Parallelized Optimal Solution Solver for the Vehicle Routing Problem

Summary

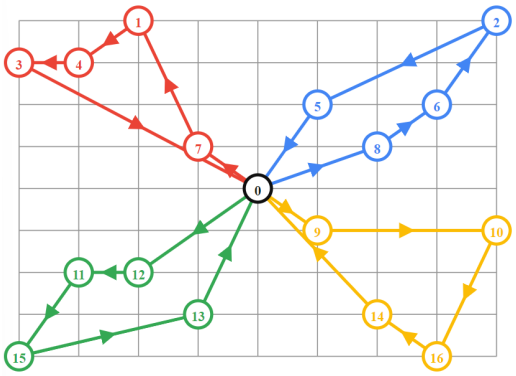
We implemented an Optimal Solution Solver for the Vehicle Routing Problem utilizing OpenMPI and various parallel optimization techniques to produce a 18.7x speedup on the Bridges-2 supercomputer.

Background

The **vehicle routing problem (VRP)** is a problem that finds the optimal set of routes for a fleet of vehicles to traverse in order to visit every single location within a graph and return to the starting location, or deliver to a given set of customers. This problem is an extension of the traveling salesman problem and has been used widely for applications involving large scale delivery optimizations and savings. Determining the optimal solution to the VRP is NP-hard and commercial solvers tend to use heuristics to generate an approximate solution due to the size of the real world VRPs they need to solve.

In the VRP, a certain number of *vehicles* must start from a central *depot* and traverse through a given *road network* to a set of *customers* and then each *vehicle* must return back to its starting location. Every customer in the road network must be visited. This information is stored in a VRP struct where the points are a vector of ints, and the number of vehicles is stored. This struct is the input to the parallelized optimal solution solver. After calculation, the program returns a set of *routes,* S, in VRPsolution struct that specifies the route that each vehicle must take throughout the road network. The goal of the VRP is to return a set of routes

with the global transportation cost minimized. This cost may be monetary, distance, or time. In our implementation these parameters are all represented in the VRP and VRPsolution structs, and we chose to optimize for least cost time solutions. The diagram below shows an example solution to a vehicle routing problem involving 4 vehicles, with each vehicle's route being represented by a different color.



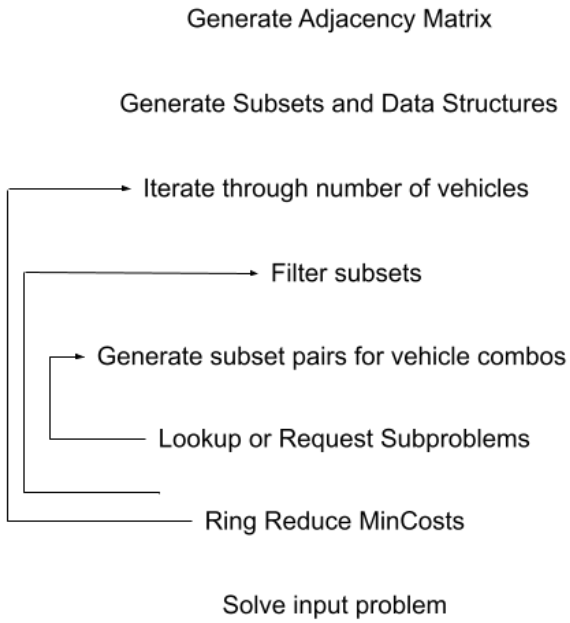
In the optimal solution solver, the road network is described using a graph of nodes and each node refers to a single customer, the cost of traversing from one node to the next is assigned to the edges in the graph. All of these edges are represented within an adjacency matrix.

