faster computation of expensive heuristic functions and a higher bandwidth for memory access [17].

While there are parallel domain specific languages for graph computations, such as GraphLab [18] and GraphX [19], we elected not to explore these because we want to have the freedom to manipulate the graph directly.

Considering Boost Graph is our graphing library, we briefly explored Parallel Boost Graph [20], but found the package to be too early in the development stages to be stably used.

III. A* and Parallelism

We first describe sequential A* and the opportunities for parallelism that we exploited. We overview HDA* and our adaptions to message passing and shared address space. HDA*, when first presented, discussed both of these methods, however the main focus of the paper was on the message passing implementation, with only a brief aside to the shared address space version. Additionally we discuss various interesting algorithmic details such as path reconstruction and the termination condition.

A. Sequential A*

Sequential A* takes in a graph G, a source vertex s, a destination vertex d, and a heuristic function h (Algorithm 1). The algorithm utilizes an openSet of vertices to be expanded and a closedSet of vertices which have already been expanded.

We start our search at s, searching towards d. At each step, we examine the vertex n with the lowest $f(n)$ where $f(n)$ is given by:

$$f(n) = g(n) + h(n)$$

Where $g(n)$ is the cost to come, the cost of the path from the start vertex to n, and $h(n)$ is the cost to go, the estimated cost of the path from n to the end vertex. In our methods, we implement $g(n)$ as a lookup into a costToCome table. When we expand n, we move it from the openSet to the closedSet, and add all of the neighbors of n, that are not in the closedSet, to the openSet. This allows a vertex to be added to the openSet more than once.

A heuristic is admissible iff it never overestimates the goal. In short, $h(v) \leq c(v, d)$ for all vertices v, where h is the heuristic function, d is the destination vertex, and $c(v, d)$ is the length of the shortest path from v to d. A heuristic is consistent iff it follows the triangle inequality. That is to say that $h(u) \leq w(u, v) + h(v)$, where $w(u, v)$ is the weight of the edge from u to v.

Algorithm 1 Sequential A* Search

```
1: Given: Graph $G$, Source $s$, Destination $d$, Heuristic $h$
2: openSet := $\{s\}$
3: closedSet := $\{\}$
4: costToCome := [INT_MAX if $v \neq s$ else $0 | v \in V$]
5: procedure WHILE !OPENSET.ISEMPTY
6:     curr = OPENSET.POP
7:     if curr = $d$ then
8:         RECONSTRUCTPATH(costToCome, curr)
9:     if $curr \in$ closedSet then
10:        CONTINUE
11:    CLOSEDSET.ADD(current)
12:    procedure FOR $n \in$ NEIGHBORS(CURR)
13:        if $n \in$ closedSet then
14:            CONTINUE
15:        gcost = costToCome[curr] + WEIGHT(curr,n)
16:        fcost = gcost + H(n)
17:        if gcost > costToCome[n] then
18:            CONTINUE
19:        costToCome[n] = gcost
20:        OPENSET.PUSH(n, fcost)
```

If a heuristic is admissible and consistent, then we have the guarantee that the first time a vertex is expanded, it will be with the lowest possible cost. Hence, even though a vertex might be added to the openSet more than once, if will only be expanded once.

Another consequence of this guarentee is that the first time we arrive at d, we can terminate because we have the shortest path.

During the search we maintain a parent lookup table that maps a vertex n to its shortest path parent. At the conclusion of the search we use this table to reconstruct the path Sec. III-G.

B. Parallel A*: First Attempt

We begin by exploring the opportunities for parallelism in the A* algorithm, taking inspiration from the related work discussed in Sec. II. These various methods explored parallelize over vertices, edges or the search itself.

Early in our discussion, disregarding previous literature, we formulated what we consider the simplest version of parallelizing A*. We did not implement or compare with this approach both because we do not believe it will be efficient and because we wanted to compare strictly within variants of HDA*. However, because it presented a good starting exercise in reasoning about A*, we present this below.

The approach is to greedily divide up work among processors, allowing for a dynamic allocation of work and hence good work distribution. Each processor, when it becomes idle, expands the vertex with the lowest $f(n)$ cost from the global openSet.

A major disadvantage of this solution is that all the data structures require locks, so there is likely to be quite a bit of contention, notably slowing down progress.

We note, however, that this is less problematic when the edge weight and/or heuristic computation is expensive since more time would be spent processing each vertex, decreasing the chance that two processors are trying to access the openSet at the same time.

We believe the issue of contention to be a fairly major fault of this simple approach and, after reviewing related work, opted to dive into HDA*, presented below.

C. HDA*

HDA* assigns vertices to processors using a hash function, allowing for a static allocation of work [1].

Definition *Ownership.* If a vertex is assigned to a particular processor, we say that the processor **owns** that vertex and that the processor is the vertex's **owner**.

