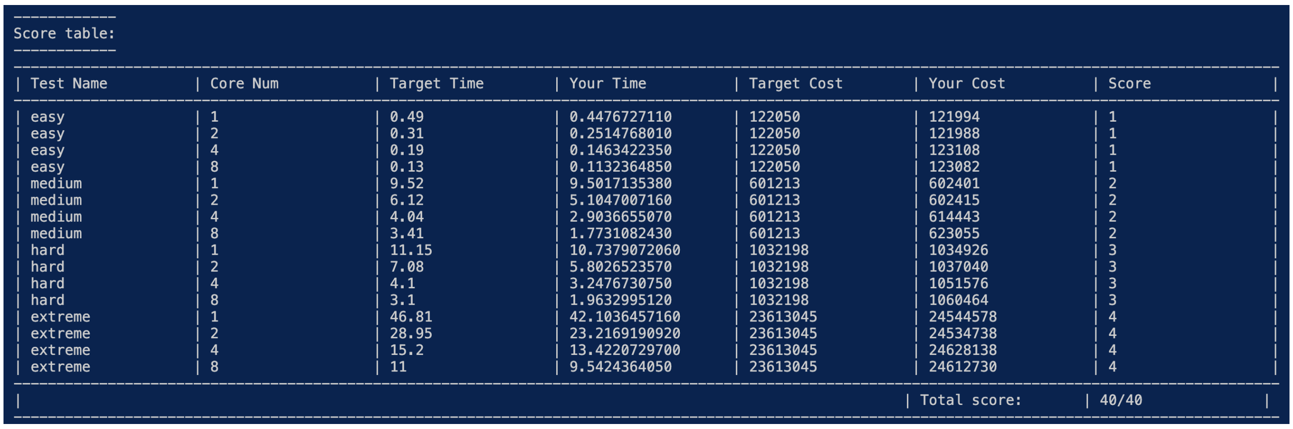
15-618 Parallel Computer Architecture & Programming

Project 4: Report **Vibha Arramreddy (varramre), Vedant Bhasin (vedantb)**

Results



```
------------
Score table:
------------
-----------------------------------------------------------------------------------------------------------------------------------------------
| Test Name          | Core Num           | Target Time        | Your Time          | Target Cost        | Your Cost          | Score              |
-----------------------------------------------------------------------------------------------------------------------------------------------
| easy               | 1                  | 0.49               | 0.4476727110       | 122050             | 121994             | 1                  |
| easy               | 2                  | 0.31               | 0.2514768010       | 122050             | 121988             | 1                  |
| easy               | 4                  | 0.19               | 0.1463422350       | 122050             | 123108             | 1                  |
| easy               | 8                  | 0.13               | 0.1132364850       | 122050             | 123082             | 1                  |
| medium             | 1                  | 9.52               | 9.5017135380       | 601213             | 602401             | 2                  |
| medium             | 2                  | 6.12               | 5.1047007160       | 601213             | 602415             | 2                  |
| medium             | 4                  | 4.04               | 2.9036655070       | 601213             | 614443             | 2                  |
| medium             | 8                  | 3.41               | 1.7731082430       | 601213             | 623055             | 2                  |
| hard               | 1                  | 11.15              | 10.7379072060      | 1032198            | 1034926            | 3                  |
| hard               | 2                  | 7.08               | 5.8026523570       | 1032198            | 1037040            | 3                  |
| hard               | 4                  | 4.1                | 3.2476730750       | 1032198            | 1051576            | 3                  |
| hard               | 8                  | 3.1                | 1.9632995120       | 1032198            | 1060464            | 3                  |
| extreme            | 1                  | 46.81              | 42.1036457160      | 23613045           | 24544578           | 4                  |
| extreme            | 2                  | 28.95              | 23.2169190920      | 23613045           | 24534738           | 4                  |
| extreme            | 4                  | 15.2               | 13.4220729700      | 23613045           | 24628138           | 4                  |
| extreme            | 8                  | 11                 | 9.5424364050       | 23613045           | 24612730           | 4                  |
-----------------------------------------------------------------------------------------------------------------------------------------------
|                                                                                                          | Total score:       | 40/40             |
-----------------------------------------------------------------------------------------------------------------------------------------------
```

Figure 1: checker.pl output

Design and Performance Debugging Journey

Design Journey

Broadcasts

Our initial implementation was designed somewhat like the parameter server communication model, except for the fact that the root node performs computations as well. The main issue with this approach was how costly broadcasts were and the fact that 1 point to point commuication per processor and a broadcast were required to communicate wire changes.