To find the solution to the VRP, the complete problem is split into smaller VRPs which at the base level will have 1 vehicle and will be a traveling salesman problem (TSP). These TSPs are solved using a branch and bound search algorithm implemented through matrix reduction operations on the adjacency matrix and a priority queue. Each decision in the path through the points results in a new "Node" which is associated with a cost to arrive at the node, and a cost to complete travel from the node. The priority queue is used to continue from the least cost node and explores until it reaches the final destination. This allows us to guarantee that the

final solution is the optimal one, while trimming away areas of exploration that we know will not be optimal. Our VRP solving algorithm uses a bottom up dynamic programming approach which generates all possible subsets of the original points and valid number of vehicles and produces solutions for smaller subproblems until an optimal solution for the full problem is found. There is data dependency in the VRP algorithm because each layer of the bottom up programming depends on all layers before it. Larger subsets of nodes and vehicles need all smaller subsets in order to calculate their minimum cost route assignment. In our implementation each number of vehicles is considered a different level and we take advantage of the fact that VRP subproblems on the same level are independent of each other's data, allowing random distribution of the work on each level. With synchronization we are able to ensure that each processor will be working on the same level at the same time which allows us to ensure that all data dependencies are non-blocking. This data dependency is managed through sharing of data between processors, which results in large amounts of communication being required. The VRP scales with n nodes and v vehicles at a rate of O((n^2 + v)*2^n), so at larger node numbers the problem gets very computationally expensive. These data dependencies require us to communicate solutions instead of recomputing them, as we discovered that it is almost never more efficient to avoid communication and compute a result that is not in a current processor's lookup table. Therefore it is possible for this problem to benefit greatly from parallelism when processors split the work required at each level of vehicles. This also creates a large amount of data reuse, which is capitalized upon through the use of lookup tables which have O(1) access times.

Approach

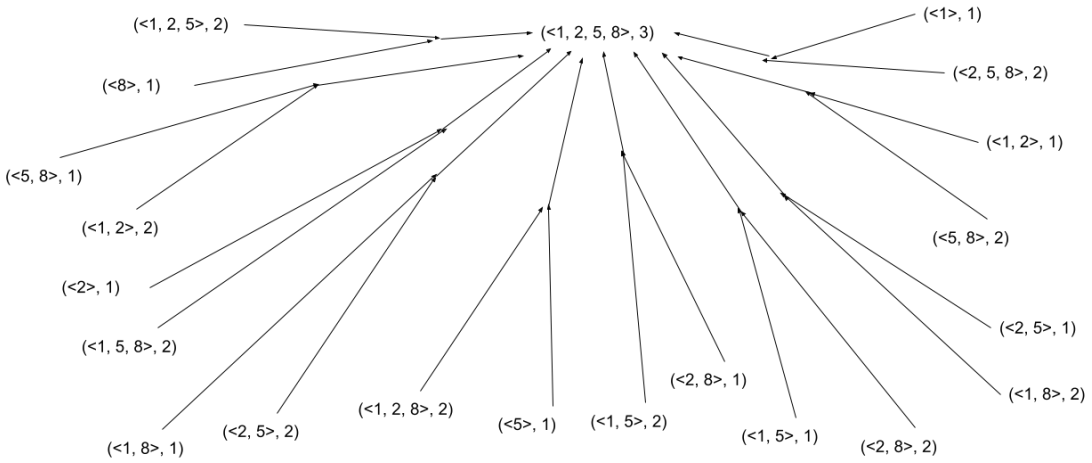
Our approach and implementation utilized OpenMPI communication and was designed to be run on the PSC Bridges-2 machine due to the computation size of the large VRP problems.



The first step of the approach is to generate all sub problems for a given VRP so we can distribute the subproblems among the processors and parallelize our computation. Each subproblem should be specified by a subset of the total nodes and a number of vehicles traversing that subset. For future reference each sub problem will be denoted by the following format: (<n1,n2,n3…>,v) where each n is a distinct node and v denotes the number of vehicles in the subproblem. We first generated every possible subset of nodes and then paired those nodes with every possible valid number of vehicles. Each pairing is stored inside a VRP struct to keep track of the subproblems.