When a processor expands a given vertex, instead of adding all of that vertex's neighbors to its own openSet, we hash each neighbor to determine which processor owns that neighbor. That neighbor is then given to that processor, for it to be processed.

Hence, we use the hash function to determine how to distribute work amongst processors. Otherwise, each processor expands vertices and evaluates the edge weight and heuristic the same way as the sequential method.

One notable difference between HDA* and sequential A* is that HDA* does not provide us with the same guarantees that sequential A* does with a heuristic that is both admissible and consistent. As mentioned above, with such a heuristic, sequential A* guarantees that each vertex will only be expanded once and that the first time we expand the vertex we have found a shortest path to it. In HDA* however, we lose these guarantees.

We cannot know in which order processors will finish their work or whether they will work at a consistent rate. Therefore, a higher cost path to the destination may be processed before the lowest cost path arrives in the destination owner's openSet. Therefore, we may need to open a vertex many times before the algorithm completes. Proper termination detection proved to be rather tricky and we discuss it further in Sec. III-D.

D. Distributed Termination Detection

The question then becomes: When has the algorithm completed? With sequential A* and an admissible and consistent heuristic, we know that once the destination has been reached, the algorithm is complete.

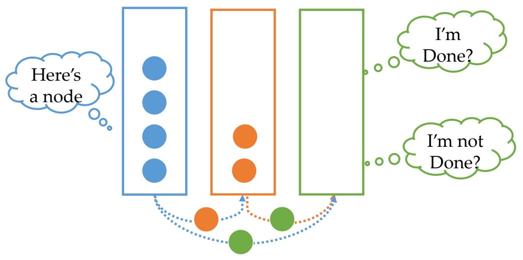


Fig. 2: This example illustrates some of the difficulty with distributed termination. Essentially, a processor could believe it is done when it temporarily has no work in its queue. However, this processor is not actually done as more work is being sent to it.

Even if the heuristic is not admissible or consistent, we could continue until the openSet is empty. With HDA*, however, it's not so simple.

Viewing a processor's openSet as it's work list, as a processor works through this list, it is constantly sending more work to other processors. Therefore, it may happen that a processor has an empty openSet and thinks itself done with all its work, but another processor is about to send it more work, as seen in Fig.2.

If we shut down a processor with an empty openSet, the work it receives in the future will never be processed and the algorithm may never terminate.

The first, simple step to address this is to notify all processors when the destination has been reached. That way, at the very least, no processor will stop before this point.

We will call the most simple approach we took which implemented a correct termination condition the *barrier method*. For this method, each processor hits a barrier when its openSet is empty. When all processors arrive at the barrier, each checks if its openSet is still empty and notifies all processors if it's not.

Each processor returns if no processor has more work in its openSet. The temporal guarantees of this method make it both easy to reason about and less time efficient. Each processor must wait for all others to temporarily finish every time its openSet is empty. Even if it receives more work while waiting, it cannot "short cut" out of the barrier.

In an attempt to resolve this, we also explored the *sum flags method*. For this method, each processor maintains a binary flag denoting whether or not the processor's openSet is empty. If the sum of the flags is equal to the number of processors, all processors are done.

We implemented both the sum flags and barrier methods for our shared address space implementation, coining them SAS-S and SAS-B respectively. For SAS-S, we utilized locks with respect to updating the flags.

We implemented the barrier method for message passing but were unsuccessful in using the sum flags

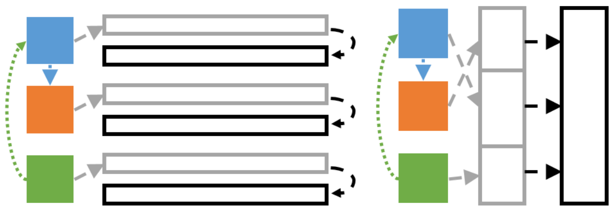


Fig. 3: The left shows the open and closed sets' connections to processors (shown in blue, red and green) for message passing. Each processor has its own private open and closed set and processors send vertices to each other. On the right is the analogous model for shared address space. Here there is one global closed set and a shared open set. Since the open set is in shared memory, processors can place vertices directly in the open set of another processor.

method. In message passing, a processor may have an openSet at one moment and send this information to other processors. It then may continue working again but is unable to "cancel" the sent data. Hence, the algorithm could terminate based off of stale data.

As we dove deeper into the issue of distributed termination detection, we realized that it alone is its own research question. Thus a full comparison of methods is beyond the scope of this project.

We did, however, explore, but not fully implement one other method: the four counter method [21]. This method tracks sends and receives, observing that if the sum of the total number of receives across all processors is equal to the sum of the total number of sends across all processors at a later time, all processors must be finished.

We began implementing this method. However, we realized that the necessary communication would be time consuming and would still result in processors waiting for others to arrive. Therefore, we did not pursue this method further.

