Parallel Linear Programming Solver Using MPI

Yuan(Alex) Guan and Yixuan(Ivan) Zheng

https://github.com/IvanLenn/15-418-Project

May 6, 2024

1 Summary

We implemented and optimized parallel versions of Linear Programming (LP) solver based on the Simplex algorithm with message-passing interface (MPI). We experimented our program on the GHC Clusters and PSC machines by applying the LP solver to max-flow problems, achieving over 70x speedup with 128 processes on large inputs (4000 edges). On smaller inputs, our algorithm still demonstrates good scaling in terms of memory and communication overhead.

2 Background

- Linear programming (LP) is a technique for the optimization of a linear objective function. The development of better and faster LP algorithms is an interesting ongoing area of research. Given a set of variables and constraints on them, we want to find an optimal assignment of values to the variables. In standard form, an LP can be written as:

  $$\text{For a variable vector x: maximize } c^T x$$

  $$\text{subject to } Ax \leq b$$

  $$x \geq 0$$

  The Simplex algorithm is a common method to solve LP. Starting with a corner of the feasible region, we repeatedly look at all neighboring corners of our current position and go to the best one. When we reach a position without better neighboring corners, it means we have found the optimal solution because the feasible region is convex.

  This algorithm can be greatly customized by large numbers of parameters and tolerances. We wish to speed up the algorithm using parallel techniques with message-passing.

- When implementing the sequential Simplex algorithm, we first examine whether the linear program is feasible (namely, whether there exists a solution that satisfies all the constraints.) To do this, we choose

one variable x_i from the variable vector x. We increment this variable and pivot to a vertex in the polytope. Then we check whether any constraint is broken. We pivot among the constraints to try to satisfy all of them. If eventually, we find any constraint impossible to satisfy, we claim that this linear program is infeasible. Otherwise, this linear program is solvable.

To solve a feasible linear program, we search along the edges of the polygon. At each iteration, we find a constraint that by incrementing the variable we increase the objective value the most. We call it a pivot. By maximizing the value of the variable in this constraint, we get a new vertex on the convex hull. We keep performing this procedure until we cannot further improve the objective value.

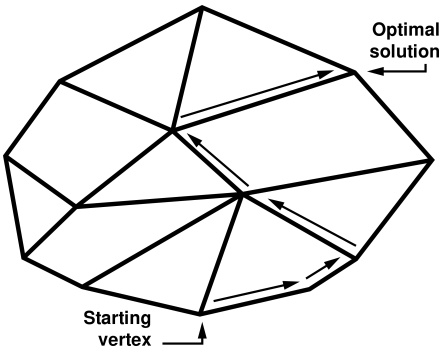

Figure 1: Simplex algorithm on a convex hull

- We used a 2D array to represent the input constraints. To output the answer, we have a data structure that has three statuses: (i) if raises the "Infeasible" flag, it means that there's no solution to the linear program; (ii) if raises the "Unbounded" flag, it means that the optimal answer is unbounded (not finite answer is optimal); (iii) otherwise, the LP is bounded and feasible, then we store the assignment of values to each variable in a 1D array.

- First and foremost, LP solver algorithms are often highly sequential in nature as we are finding each new vertex based on the one found in the previous iteration. This dependency makes it unintuitive to divide the work of the whole algorithm directly into multiple processes.

  At each iteration, we find a new vertex on the convex hull to pivot the constraints. This can be highly computationally expensive because we need to iterate through the constraints in order to find a pivot point.

  Two different approaches are considered by us to parallelize the algorithm at the beginning.

  i. We let each thread store a portion of the tableau and synchronize when information from other portions of the tableau is needed for some process.

  ii. We let each thread store a copy of the tableau and explore a distinct path along the border of the feasible region.

  For the first approach (partitioning the tableau), the main challenge is to design a proper data partitioning strategy so that the communication cost is minimized (each thread will have the information it needs in as

many cases as possible) and the work assignment is not so unbalanced. For the second approach (where we try different paths in parallel), the main challenge is to work out an efficient synchronization schedule. This is because the performance of this approach highly depends on how we select a well-performed path and keep the communication cost low at the same time. Overall, as we are using message-passing, data partitioning and communication are always two major concerns.

Meanwhile, locality can be an issue for some inputs. The distribution of input constraints on the matrix can be very random. As a result, when we are pivoting the constraints, it might appear that the next vertex to move to leads to memory accesses with poor spatial locality because we are moving between distant rows of the matrix.