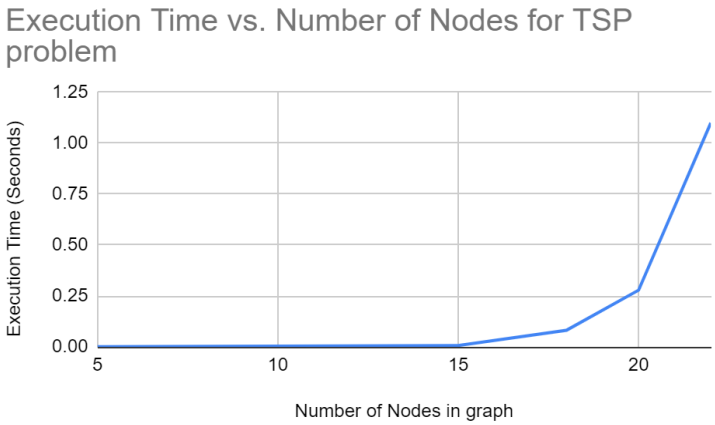
In our initial approach, the “master” processor would generate all of the possible subproblem combinations for a VRP and then assign that generated work to a large vector which kept track of which subsets each processor would work on. In our process of implementing this version we realized that communicating the subproblems from the master processor to all other processor was a very inefficient process since it required us to reformat our subproblems into an array and communicate multiple times with each processor to ensure that the subproblem being sent was received properly. Regardless of whether we utilized a broadcast or ring communication, the cost would still be the same. Instead, we found that the more time efficient solution in this case was for each processor to generate all possible work from the beginning and filter the subproblems based on a hashing function, reducing our need for initial communication.

After all valid subproblems are generated by the different processors, the processors begin calculation. Due to the nature of the VRP, each subproblem relies on the data from almost all smaller subproblems below it, an example of this is shown in the figure below where the arrows represent the dependency between the subproblems.



Our initial implementation of the VRP solver looked at the problem in a top-down dynamic programming approach. We had originally thought to do a recursive implementation where processors would start with large problems and would recursively call itself on smaller problems in order to compute the optimal solution. However, we quickly found that this implementation was very slow. Recursively approaching the problem resulted in a large call stack and resulted in inefficiencies when it came to data lookup. It also did not pair well with our idea to implement a granularity limit on communication, where small subproblems would be recomputed. From our testing we realized that it is almost always less time consuming to request the solution of a subproblem. This resulted in large stalls as processors had to wait on each other for information, and overall became quite messy and slow because of data synchronization issues. Our following implementation saw us approaching the problem from the bottom up, calculating the smallest problems first so that we could share that solution with processors working on larger problems which depended on those smaller problems. In order to optimize this dependency, we chose to implement a hash table in each processor that would hash the different subproblems/solutions and keep track of which processor was in charge of calculating those subproblems. We chose to use a hash table due to the O(1) average lookup time. When a processor is calculating a subproblem and needs the information of the smaller subproblems it would first look at its own hashtable to see if it already has the solution to the subproblem and if it doesn't, request the corresponding processor for the solution to the subproblem. If the other processor hasn't solved the subproblem the processor would continue to calculate the smaller dependencies itself, otherwise it would receive the precalculated solution. This hash table implementation is especially important as the number of nodes in the

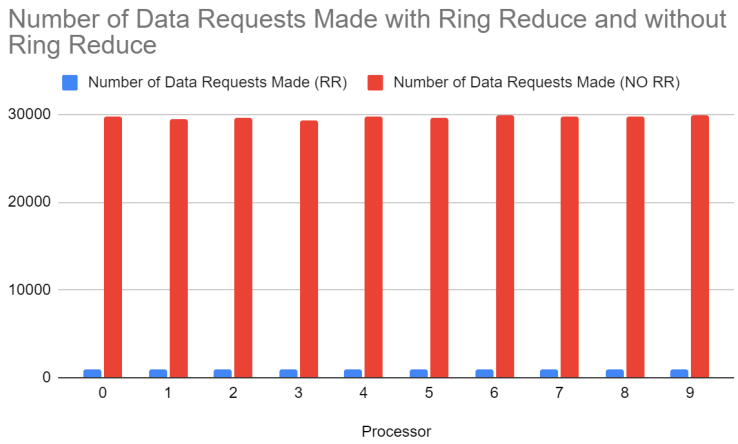
VRP grows. As shown in the figure below the execution time for a single TSP grows exponentially depending on the amount of nodes in the graph.