E. Message Passing Model

In the message passing model, each processor has its own address space and can only communicate with the other processors via messages which are explicitly sent and received. Each processor maintains its own openSet, costToCome lookup table, and parent lookup table Fig.3.

While a processor still has work in its openSet, it expands these vertices, hashing each vertex's neighbors appropriately. When vertex n is hashed, the processor that owns n is sent $f(n), g(n)$ and n's parent.

If the destination's owner finds a path to the destination or a lower cost path than what it has found so far, it broadcasts the new cost to all other processors.

When a processor no longer has work and the cost to the destination is less than INT_MAX, hence we do have a path, we terminate according to the barrier method described in Sec. III-D.

Once we terminate, we reconstruct the path as described in Sec. III-G.

F. Shared Memory Model

In the shared address space model, all processors access the same address space but cannot communicate with each other explicitly. As a result, this model requires a number of locks.

There is a global array of openSets, each belonging to a different processor and each requiring its own lock. The costToCome and parent lookup tables are the same as in the sequential algorithm, but also require a lock for every vertex.

While a processor still has work in its openSet, it processes those vertices as in the sequential algorithm. It places each vertex in its owner processor's openSet as can be seen in Fig.3. Placing the vertex in the openSet requires using a lock and occurs after updating costToCome and parentVertex using another lock.

If the destination's owner finds a path to the destination or a lower cost path than what it has found so far, it updates the global cost variable.

When a processor no longer has work and the cost to the destination is less than INT_MAX, and hence a path has been found, we terminate according to the barrier or sum flags method described in Sec. III-D.

Once we terminate, we reconstruct the path as described in Sec. III-G.

G. Path Reconstruction

Upon A* termination, we only have the cost of the found path, not the path itself. Thus we must reconstruct the path as a post-processing step.

The path is reconstructed from the parent lookup table by tracing backwards along the path from the destination to the source. The destination vertex is added to the path, and then the parent of the current vertex is added until the source is found. At the end, the path is reversed so that it correctly goes from the source to the destination.

In the shared memory model, we can use the standard sequential reconstruction method described above.

In the message passing model, path reconstruction is still sequential, but the processor that found the destination vertex does not have all the information it needs. This is because it only knows the parent vertex for the vertices it owns, not necessarily all parents in the path.

We considered a few different methods of path reconstruction for this case. One solution would be to have the destination's owner, which we will call root, collect information of all vertices' parents up front and then have root reconstruct the path as usual. However, in a graph with 2000 vertices with a path of length 10, for example, this requires communicating *far* more information than is actually used, making it highly inefficient.

The next alternative we considered, depicted on the lefthand side of Fig.4, was to have root create an empty

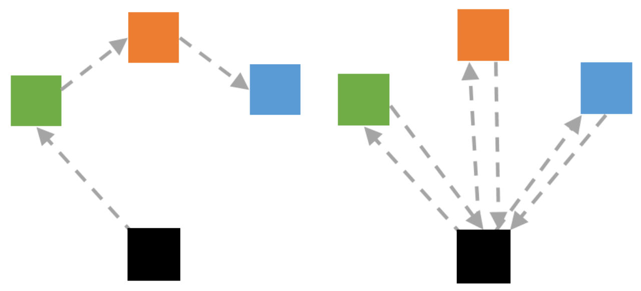


Fig. 4: A visualization of two proposed path reconstruction techniques for message passing. The left side sends along the path, incurring a higher cost than the righthand side, which individually requests for information.

data structure to hold the path, add the destination and its parent to the data structure, and send it to the owner of the destination's parent. Then each subsequent processor would add the parent of the final vertex in the structure and send the data structure to the owner of the vertex it added. The owner of the source would have the complete path in the end.

Looking at the communication cost of this method, we note that a data structure with the length of the path is sent once for every vertex in the path. Since the path is composed of vertices which are represented by ints, this is a total of len(path) * len(path) * sizeof(int) bits being sent between processors.

In the end, we found a more efficient method, depicted on the righthand side of Fig.4. As the root backtracks through path reconstruction, when it encounters a vertex it does not own, it sends a message to the vertex's owner requesting its parent and waits to receive the response. When it has found the source vertex, it reverses the path exactly as in the sequential case.

This requires two messages be sent for every vertex in the path not owned by root. Since each of these messages is a vertex, this is a total of 2 * len(path) * sizeof(int) bits being sent between processors in the worst case. This is asymptotically better than the previous method of passing an entire data structure between processors, so we chose this method.

IV. Experimental Setup

In order to evaluate the quality of our algorithms described in Sec. III, we carefully considered our problem size and specifications. Below we detail our graph creation, the heuristic used for A* search and how that informed another aspect of our creation and finally our experimental procedure.