- Linear programming offers a robust framework for addressing a variety of complex problems across multiple fields, including network theory, game theory, and operations research.

  In network theory, the max flow problem can be formulated as a linear programming problem to find the maximum flow from a source to a sink in a flow network. Each edge in the network has a capacity, and the goal is to maximize the flow through the network without exceeding these capacities. By setting up an LP model where the variables represent the flow on each edge and the constraints ensure that the flow does not exceed edge capacities and satisfies conservation of flow at each node (input flow equals output flow at every node except the source and sink), linear programming techniques can efficiently compute the maximum flow.

3 Approach

- **Technologies used**

  We implemented the program with C++ and used MPI to design the parallel version. We chose C++ because it has excellent support for MPI interface for parallel programming with MPI. We used GHC machines for 8-core CPU testing for experiments for our basic implementation. We also used PSC machines for experiments with higher process counts (up to 128) with single node.

- **Parallel approach 1: Path Parallel**

  The first approach we tried was to parallelize the path to search on the polygon. We let each process keep its own copy of the matrix, but each of them starts searching from a distinct vertex of the convex hull. In this way, we are able to avoid some expensive paths and output the result we find with the quickest search direction.

  Then we modified this approach by adding periodic synchronization: for every constant number T (batch size) of iterations, we synchronize the results of all processors and choose to continue with the most progressing thread. We use the result of this thread as the starting point of the next T iterations. We keep the batch size relatively high in order to avoid too frequent synchronization and communication

across threads.

The problem with this approach is that although it avoids the more computationally expensive cases, it does not effectively parallelize the computationally expensive part inside the algorithm. The communication overhead to update the results of searching among threads also limits the performance. In order to effectively parallelize the computationally expensive portion of the algorithm, we move to our second approach.

- **Parallel approach 2: Data Parallel**

  There are several iterations of optimization before arriving at our final version. We will give a detailed description of our final approach, and along the way also state what experiments and optimizations lead us to those ideal design choices (notice each **Optimization** section). Our final version modifies the Simplex algorithm to make it more MPI and asymptotic-friendly, but all correctness proof follows from the Simplex algorithm. At the end of this section, we will also provide experiment results for those **Optimization** and our final design choice.

  There are 2 main sections of our algorithm: the data representation and the algorithm.

  1. Data representation

     Suppose there are m constraints and n variables in standard form, we can stack those constraints row-wise for a matrix (tableau) $T \in \mathbb{R}^{m\times(n+1)} \in O(nm)$. Our data parallel approach lets each process store a contiguous chunk of rows of the tableau, and a separate copy of the target array. Each p process is responsible for updating their portion of the tableau only. Figure 2 is a visualization of how we partition data over processes.

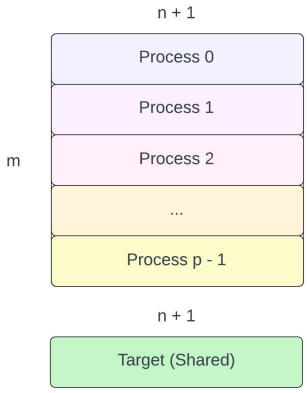

Figure 2: Visualization of how data are distributed among processes

Optimization 1 (memory; algorithm):

The classic simplex approach augments the tableau by adding an extra identity matrix $I \in \mathbb{R}^{m\times m}$ to the end of tableau. This is equivalent to introducing another m dummy variable representing each constraint. During later computation, the classic simplex algorithm maintains a set of m linearly independent columns (basis columns) during the whole simplex method.

Doing those eliminations is more intuitive and easier to code, but this brings much worse constants: the total tableau size is now $T \in \mathbb{R}^{m \times (n+m+1)} \in O(nm + m^2)$. In general LP problems where the number of constraints (m) is much larger than the number of variables (n) produce an asymptotically slower algorithm.

In our approach, we maintain the tableau of size $T \in \mathbb{R}^{m \times (n+1)}$ and another 2 arrays of total length $n + m$ representing indices of basis and non-basis columns. Later during computation, we modify these 2 arrays accordingly to keep track of the m linearly independent columns. This incurs some overhead linear to input size but greatly reduces the side effect of larger tableau and the cache misses it result in. We don't get into too much detail about why this modified algorithm is correct, because we focus more on parallel-related optimization.