Ring Reduce

The ring reduce method didn't work well since we required multiple rounds of point to point communications to reach consensus among processors for wire count.

All Gather

Our final approach uses all gather as we found this to work best in terms of efficiency and correctness.

Transfer Unit

To communicate changed routes across processors we intialize a struct called `UpdatePacket` which contains three ints: `idx, bend1_x, bend1_y` the idx allows us to index into the global wires array for which each processot has the same ordering, since we index into the array to get the correct old wire, we don't need to store the start or end coordinates thereby saving memory. Furhtermore incorporating the global index into the update packet allows for flexibility in in workload distribution and targeted updates (i.e. only communicating wires that have changed).

Synchronization

We employ the asynchronous allgather operation, i.e. `Iallgather`, the main reason for this is that it allows us to overlap communication and computation i.e. initiate the allgather (for the previous batch) perform the routing computations for the current batch and then wait only before applying the received changes from other processors. Therefore synchronization for each processor only occurs before applying updates received from other processors.

Imperfect Speedup

Despite using asynchronous communication it still has an unavoidable overhead, and it scales poorly with the number of processors incurring $O(N^2)$ cost complexity in terms of processors.

High Thread Count Performance

At high threadcounts communication becomes more expensive due to allgather scaling poorly with the number of processors. Additionally, as the computation time decreases drastically, the bottleneck shifts to the initialization time which increases with the number of processors due to large data structures that need to be broadcast to all processors.

GHC Experiments

Speedup Graphs:

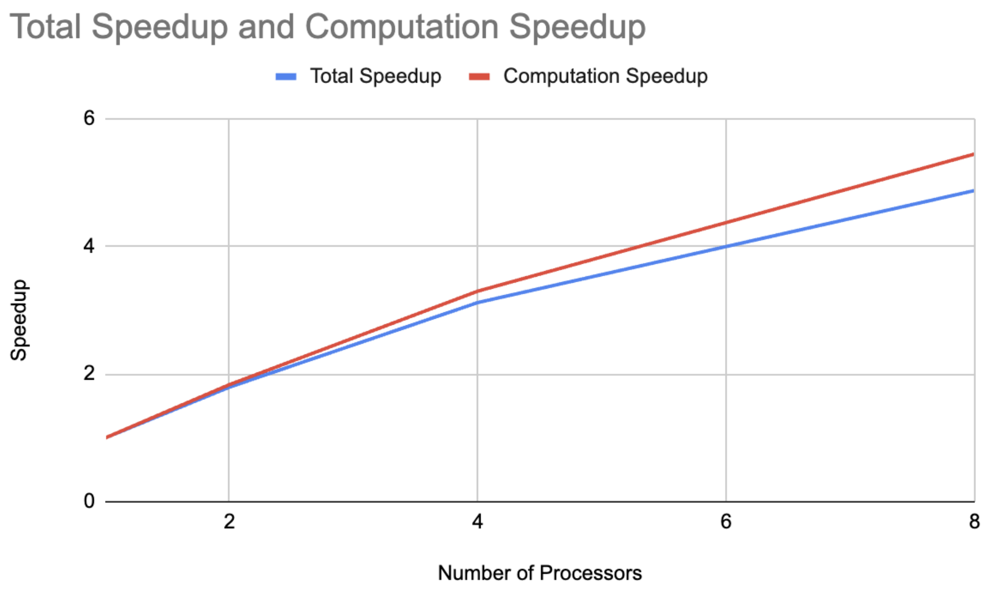


Figure 2: GHC: Total & Computation Speedup vs. Number of Processors on hard_4096

The speedup increases as we increase the number of processors. This is due to the fact we able to split the workload greater as we have more processors. However, we are not able to achieve perfect speedup due to the overhead of the MPI communication. We also note that as we increase the number of processors, the speedup increases by less. For example the speedup increase from 2 to 4 is greater than the speedup increase from 4 to 8 processors. This is due to the fact that communication overhead increases as we add more processors. Therefore, although we improve performance by spliting the workload more, we also face the drawback of increasing communication cost. This increase in communication cost also explains why the speedup gets further from ideal as we increase the number of processors. Additionally, we noticed that the total speedup increases less than the computation speedup as the number of processors increases. This means that our initialization time increases as the number of processors increases. This makes sense as in our initialization process we are broadcasting all the start-up information from the root node to the rest of the processors. As the number of processors increases, this broadcast takes longer, increasing initialization time and decreasing total speedup.