A. Graph Construction

We construct our graph on a 2-dimensional grid. We randomly generate N unique vertices, representing each vertex as a point on the 2-d grid, hence with an x and y coordinate. Given these set of vertices, V, we next need to define our edge set.

One possibility would be to define edges randomly, similarly to our vertices. However, we can only complete a search if the source s and destination t are in the same connected component. Any vertices not in this connected component will not be searched because there is no edge connecting them. Therefore it is critical that our graph be connected.

For probabilistic guarantees on correctness, we turned to two random geometric graphs [**?**]: k-nearest and r-disc. In a k-nearest neighbor graph, each vertex is connect to its k nearest neighbors under some metric on the space. In a r-disc graph, an edge exists between two vertices, x, y iff $d(x, y) < r$, where d is our distance metric [22]. For our graph, d is the L_2 norm. We also use this distance to define the weight of each edge.

To have some guarantee of connectedness for our r-disc graph, our choice of r is dependent on the number of vertices in the graph and the size of our space [23], [24].

$$\gamma^* = 2\left[\left(1+\frac{1}{d}\right)\left(\frac{\lambda_d(c)}{\xi_d}\right)\right]^{\frac{1}{d}}$$

$$r(n) = \gamma^*\left(\frac{log(n)}{n}\right)^{\frac{1}{d}}$$

Such that d is the dimension of the space, $d = 2$ for our case, $\lambda_d(c)$ is the volume, ξ_d is the unit ball volume in d dimensions, and n is the number of vertices in the graph.

For our k-nearest neighbor graph, our choice of k is only dependent on the number of vertices, n, in the graph [23], [25] and is given by $k = 2e\log(n)$.

In practice, following our graph creation, we assert that the graph must be connected to continue.

Initially we were planning to compare k-nearest and r-disc graphs as different problem instances, since they vary with respect to average out-degree. However, preliminary comparison results did not show any interesting difference between the two graph types. Therefore, at each graph creation, we randomly choose the type.

B. Heuristic

The heart and power of A* search is the heuristic. We define our heuristic $h(n)$ as the L_2 norm distance from vertex n to the destination vertex, d. This heuristic is both consistent and admissible. Since our heuristic is the straight line distance between the vertex and the goal, it provides the minimum distance between them and thus is never an overestimate. Additionally, our metric is consistent since it satisfies the triangle inequality.

Our heuristic is quite strong and will guide the search towards a straight line path from the source to the destination. As we densify our graph and add more vertices, probabilistically, the search will be able to achieve a straighter and straighter path. Therefore, even as we increase the number of vertices we will only search the limited straight line area.

In order to force the search to explore more areas of the graph, we add obstacles to our search space. In each graph creation, we create a constant number of obstacles of constant size but placed in random locations. We maintain that no vertices may lie inside an obstacle, although, as a simplification, we permit edges to cross over obstacles. By creating an environment with obstacles, we create a much more realistic problem space.

C. Algorithm Comparison

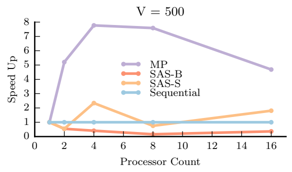
For each testing iteration, we specify the number of processors, the number of vertices, and the search algorithm. We then generate our graph, which is either a k-nearest graph or r-disc graph with equal probability, with a random set of obstacles. We randomly select a vertex to be our source and another to be the destination.

We test on processor counts $P = \{1, 2, 4, 8, 16\}$ and vertex counts $V = \{100, 500, 1000, 2000\}$ on our Sequential A*, our message passing implementation (MP) and our shared address variants (SAS-B, SAS-S). For our message passing algorithm we elected to use only 1 node and vary the number of processors per node. Since we already explored the effects of multiple nodes previously, we did not feel any additional insight would be gained.

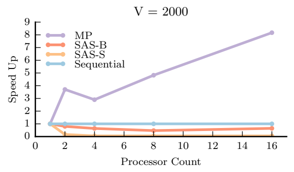
Since randomness is involved in our graph generation, we ran each iteration many times and averaged across the runs.

V. Results

We show the speedup of each of our algorithms for each size test graph Fig.5. We found that overall the message passing implementation won out over either shared address space implementation. This can quite clearly be seen in Fig.6a and Fig.6b. (For our graphs, graph shows standard error). In graphs with 100 or 1000 vertices, there appears to be a bit more variation, which will be discussed in more detail below. We believe that message passing achieves better speedup than a shared address space implementation due to contention and the need for locking on a shared memory model.



(a) Speedup for $V = 500$



(b) Speedup for $V = 2000$

Fig. 6: Speedup Plots across Problem Size

We can also see that speedup was not consistently achieved by adding more processors. However, the example of 2000 vertex graphs with our most effective algorithm, the message passing implementation, hints that perhaps parallelism is more effective for larger graphs.