Optimization 2 (communication; memory)

We use vectors as containers for storing all data in our previous version. There is no issue with vectors representing 1-dimensional arrays, such as the Target array, because we can still apply MPI collective communications with **vector.data()**.

However, when working with 2-dimensional array data all cache locality, memory efficiency, and communication cost is bad. Each 1-dimensional vector stores value contiguously, but the outer vector will allocate memory in different address. For example, **vector[0][n]** and **vector[1][0]** will incur a cache miss, and since they are not contiguous we need separate MPI calls to pass memory among processes.

Instead, we use 2 arrays to efficiently represent a contiguous chunk of 2-dimensional array. For example, suppose process 0 stores a Tableau of size $T \in \mathbb{R}^{\frac{m}{p} \times (n+1)}$. Then we use a double array of length $\frac{m}{p} \cdot (n + 1)$ under the hood for contiguous memory storage, and another double pointer array of length $\frac{m}{p}$ storing address of the first element of each row. The double pointer array provides using convenience and we get our contiguous memory property back. Figure 3 gives a visualization of advantageous contiguous memory storage.

For example, we can use one MPI call **MPI_Scatter(MatrixData, ...)** to distribute all data at once, instead of making $O(\frac{m}{p})$ calls of **MPI_Scatter**. This also provides nice cache locality, as the process need to iterate rows constantly.

2. Algorithm

Figure 4 provides a detailed flowchart of our optimized simplex algorithm. We use triple arrows to indicate each collective communication function we make. Single-threaded execution, which we want to avoid, is denoted as a single rectangle with dotted boundary. The ideal parallel block is denoted as a 3-stacked rectangle with a solid boundary.

The runtime is mainly determined by 3 key functions, which we will explain in great depth:

(a) Eliminate

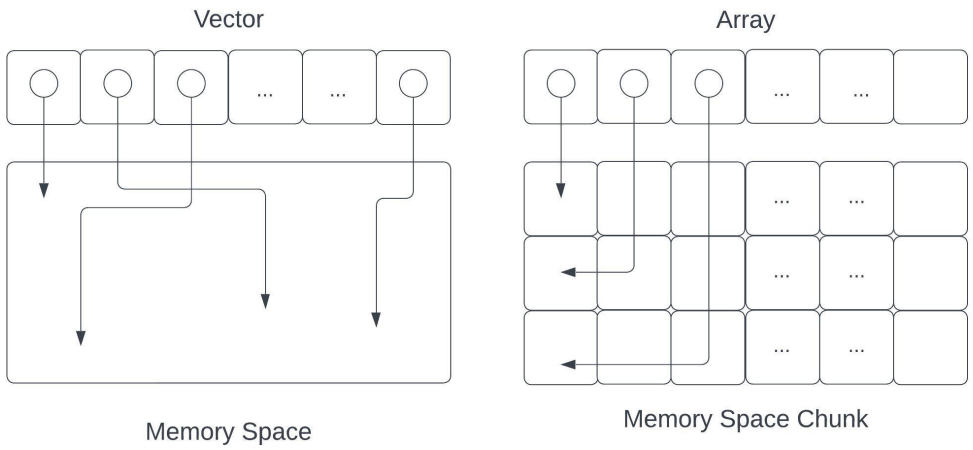

Figure 3: Visualization of how tableau memory are stored

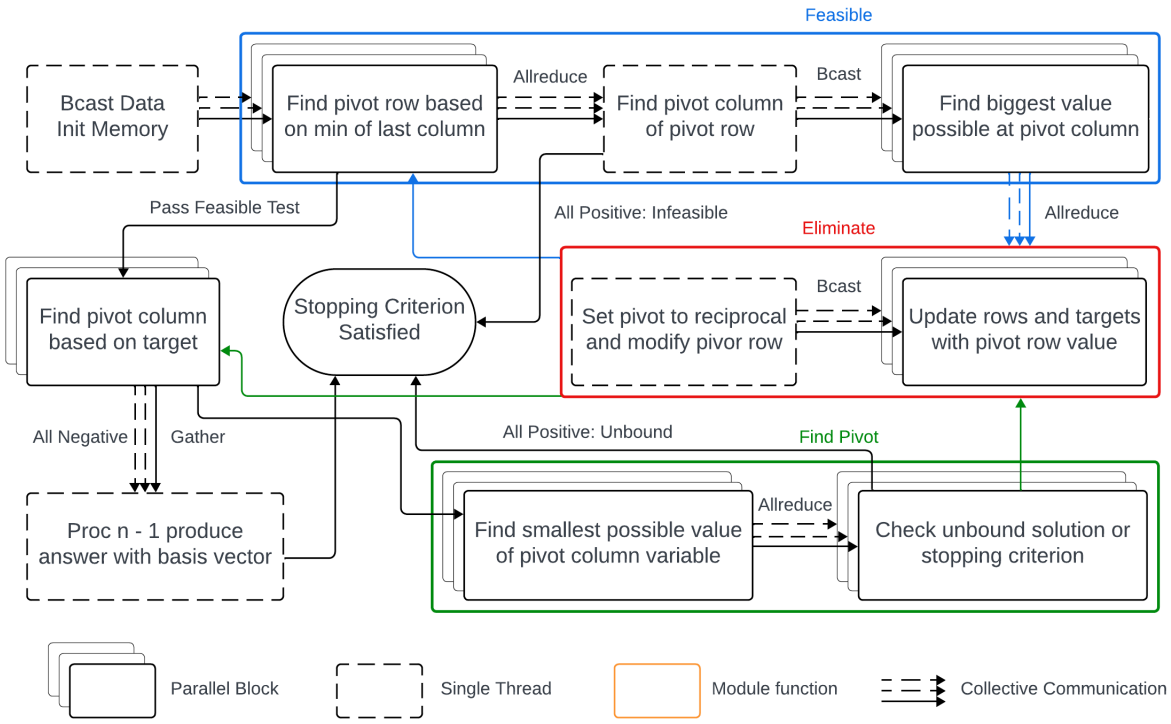

Figure 4: Flowchart of modified Simplex algorithm

This is the most computationally heavy function. Given input of **pivot_row** and **pivot_column**, we are modifying every element in the tableau, similar to the Gauss-Jordan elimination.

The flowchart of **Eliminate** contains 2 blocks. The first block is executed by the **pid** containing **pivot_row**. The algorithm requires us to modify the **pivot_row** to its entirety, and modification of all other rows depends on the results of **pivot_row**. Therefore, after **pivot_row** gets modified, we call **MPI_Bcast** on $n+1$ elements to broadcast the whole row to all other processes. This dependency is necessary because in **Optimization 1** we use a different representation of tableau for asymptotically less memory and calculation. There is a little overhead of divergent control

flow here because while this "lucky" process is computing other processes are stuck waiting for the broadcast data. This $O(n)$ factor is still preferable compared with $O(\frac{m^2}{p})$ more work at every process as typically $m \gg p$ (otherwise it just doesn't make sense to have more processes).

In the end, each row independently modifies all their tableau and target data and returns the function call.

To quantify the communication-computation ratio, we treat each individual computation as 1 unit. Each process does $2 \cdot \frac{m}{p} \cdot n$ computation; and there is 1 **MPI_Bcast** of n elements.

Optimization 3 (communication)

In this Eliminate function we only have 1 call to **MPI_Bcast**, which is the best we can get other than storing the whole tableau in each process. However, it is a huge broadcast with n elements that is blocking.

We tried to apply **MPI_Ibcast**, the asynchronous version, but it doesn't work well (no improvements detected). In retrospect this also makes sense: by dependency relation, literally no extra work can start before receiving the data. Adding those function calls even might incur some additional overhead.

This leads us to another thought: applying multiple **MPI_Ibcast** with each call sending a chunk of data. Our initial hope is to find the "sweet" spot of balancing the number of calls and the number of data each call. For example, suppose each process needs to receive n units data to make $\frac{m}{p} \cdot n$ calculations. It takes a longer time (relatively) to get all the data. If we send k packages of $\frac{n}{k}$ data asynchronously, and after receiving any chunk we start doing the computations on those data, ideally we can start computation earlier!

During our experiments, we set a granularity of k chunks for k ranging from 5 to 50. The result is not very interesting (no improvement). The possible explanation is we strive to reduce communication over small inputs (for flows with edge number in hundreds) and under small inputs with about 50 vertices the overhead of doing so dominates that of optimization. We expect this trick to work well on larger graphs, but larger graphs already have very nice scaling performance so this optimization doesn't get noticed.