Cache Misses:

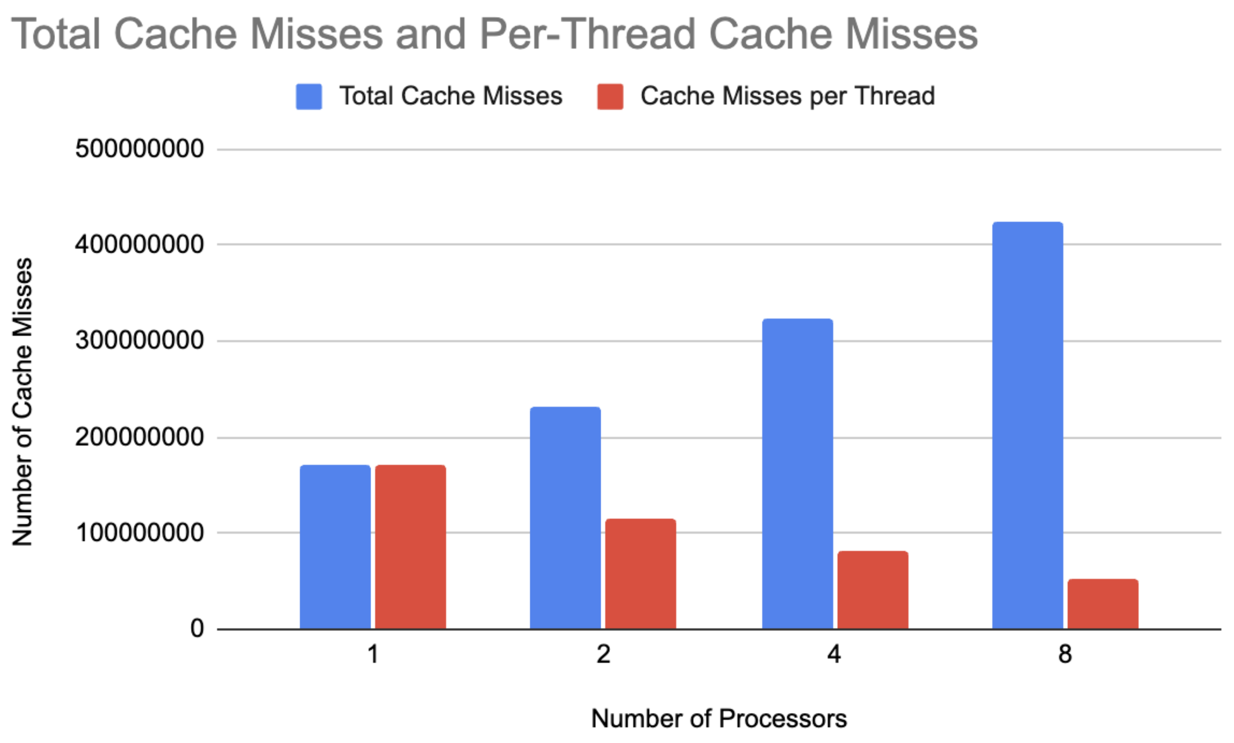
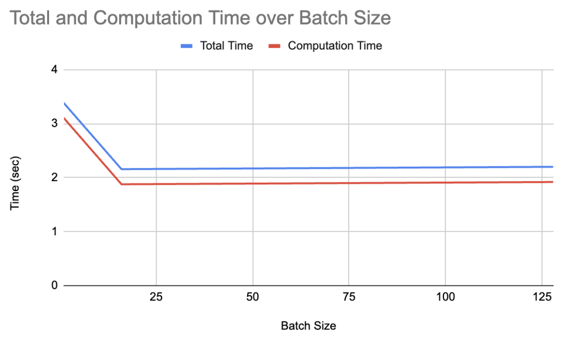