Implementing the hashtable allows us to reduce the amount of computation greatly by sharing precalculated solutions among processors. While extra communication is needed to share this data, this improved performance because computation was reduced. After communication analysis, we noticed that this implementation resulted in a large amount of requests being made by each processor for data that it did not have. While these requests mean that we are successful in reducing our computation per processor, there were a lot of optimizations that could be performed to reduce unnecessary communication.

Our next step was to implement a way to periodically synchronize the hash tables across all processors, this way we would reduce the amount of data requests and reduce communication costs. With a more recently updated hashtable, each processor would have the solutions to more subproblems and would be more likely to find a needed dependency within its own hashtable, greatly reducing the need for data requests to other processors. To implement this functionality, we considered using broadcast and all scatter methods but we deemed both to be far too inefficient. Instead, our implementation utilized a Ring Reduce(RR in

some graphs) to synchronize and merge the hash tables of different processors. The Ring Reduce eliminated all data requests for standalone cost information, and the figure below displays how effective the method was at reducing communication. The program was run with 64 Processors, 16 nodes, 3 vehicles, and seed 25.



Processor	Number of Data Requests Made (RR)	Number of Data Requests Made (NO RR)
0	969	29720
1	962	29521
2	964	29545
3	962	29280
4	951	29780
5	972	29605
6	951	29822
7	946	29741
8	960	29685
9	962	29836

Furthermore, we chose to optimize the data communicated due to our increasing concerns with memory capacity limitations. In our ring reduce, we did not communicate the solution routes to the subproblems, but instead only the minimum costs. So, we got rid of the common case of only requesting cost information, since the majority of subset's optimal routes are not relevant

to the minimum cost comparison that takes place. Now all lookups to subsets which routes are irrelevant to the final solution take place in O(1), but this does increase the overhead of lookups and data storage.

Results

For our project, we defined performance by the speedup of our parallel program over the single core implementation of the VRP solver.

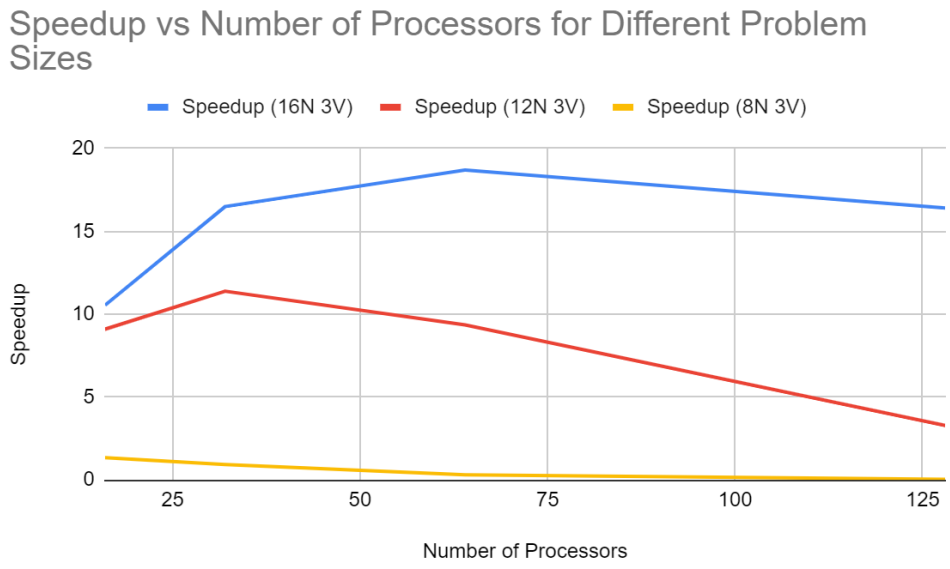
For our experimental setup, we originally wanted to incorporate official Uber data to create our graph and simulate the VRP using Uber costs and travel distances, however, the process of reformatting and manipulating the Uber data to be well suited for our problem was taking too much time and ultimately not worth it. The main issue arose with the fact that the Uber data did not form a fully connected graph and that the cost/distance data involved in the dataset were from a span of 10 years with a lot of variation from the same trip. We instead pivoted to using a seeded randomly generated adjacency matrix for all of our testing purposes. We tested our program with a variety of input sizes reaching up to 16 nodes and 5 vehicles. Because TSPs and VRPs scale exponentially with the number of inputs, we found that 16 nodes and 5 vehicles was about the largest we can get to without running into memory constraints which we will talk more about below.