(b) Feasible

As the name suggests, our solver calls this function once to determine if the linear programming system has any solution at all (i.e. the polytope of feasible regions in not empty).

This function is a big while loop doing those 3 blocks on top of chart, calling **Eliminate**, and returning when feasibility is defined. We can show this function will enter the while loop no more than m times, so in terms of performance it's usually ignorable compared with the next section.

Based on the flowchart, each process first computes their local minimum of the last column, and they call **MPI_Allreduce** for the global minimum. If the global minimum is non-negative, we automatically have a solution, which is setting all variables to 0, thus quitting this function.

Otherwise, one "lucky" process will find the **pivot_column** based on **pivot_row**, and broadcast its choice to all threads. If the condition is not met, we can state there is no solution, thus quitting this function.

In the end, each rows with indices bigger than that of **pivot_row** will collective generate a new **pivot_row** with 1 call to **MPI_Allreduce** and then call the **Eliminate** function. The fact that we only operate on rows with bigger indices ensures the program to execute the while loop at most m times.

For each while loop iteration, each thread does $2 \cdot \frac{m}{p}$. In terms of communication, there are 2 **MPI_Allreduce** of 2 elements and 1 **MPI_Bcast** of 1 element.

(c) Find_pivot Given each process has an identical copy of the Target array, they each find the **pivot_col** separately. Note here they are doing repetitive work, but its advantage will be discussed in **Optimization 4**. If the Target array is non-negative, we have find the optimal answer. From the flowchart, we enter the bottom left box, where each process sends process $p-1$ their copy of the last column through a **MPI_Gather** of m elements in total. The last process then retrieves the answer from the basis vector array. This block will execute at most once before returning, so we are not counting this collective communication function call into account because it is negligible (the number of iterations is high, so we will make thousands of calls to the other functions).

Otherwise, all processes will find the smallest value of the pivot column variable, calling **MPI_Allreduce** for a consensus **pivot_row**, and enter the **Eliminate** function call as described above.

During each iteration, each process does $n + \frac{m}{p}$ work. There is also 1 call to **MPI_Allreduce** of 2 elements.

Optimization 4 (communication)

In our previous versions, only process $p-1$ has the correct copy of the Target array, and for other processes, the Target field is just nullptr. We later enforce each process to maintain a copy of target array, and maintain the invariant that the target array of all processes at the end of any **MPI_Barrier()** are identical.

We can get rid of 2 calls of **MPI_Bcast** at the start of **find_pivot** because each process already has the information required.

This optimization greatly reduces the communication-computation ratio, and also reduces interconnect traffic. We reduce the call to **MPI** function from 3 to 1 in this function, and results show we do get decent improvement. Moreover, there is no overhead associated with this optimization. Even though each process are doing redundant work, in our previous version all but one process are spinning doing nothing for data still. This provides a more consistent control flow in that all processes are executing the same operations, which could potentially benefit further optimizations on using SIMD or similar tricks.

Since all of our algorithms differ only within iterations (we always select the same **pivot_row** and **pivot_col** and updates identically), we can just consider communication-computation cost per iteration. Moreover, in cases of divergent control flow where all other processes are waiting for the "lucky" process, we also count its time onto every process, which is reasonable. We know both **feasible** and **find_pivot** made one call to **eliminate**, so adding up their communication and computations we derive the following data in table 1:

Functions	Computation	Communication
Eliminate	$\frac{2 \cdot m \cdot n}{p} + n$	$1 \times$ MPI_Bcast of n
Feasible	$\frac{2 \cdot m}{p} + \frac{2 \cdot m \cdot n}{p} + n$	$2 \times$ MPI_Allreduce of 2 $1 \times$ MPI_Bcast of 1 $1 \times$ MPI_Bcast of n
Find_Pivot	$2n + \frac{m}{p} + \frac{2 \cdot m \cdot n}{p}$	$1 \times$ MPI_Allreduce of 2 $1 \times$ MPI_Bcast of n
Answer	$n + \frac{m}{p}$	$1 \times$ MPI_Gather of $\frac{m}{p}$

Table 1: Communication / Computation intensity of final algorithm

Optimization 5 (communication; memory; algorithm)

We also experimented our approach with different data partitioning schemes. Another possibility is to partition data column-wise, meaning each process get a portion of all constraints given. The intuition behind this approach is we are making 3 calls to **MPI_Allreduce**, and therefore those 3 operations naturally fit with column-wise partition.