Figure 3: Total & Per Thread Cache Misses vs. Number of Processors on hard_4096

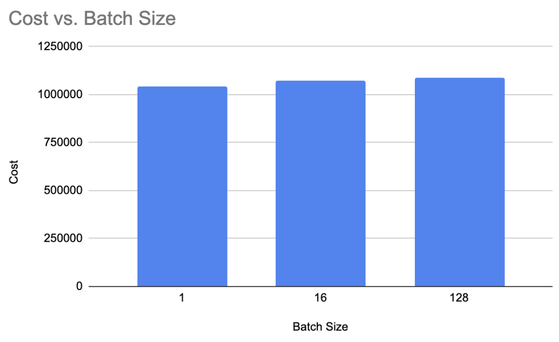
The total cache misses increases as we increase the number of processors, but the per-thread cache misses decrease as we increase the number of processors. Since this is MPI, all the processors have a private address space and their own cache. The decrease in the number of cache misses as the number of processors increases is due to the fact that each processor is working on a smaller set of data. This smaller working set can lead to better cache utilization for the processor, decreasing the number of cache misses. However, since there are more processors, the aggregate number of cache misses increases, explaining the increase in total cache misses. There aren't any surprises in the cache misses graph; it is as we expected.

These numbers correlate to the trade-off between increased parallelism versus communication cost we see in the speedup graphs. The decrease in per-thread cache misses correlates to the increase in efficiency per processor. Less cache misses means that the processor runs more efficiently within its local workload. However, the increase in total cache misses relates to the additional overhead of having more processors. The benefit from reduced per-processor cache misses can be offset by the increased communication overhead. This increased communication overhead is reflected in the increase of total cache misses.

Sensitivity Study - Batch Size:



(a) Total & Computation Time vs. Batch Size



(b) Cost vs. Batch Size

Figure 4: Sensitivity to batch size studies on hard_4096 test

The total time and computation time follow the same trend as batch size increases. We see that as the batch size increases, the total and computation time decreases. However after a certain threshold, the increasing the batch size doesn't impact the total and computation time. Increasing the batch size results in a faster time because as batch size increases, communication overhead decreases. This is because all our communication happens after each batch, which is less frequent as batch size increases. However, once the batch size becomes greater than the number of wires that the processor has to complete, the time does not decrease. This is because past this point we are always only doing one communication or synchronization routine per iteration.

The cost increases slightly as the batch size increases. The trend is similar to the relationship between batch size and time. Since the batch size of greater, the processors are getting updates less frequently, meaning that they are calculating the placement of more wires without updated information from the other processors. This leads to less accuracy in the placement of the wires, increasing cost. However, this cost increase is very minimal because, as we were experimenting, we noticed that the number of wires placement that need to be changed decrease with each iteration. This means that total updates to the wires is not that large. This means that wire placements get ”outdated” infrequently. Thus the cost increased due to old data mentioned above will not be that large. Furthermore, the threshold explained for batch size versus time also applies here.

PSC Experiments

Note: All experiments were performed on the hard spec with default parameters unless otherwise specified.

Speedup Graphs

We can see that total speedup scales rather poorly, this is primarily due to two factors: increasing initialization time, increasing communication overhead with more processors. Initialization requires broadcasting large data structures to all nodes, this scales poorly as the number of processors grows, leading to increasing initialization times. For large thread counts where each processor has a much smaller workload the total computation time is dominated by the initialization time leading to poor total speedups for larger thread counts.

For computations speedups we see an interesting pattern, sub linear speedups for threads 1 to 64 and then near linear speedup from 64 to 128. Looking at the hardware of a PSC RM node