We theorize that for smaller graphs, the overhead of communication outweighs the benefits of decreasing each processor's workload by a relatively small amount.

To go into more detail about these results, we will look at each size test graph individually.

For graphs with 100 vertices, MP took notably less time than our other algorithms Fig.7a.

Interestingly, SAS-B's time *increased* as processors were added. For such a small graph, the effects of parallelism are relatively minimal. Meanwhile, the barrier termination method may have significantly slowed progress with a large number of processors. On two processors for example, only one processor is ever waiting for another. With 16 processors, however, up to 15 of them could be waiting for that last one to complete processing, even if every processor has work that has been added to its openSet since arriving at the barrier.

For graphs with 500 vertices, MP is still notably faster than both SAS methods Fig.7b. SAS-B does quite well on 2 processors since neither processor waits very long at the barrier. However, SAS-S clearly has an advantage when run on over 2 processors. Thus the barrier is probably quite a detriment to speedup as expected.

As expected, MP is faster for graphs with 1000 vertices as well Fig.7c. One interesting contrast we see in this graph is that SAS-S and SAS-B are not so different; neither has a clear advantage. This could be because on a graph with 1000 vertices, there is so much work circulating at any given moment that a processor is unlikely to run out of work in its openSet while the algorithm is still running. Therefore, it is unlikely to spend a large portion of time stuck at a barrier. However, when running the algorithm on 16 processors, the disadvantage of the barrier method is so significant that SAS-S does do better. If a processor does reach a barrier before the completion of the algorithm, it may be waiting for quite a while since so much work is circulating among the other processors.

Graphs with 2000 vertices give evidence of the same phenomenon Fig.7d. SAS-S does do better than SAS-B. However, the difference is relatively small. While processors running SAS-B rarely get to the barrier, when a processor does get to the barrier it slows the algorithm quite a lot.

We will also look particularly at the results for SAS-B as they point to some interesting notes Fig.8. We see that, as expected, running A* on graphs with more vertices takes longer. We also see however that running A* on graphs with 2000 or 500 vertices on 8 processors takes longer than would be expected given the other data points. For the 500 vertex case, we can see in

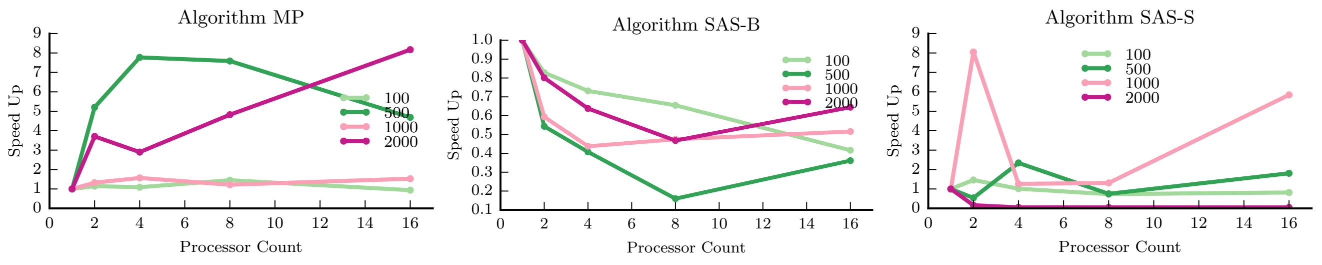


Fig. 5: Speedup Plots across Algorithms

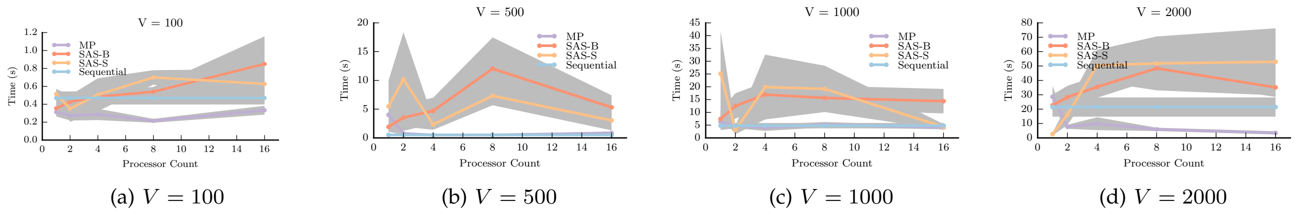


Fig. 7: Timing Plots across Problem Size (Number of Vertices)