In our best case, our speedup reached up to 18.7x speedup over a serial implementation. The table below shows the speedup results from running the program on a VRP with 16 nodes and 3 vehicles on different amounts of processors. Our highest speedup was achieved with 64 processors which we believe is near the sweet spot where the net-benefit of

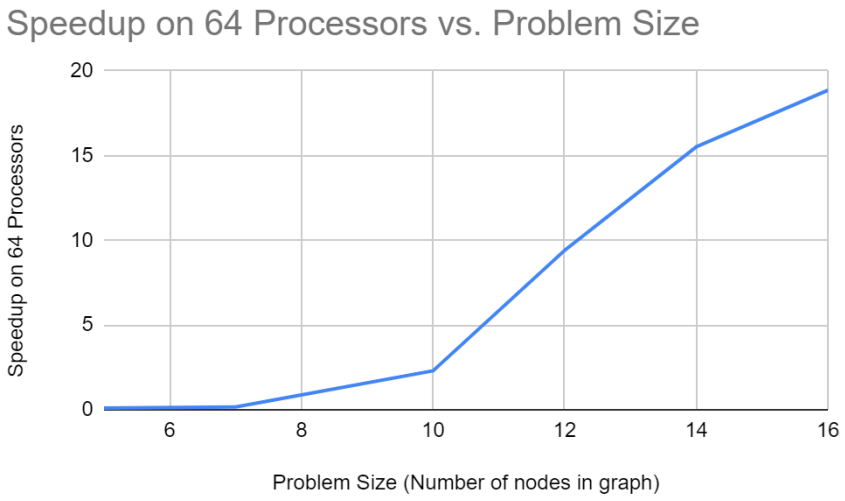
the communication is the most positive. We realized our original goal of near 0.5 times linear speedup was not feasible given the large communication cost, computational complexity, and size of the working set of the problem, and we are satisfied with our results as we have gone through many iterations of optimizations.

Number of Processors	Speedup (16N 3V)	Speedup (12N 3V)	Speedup (8N 3V)
16	10.53865979	9.089579525	1.342519685
32	16.48790323	11.38028272	0.9266304348
64	18.69684499	9.345864662	0.3072072072
128	16.39797883	3.283569078	0.03344776851

On VRPs that were larger than 16 nodes and 3 vehicles we continued to see increased performance. Our speedup with 64 processors on a problem size of 17 nodes and 3 vehicles was over 22x but unfortunately we couldn't test these larger problem sizes more due to memory constraints that we'll mention below. The figure below displays the relationship between the scaling of speedups and the size of the problem. It is evident that our program achieves greater speedups with large problem sizes and this makes sense because with larger problems our communication reduction optimizations are more powerful. It is also interesting to note that our speedups start decreasing at different numbers of processors depending on the size of the problem, this is due to the benefit of communication and reducing computation changing based on problem size as well as the communication costs of ring reduce not being worthwhile for smaller problem sizes.



For further demonstration of how the speedup almost exponentially scales with the problem size, the figure below displays the relationship between the speedup and problem size for a fixed 64 processors.



Additionally, our implementation of ring reduce was very successful and achieved around 3x speedup at larger problem sizes across all numbers of processors. This is due to the

fact that we eliminated nearly all of the data requests from individual processors, leaving only requests for routes in the optimal solution of a subproblem.