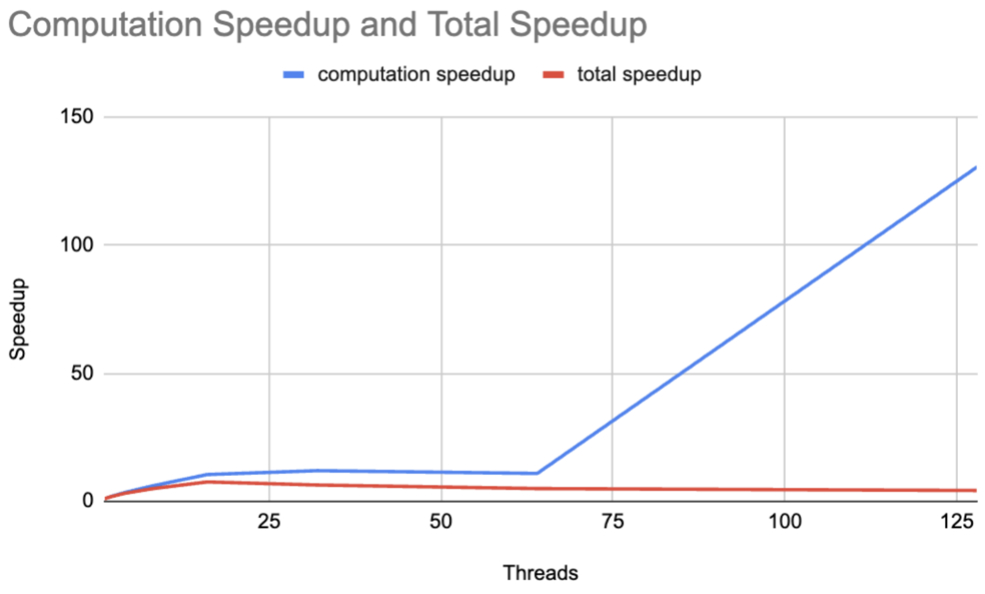


Figure 5: Computation and Total Speedups vs Threads for One PSC RM Node

Attribute	**PSC**	**GHC**
CPU(s)	128	8
Thread(s) per core	1	1
Core(s) per socket	64	8
Socket(s)	2	1
L1d cache	32 KiB	32 KiB
L1i cache	32 KiB	32 KiB
L2 cache	512 KiB	256 KiB
L3 cache	16384 KiB	12288 KiB
NUMA node(s)	2	1
NUMA node0 CPU(s)	0-63	0-7
NUMA node1 CPU(s)	64-127	-

Table 1: Comparison of PSC and GHC Hardware Specifications

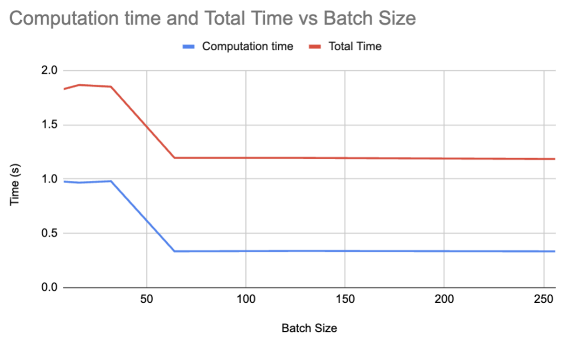
summarized below in Table 1. we see that the threads are mapped one to each core, and the cores are evenly split across two sockets. Utilizing all 128 threads across both sockets leads to better speedup for this MPI algorithm likely because we have larger memory bandwidth when utilizing both sockets. Since each MPI thread operates on the same batch of wires, we have great spatial locality, now with each NUMA node having it's own cache and memory there's less contention for memory resources likely leading to less evictions/misses. The effect of better speedups when scaling across two sockets likely stems from this improved memory bandwidth.

Comparison with GHC machines

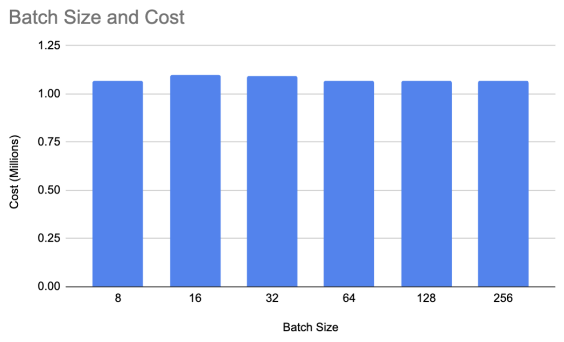
The patterns observed with speedup are largely similar as those discussed with the GHC machines, the key difference comes in the interval of threads 64 to 128 which was discussed in the previous subsection. We can also observe greater divergence between total speedup and computation speedup that occurs at higher thread counts.