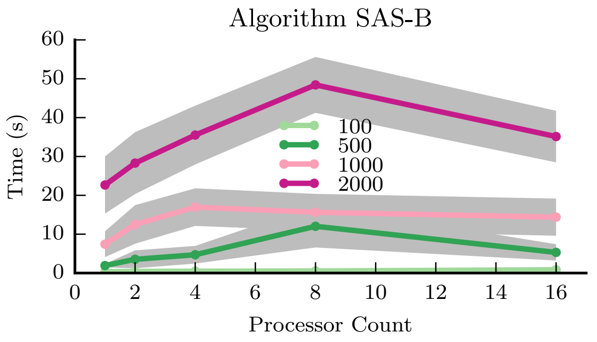


Fig. 8: Time Comparison for SAS-B across Problem Sizes

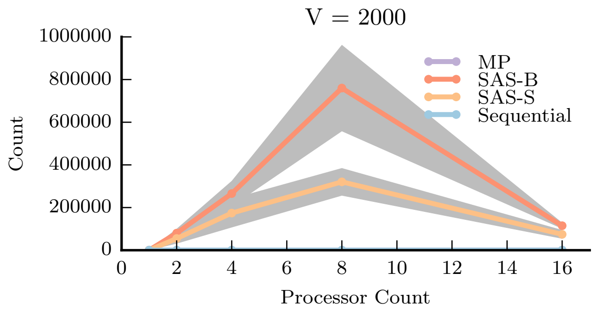


Fig. 10: The count of context switches across algorithms

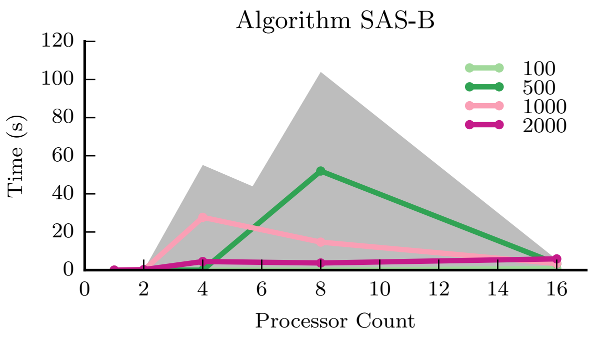


Fig. 9: The amount of time spent at the barrier for SAS-B

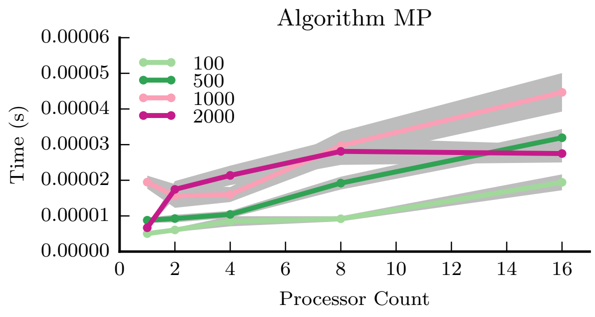


Fig. 11: The time to reconstruct the path for MP

Fig.9 that a huge portion of time is spent waiting at the barrier. For the 2000 vertex case, we can see in Fig.10 that this case causes an unusually large number of context switches as compared to the other cases. These may be the causes of the unusually large runtimes for SAS-B on graphs with 2000 or 500 vertices.

Unsurprisingly, the path reconstruction time increases with the number of processors. First note that since path reconstruction is sequential, the addition of more processors does not provide any benefits for this piece of the algorithm. The owner of the destination vertex must send and receive a message for each vertex in the shortest path which it does not own. With more processors, it is less likely that any given vertex in the path belongs to the owner of the destination vertex. Therefore, with more processors, path reconstruction requires more communication and thus takes more time.

VI. Discussion

In summary, we conclude that a message passing model is more efficient than a shared address space model for implementing hash distributed A*. We also hypothesize that the effects of parallelism are most evident on large graphs.

One significant limitation we faced in obtaining useful results was the high level of randomization in our test structure. We wanted a realistic set of test graphs, so we randomized many aspects of our graph creation. As a result, we had to run all of our algorithms on many iterations of each graph size and processor count in order to obtain reasonably accurate results. This took a fair amount of time and limited the quality of our final results, especially in the case of outliers which may have significantly affected the results we saw.

Another notable limitation we faced in efficiently producing results was the challenge presented by the type of graph on which we chose to test. We used random geometric graphs as our test set. While we believe this is realistic for many use cases, it will not apply to all use cases and also provides quite dense graphs. Since the graphs we used for testing were so dense, running each algorithm on these graphs was quite slow and included the expansion of many edges as compared to less dense graphs. Obtaining the number of data points we found we needed for accurate results became slightly more challenging given the time taken to run the algorithm on such dense graphs.

In response to these limitations, in the future we would of course like to obtain a larger body of results data. We would also like to test our algorithm on other types of graphs aside from random geometric graphs to see if our results hold across all graph types. Lastly, we would like to further explore the rich area of distributed determination detection and compare across different detection methods for both shared address space and message passing.

Work Distribution

Equal work was performed by both project partners.