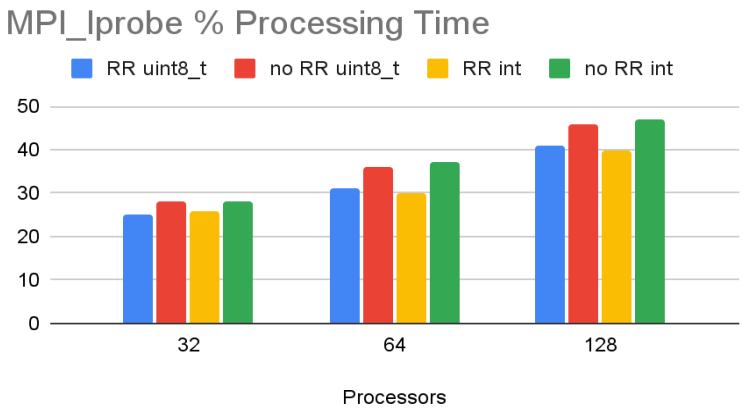
Number of Processors	Execution Speed (RR)	Execution Speed (No RR)	Ring Reduce Speedup
16	3.88	11.759	3.030670103
32	2.48	7.2931	2.940766129
64	2.187	6.021	2.75308642
128	2.4936	7.916	3.174526789

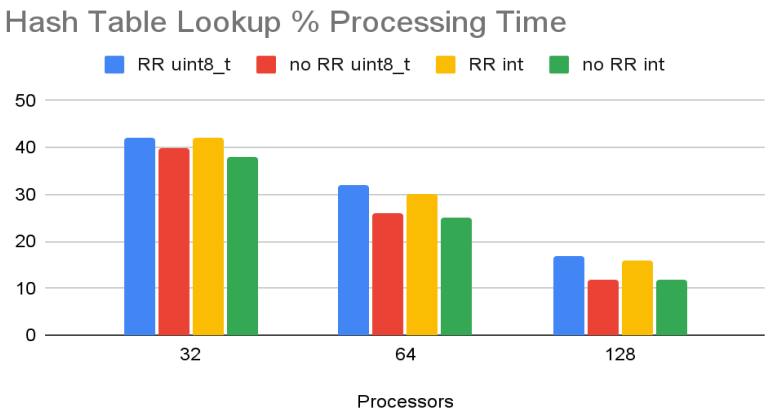
Unfortunately, we were not able to increase our speedup further even though we tried multiple methods of optimization. After a lot of analysis, we found that we had good workload balance between processors, most likely due to our random hashing of the subproblems. Our speedup was mainly limited by two operations in our program, hashtable lookup and message probing. The limitation by the hash table lookup is unavoidable. The hash table reduces the frequency of communication by 30x, as seen in an earlier chart, and our ring-reduce speedup figures demonstrate its effectiveness. To get past this limiting factor we would've needed to use a different data structure to store our subproblems and results, possibly one that has fewer pointer chasing steps or greater locality. Our message probing limitation could not be significantly optimized further in our current framework unless we chose to switch to an interrupt driven method of communication with OpenMPI. We did not find good resources and documentation on an interrupt driven method of communication with OpenMPI with the remaining time in our project, so we continued with our polling method. The image below shows the exact statistics for our program's limiting factors.

```
100.00%        VRP-release
-   41.81%        VRP-release
       36.82%        [.] std::_Hashtable<VRP, std::pair<VRP const, VRPsolution>, std::allocator<std::pair<VRP const, VRPsolution> >, std::__detail::_Sele
        2.96%        [.] fillRequests
        0.47%        [.] reduce
        0.34%        [.] syncLevel
        0.23%        [.] MPI_Iprobe@plt
        0.22%        [.] genSubs
        0.14%        [.] std::_Hashtable<VRP, std::pair<VRP const, VRPsolution>, std::allocator<std::pair<VRP const, VRPsolution> >, std::__detail::_Sele
        0.10%        [.] std::vector<unsigned char, std::allocator<unsigned char> >::_M_realloc_insert<unsigned char const&>
        0.10%        [.] std::vector<std::vector<unsigned char, std::allocator<unsigned char> >, std::allocator<std::vector<unsigned char, std::allocator
        0.06%        [.] vrpSolve
        0.05%        [.] ringReduce
-   29.96%        libmpi.so.40.20.5
       21.51%        [.] mca_pml_ucx_iprobe
        8.28%        [.] PMPI_Iprobe
        0.13%        [.] opal_progress@plt
-    9.99%        libucp.so.0.0.0
        6.95%        [.] ucp_tag_probe_nb
        3.03%        [.] ucp_worker_progress
-    6.56%        libuct_ib.so.0.0.0
        6.55%        [.] uct_dc_mlx5_iface_progress
-    5.25%        libc-2.28.so
        2.59%        [.] printf_positional
        0.84%        [.] malloc
        0.81%        [.] _int_malloc
```

Statistics of 16 nodes and 4 vehicles running 64 processors with ring reduce and short messages.