Sensitivity Studies

In terms of sensitivity to batch size we see a similar pattern as the GHC machines. The initialization time remains about constant across batch sizes, this is to be expected since any intiialization is compeletely independent of batch size, due to this we observe practically identical trends for total and compute time. We observe initial reductions in time, until it plateaus for the remainder. Initial increases in batch size see performance gains as the communication overhead is incurred less frequently, we see the performance plateau after we reach a point where the batch size completely encapsulates one processors share of the wires.



(a) Total & Computation Times vs Batch Size



(b) Cost for Different Batch Sizes

Figure 6: Performance and Cost Plots for Batch Size Variations

In terms of cost we see no strong correlation between batch size and cost. This is likely due to the fact that over five iterations the algorithm nears convergence for all batch sizes tested. If we were to plot the total cost after each iteration we would likely see that it converges faster for larger batch sizes, however at the end of all five iterations there is not much difference.

Scaling Across Multiple Nodes

For scaling across nodes we can see that the single node case total time decreases till 16 threads and then starts to degrade. With two nodes we see that performance scales better to higher threads per node with minimal performance degradation. Since each processor routes the same set of wires our program exhibits good spatial locality, when scaling across multiple nodes we have more memory bandwidth which means better cache access patterns i.e. fewer evictions/ misses.

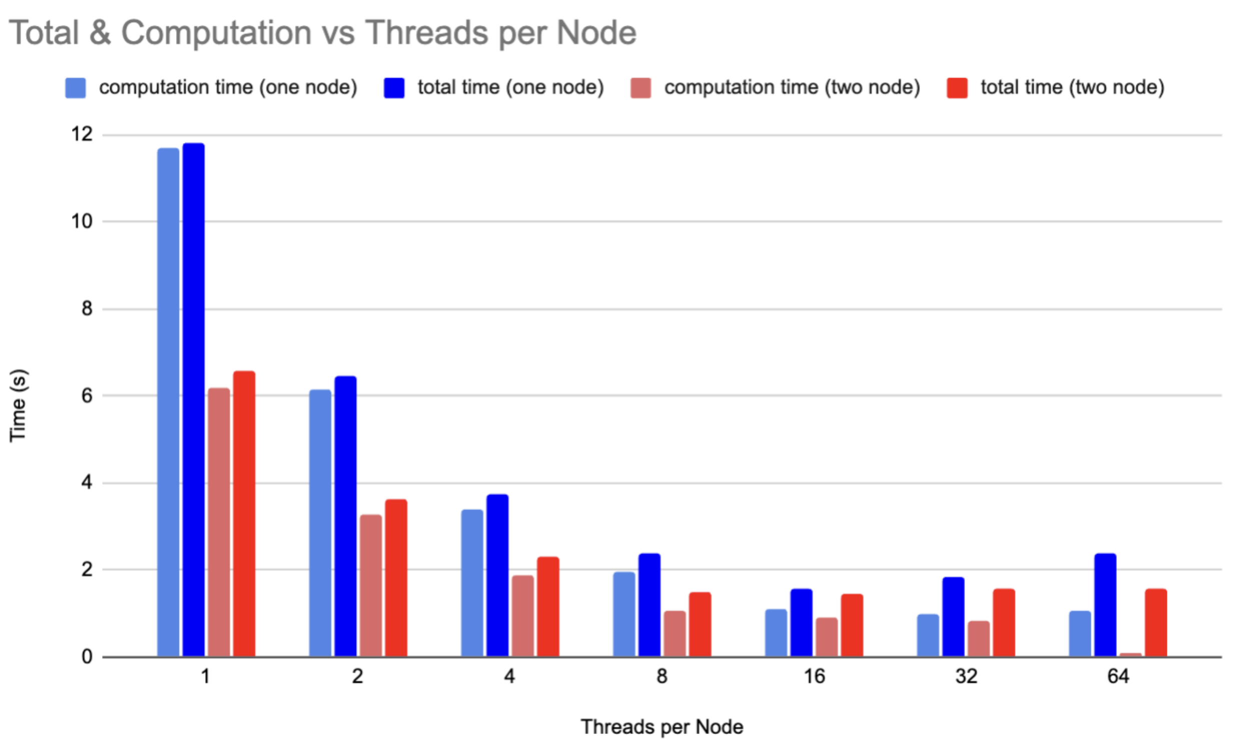


Figure 7: Performance for Scaling Across Nodes with Different Threads per Node