ACKNOWLEDGMENT

We would like to thank the members of the 15-418 course staff for their guidance on our final project. We would also like to thank Siddhartha Srinivasa for his helpful discussion and Oren Salzman for his advice on random geometric graphs. Finally, we appreciate the Computer Science Department's Finals Week pastries, which greatly helped in the writing of this report.

References

[1] A. Kishimoto, A. S. Fukunaga, A. Botea, *et al.*, "Scalable, parallel best-first search for optimal sequential planning.," in *ICAPS*, 2009.

[2] L. E. Kavraki, P. Švestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *TRA*, vol. 12, no. 4, pp. 566–580, 1996.

[3] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Batch informed trees (bit*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs," in *ICRA*, pp. 3067–3074, IEEE, 2015.

[4] E. Rippel, A. Bar-Gill, and N. Shimkin, "Fast graph-search algorithms for general-aviation flight trajectory generation," *Journal of Guidance, Control, and Dynamics*, vol. 28, no. 4, pp. 801–811, 2005.

[5] R. Dalvi, A. Suszko, and V. S. Chauhan, "Identification and annotation of multiple periodic pulse trains using dominant frequency and graph search: Applications in atrial fibrillation rotor detection," in *International Conference of the Engineering in Medicine and Biology Society*, pp. 3572–3575, IEEE, 2016.

[6] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp. 100–107, 1968.

[7] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *Boost Graph Library: User Guide and Reference Manual, The*. Pearson Education, 2001.

[8] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, *et al.*, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pp. 97–104, Springer, 2004.

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

[10] A. Grama and V. Kumar, "Parallel search algorithms for discrete optimization problems," *ORSA Journal on Computing*, vol. 7, no. 4, pp. 365–385, 1995.

[11] S. Dutt and N. R. Mahapatra, "Parallel a* algorithms and their performance on hypercube multiprocessors," in *Parallel Processing Symposium, 1993., Proceedings of Seventh International*, pp. 797–803, IEEE, 1993.

[12] M. Evett, J. Hendler, A. Mahanti, and D. Nau, "Pra*: Massively parallel heuristic search," *Journal of Parallel and Distributed Computing*, vol. 25, no. 2, pp. 133–143, 1995.

[13] L. H. O. Rios and L. Chaimowicz, "A survey and classification of a* based best-first heuristic search algorithms," in *Brazilian Symposium on Artificial Intelligence*, pp. 253–262, Springer, 2010.

[14] E. Burns, S. Lemons, W. Ruml, and R. Zhou, "Best-first heuristic search for multicore machines," *Journal of Artificial Intelligence Research*, vol. 39, pp. 689–743, 2010.

[15] L. H. O. Rios and L. Chaimowicz, "Pnba*: A parallel bidirectional heuristic search algorithm," *Proceedings of the XXXI Congressa da Sociedade Brasileira de Computaçao*, 2011.

[16] I. Rafia, "A* algorithm for multicore graphics processors," 2010.

[17] Y. Zhou and J. Zeng, "Massively parallel a* search on a gpu.," in *AAAI*, pp. 1248–1255, 2015.

[18] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1408.2041*, 2014.

[19] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *First International Workshop on Graph Data Management Experiences and Systems*, p. 2, ACM, 2013.

[20] D. Gregor and A. Lumsdaine, "The parallel bgl: A generic library for distributed graph computations," *Parallel Object-Oriented Scientific Computing (POOSC)*, vol. 2, pp. 1–18, 2005.

[21] F. Mattern, "Algorithms for distributed termination detection," *Distributed computing*, vol. 2, no. 3, pp. 161–175, 1987.

[22] M. Penrose, *Random geometric graphs*. No. 5, Oxford University Press, 2003.

[23] E. N. Gilbert, "Random plane networks," *Journal of the Society for Industrial and Applied Mathematics*, vol. 9, no. 4, pp. 533–543, 1961.

[24] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *IJRR*, vol. 30, no. 7, pp. 846–894, 2011.

[25] L. Janson, B. Ichter, and M. Pavone, "Deterministic sampling-based motion planning: Optimality, complexity, and performance," *arXiv preprint arXiv:1505.00023*, 2015.

[26] K. Solovey, O. Salzman, and D. Halperin, "New perspective on sampling-based motion planning via random geometric graphs," in *RSS*, 2016.