In the below bar graph you can also see how the percent processing time of the processes changed as the number of processors increased, and with respect to different data storage and communication techniques. As expected, the percentage of processing time dedicated to polling for incoming messages increased with an increase in processors, while the percent time spent looking through hash tables decreased as communication became more expensive.



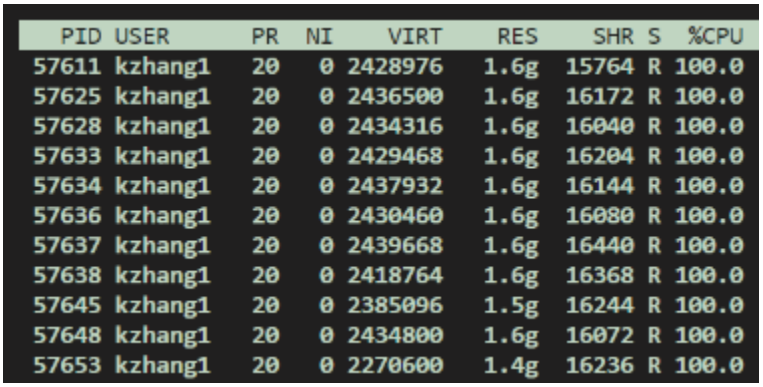


Another thing seen in these charts is the comparison between programs running ring reduce or not. As we expected, the time spent in hash table lookups is higher for the processes using ring reduce as their hash tables tend to be larger. But something we did not expect is that the decreased data storage size to uint8_t from int did not decrease lookup time. We now realize that this is likely because the lookups are a pointer chasing process and do not result in large amounts of locality. We had hoped that decreasing data size would increase data per cache line and increase cache hits resulting in decreased lookup times, but that was not the case according to these graphs. It likely did not have an effect due to pointer chasing having low locality.

Additionally, something that these charts made us realize is that when we switched to using a ring reduce algorithm, we didn't modify the frequency of polling for data requests. An optimization would be to reduce this frequency to decrease processing time dedicated to polling, or even switching to an interrupt based method as we discussed above.

For the size of the problems we could run, we were mainly limited by the fact that VRP problems are very large and take a lot of memory. More specifically, running our program on larger problem sizes resulted in crashes. These crashes were due to the fact that at larger

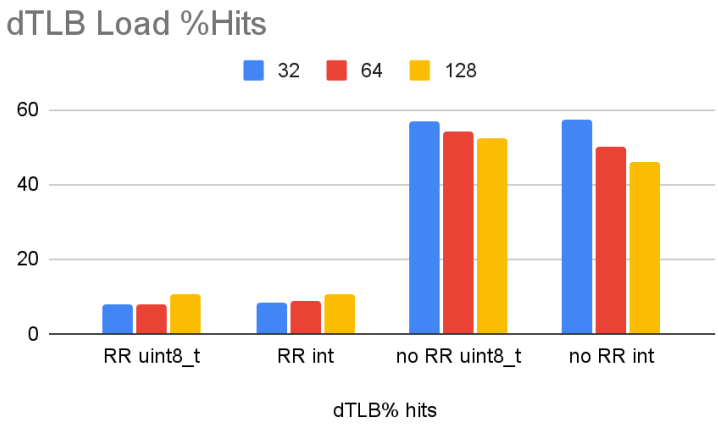
problem sizes, each of our processors were using 1-3 GB of physical memory. Each RM node on the Bridges-2 machine only contains 256 GB of RAM, so at a certain point, depending on how many processors we were using, the program would run out of memory to use and crash. Our memory analysis was mainly done through using the top command while our program was running in the background. In the screenshot below, each processor is using 1.6 GB of physical memory (RES). One of our main goals was to get problems of 16 nodes and 6 vehicles running without crashes, but despite reducing the memory footprint of the hash table data by a little under 2x by switching from ints to uint8_t's and not storing every route in every hash table, the problem size increased too drastically for these changes to be effective.



PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU
57611	kzhang1	20	0	2428976	1.6g	15764	R	100.0
57625	kzhang1	20	0	2436500	1.6g	16172	R	100.0
57628	kzhang1	20	0	2434316	1.6g	16040	R	100.0
57633	kzhang1	20	0	2429468	1.6g	16204	R	100.0
57634	kzhang1	20	0	2437932	1.6g	16144	R	100.0
57636	kzhang1	20	0	2430460	1.6g	16080	R	100.0
57637	kzhang1	20	0	2439668	1.6g	16440	R	100.0
57638	kzhang1	20	0	2418764	1.6g	16368	R	100.0
57645	kzhang1	20	0	2385096	1.5g	16244	R	100.0
57648	kzhang1	20	0	2434800	1.6g	16072	R	100.0
57653	kzhang1	20	0	2270600	1.4g	16236	R	100.0

There were various optimizations we did to reduce memory usage, but the problem size scales exponentially so ultimately we were not able to achieve significantly better memory usage. We implemented a method to switch between "long" and "short" messages where the short messages would not send the routes and would only send the minimum cost of the VRP and the long messages would send both. The hope with this feature was to reduce the amount of memory used in communication and storing the messages but it did not prove to make a real difference in our communication as most time was spent enabling communication, and it didn't make a significant enough impact on memory usage to be beneficial since there wasn't as much

data reuse to benefit execution times. We also implemented a partial ring reduce in addition to a full ring reduce. In this partial ring reduction we would only be ring reducing with a specified number of processors in the ring. This feature was aimed at reducing the size of the hash tables that each processor had to store, but also increasing the likelihood of data residing in a processor's hash table. We also chose to save our messages as uint8_t instead of ints to save space since our values would always be nonnegative below 255. While these additions did provide a reduction in memory usage, due to the exponentially increasing size of the problem they weren't sufficient to allow us to reach our 16 node 6 vehicle obstacle.



On a side note, as seen in the chart above, adding ring reduce decreased the data TLB load hits, but this is because processors weren't fielding requests from other processors for minimum cost data anymore. This is not indicative of decreased cache hits. Unfortunately cache miss information was unavailable using perf on the PSC nodes.

We had to change our goals throughout the duration of this project as we learned more about the problem. We successfully implemented a parallelized version of the Vehicle Routing Problem in C++ using OpenMPI. We chose to parallelize a brute force approach to the VRP instead of a genetic algorithm because it was better suited for us to demonstrate our ability to

parallelize and optimize. We did not go through with using Uber data because it was too incomplete and would've distracted us from focusing on the important implementation of our various parallelism techniques and optimizations. With our initial workload calculations and synchronizing the processors at different levels of problem size, our workload balance was good and was not the limiting factor for speedup. With good workload balancing, we opted to not implement a work-stealing algorithm since that would introduce a lot more communication with little pay-off. The speedup of our program scales with the size of the problem so we were not able to test the upper bound of the speedup but we do believe that it will grow to be large based on the measurable trends. We believe that our choice of machine was sound. Due to the memory requirements of the VRP we were only able to run larger problems on the PSC Bridges-2 machine. Overall, we implemented many new optimizations as we continued to implement and test our code. Each iteration of our implementation performed better than the one before and we spent a lot of time profiling our implementations to pinpoint areas of weakness that we could further improve with the intuitions and techniques we learned in class.

References

- Branch and Bound TSP - https://iq.opengenus.org/travelling-salesman-branch-bound/
- **https://www.askpython.com/python/examples/branch-and-bound-search**
- **Laporte, Gilbert. "The vehicle routing problem: An overview of exact and approximate algorithms." *European journal of operational research* 59.3 (1992): 345-358.**
- **Laporte, Gilbert, and Yves Nobert. "Exact algorithms for the vehicle routing problem." *North-Holland Mathematics Studies*. Vol. 132. North-Holland, 1987. 147-184.**

List of work

Total Credit: (bwilhelm - 55%) (kwzhang - 45%)