TABLE I: Full Results

Algorithm	Processors	Vertices	Time (sec)	Edges Expanded	Nodes Expanded	Context Switches
Sequential	1	100	0.47	129.875	6.0	21.0625
Sequential	1	500	0.52	407.785714286	13.8571428571	33.0714285714
Sequential	1	1000	4.7909997	829.8	25.5	75.7
Sequential	1	2000	21.4555584444	2461.22222222	80.5555555556	366.555555556
MP	1	100	0.31248888	252.68	12.04	15.16
MP	1	500	4.00756333333	1458.83333333	45.625	36.9583333333
MP	1	1000	6.28185705263	1695.63157895	39.5789473684	110.105263158
MP	1	2000	28.6313063125	2969.4375	71.375	320.625
MP	2	100	0.2726844	243.066666667	10.6666666667	13.8
MP	2	500	0.769882733333	1029.26666667	28.6	19.8
MP	2	1000	4.747294	1846.9	49.2	49.0
MP	2	2000	7.728247	2090.55555556	49.5555555556	95.4444444444
MP	4	100	0.286510071429	433.285714286	18.7857142857	13.2857142857
MP	4	500	0.515689428571	1358.0	39.7142857143	19.5
MP	4	1000	4.00543788889	3820.66666667	101.777777778	46.8888888889
MP	4	2000	9.87912581818	4533.18181818	108.636363636	105.0
MP	8	100	0.215341933333	624.466666667	27.2666666667	14.4
MP	8	500	0.5282682	2259.53333333	63.6666666667	19.6666666667
MP	8	1000	5.153589	5654.66666667	143.222222222	53.2222222222
MP	8	2000	5.93648644444	5426.11111111	120.555555556	85.2222222222
MP	16	100	0.332904785714	1280.64285714	55.7857142857	14.4285714286
MP	16	500	0.855245733333	3440.66666667	97.9333333333	24.2
MP	16	1000	4.10946766667	6005.44444444	159.666666667	156.0
MP	16	2000	3.504584	70891.5	1763.0	653.0
SAS-B	1	100	0.353764066667	133.933333333	6.0	20.5
SAS-B	1	500	1.92069653333	1016.03333333	31.6666666667	47.0666666667
SAS-B	1	1000	7.42286415789	2761.36842105	79.4736842105	187.0
SAS-B	1	2000	22.6533982667	3746.86666667	90.2	543.2
SAS-B	2	100	0.4269815	314.7	14.3	24.2
SAS-B	2	500	3.53443614815	1595.07407407	47.2962962963	2037.33333333
SAS-B	2	1000	12.510616	3177.94444444	82.8888888889	14784.4444444
SAS-B	2	2000	28.2896972222	7187.5	167.388888889	80111.3888889
SAS-B	4	100	0.483658464286	328.142857143	14.6785714286	303.214285714
SAS-B	4	500	4.71011489286	4475.82142857	167.607142857	40784.4285714
SAS-B	4	1000	16.9652666111	3733.27777778	98.1111111111	98574.0
SAS-B	4	2000	35.5060253077	5143.0	126.076923077	265563.923077
SAS-B	8	100	0.539955071429	558.642857143	23.9285714286	132.785714286
SAS-B	8	500	12.0324415385	10172.1923077	321.576923077	77299.2307692
SAS-B	8	1000	15.6381565882	6023.17647059	159.294117647	198552.176471
SAS-B	8	2000	48.4169586875	9353.6875	211.125	760128.9375
SAS-B	16	100	0.849634466667	1576.73333333	64.8	331.8
SAS-B	16	500	5.31414043333	4142.46666667	118.166666667	9578.2
SAS-B	16	1000	14.3843989444	9048.0	225.166666667	32987.3333333
SAS-B	16	2000	35.1248896842	10143.0	232.473684211	114916.157895
SAS-S	1	100	0.5136088	114.2	4.6	23.4
SAS-S	1	500	5.497841	2368.0	57.6	85.0
SAS-S	1	1000	25.1283912	6625.4	186.8	429.0
SAS-S	1	2000	2.6509986	664.0	16.0	117.8
SAS-S	2	100	0.3514684	317.0	12.8	20.6
SAS-S	2	500	10.1708068	36806.4	1150.2	1499.2
SAS-S	2	1000	3.122217	927.0	25.75	3683.75
SAS-S	2	2000	16.19267925	4993.5	125.5	54929.75
SAS-S	4	100	0.5065622	260.6	10.8	255.8
SAS-S	4	500	2.3491372	552.4	15.0	7646.8
SAS-S	4	1000	19.893486	4038.25	96.5	131261.25
SAS-S	4	2000	50.32998625	2450.0	61.25	174145.75
SAS-S	8	100	0.6992188	673.0	27.4	131.4
SAS-S	8	500	7.320717	2125.4	61.2	14854.2
SAS-S	8	1000	19.1798678	7516.8	203.4	105560.6
SAS-S	8	2000	51.8222414	6657.0	143.4	320472.6
SAS-S	16	100	0.627226	704.6	29.2	137.4
SAS-S	16	500	3.0413238	2870.6	85.2	5213.0
SAS-S	16	1000	4.2998006	3177.8	88.0	13148.0
SAS-S	16	2000	52.93673975	6985.75	161.75	74384.75