The structure of column-wise partition still comprises of those 3 main functions, with each block in our flowchart getting "transposed". In other words, in places where we use **MPI_Bcast** for row-partition, we might need **MPI_Allreduce** in the column-partition setting, which might get slower because **MPI_Allreduce** incurs more overhead than **MPI_Bcast**. The good side is we can have 3 less **MPI_Allreduce** as discussed in the previous paragraph. We also did some calculation, and present the calculation versus communication table for column-wise partition method in Table 2:

Functions	Computation	Communication
Eliminate	$\frac{2 \cdot m \cdot n}{p} + \frac{n}{p}$	$1 \times$ MPI_Allgather of $\frac{n}{p}$
Feasible	$2 \cdot m + \frac{3n}{p} + \frac{2 \cdot m \cdot n}{p}$	$1 \times$ MPI_Allreduce of $\frac{n}{p}$ $1 \times$ MPI_Bcast of 1 $1 \times$ MPI_Allgather of $\frac{n}{p}$
Find_Pivot	$m + \frac{3n}{p} + \frac{2 \cdot m \cdot n}{p}$	$1 \times$ MPI_Allreduce of $\frac{n}{p}$ $1 \times$ MPI_Bcast of 2 $1 \times$ MPI_Allgather of $\frac{n}{p}$
Answer	$n + \frac{n}{p}$	$1 \times$ MPI_AllGather of $\frac{n}{p}$

Table 2: Communication / Computation intensity of column-wise partition approach

We can verify that total computation of column-wise and row-wise partitions are the same asymptotically,

with little different of constants. For communication, we notice **Eliminate** gets slower for column-wise because **MPI_AllGather** is much more time consuming than **MPI_Bcast**, as supported by our experiment results. Similarly, for the most dominant function **Find_Pivot**, column-wise approach uses one more call to collective communication function and thus will appear slower.

Finally, although the **Answer** function will be faster in column-wise approach, as it gathers $O(n)$ instead of $O(m)$ elements, this function is called only once so its effect is ignorable.

Optimization 6 (memory; algorithm)

We allocated a buffer of size not exceeding $m+p$ to temporarily store all data to be sent or received. This buffer is used every time by calling to **Eliminate** when broadcasting the whole row; and in **Find_Pivot** when finalizing the answer.

This optimization avoids the redundancy to allocate and free a temporary array frequently in the most used function **Eliminate**. In a classic LP program with millions of iteration, the constant boost of storing data in buffer is huge. It also benefits the specific MPI call of broadcast in **Eliminate** as the memory address their is held unchanged, thus allowing for more potential cache hits since we are using the same array.

The following table 3 lists all 6 main optimizations we had based on the initial version. It also indicates what aspects do those optimizations contribute to our final algorithm. With MPI, we are more concerned with reducing communication cost, and efficient memory management across processes. Therefore, we mainly analyze the reasons for our choice above, and present experiment results in those 3 aspects below.

Property	Communication	Memory / Cache	General algorithm
Optimization 1: Efficient Tableau		✓	✓
Optimization 2: Contiguous Memory	✓	✓	
Optimization 3: Ibcast Granularity	✓		
Optimization 4: Shared Target Array	✓		
Optimization 5: Different Data Partition	✓	✓	✓
Optimization 6: Message Store Buffer		✓	✓

Table 3: Categories of different optimization

We want to optimize test cases whose communication-computation ratios are at boundary. In other words, for very large test cases computation clearly dominates, and our optimization is not obvious. We experiment all following optimizations on flow problem with 40 vertices and 600 edges. We choose this set of data because in our first version after process count 16 the performance gets worse. And as we made those 6 major and several small optimizations, we observe optimization up to 64 thread counts on this input.

1. In **Optimization 3**, we experimented with different chunk number k for more calls to **MPI_IBcast**

with less message carried in each call. From figure 5 we can barely observe any difference, and they only differ after 3 decimal places after taking average of 5 runs.

As mentioned above, our interested test cases have 40 variables. Even with chunk number 5, meaning 8 elements per chunk, there aren't significant places for good parallelization. This experiment shows such optimization negligible under small test cases when the overhead of chunking shadows the benefit of more parallelization.

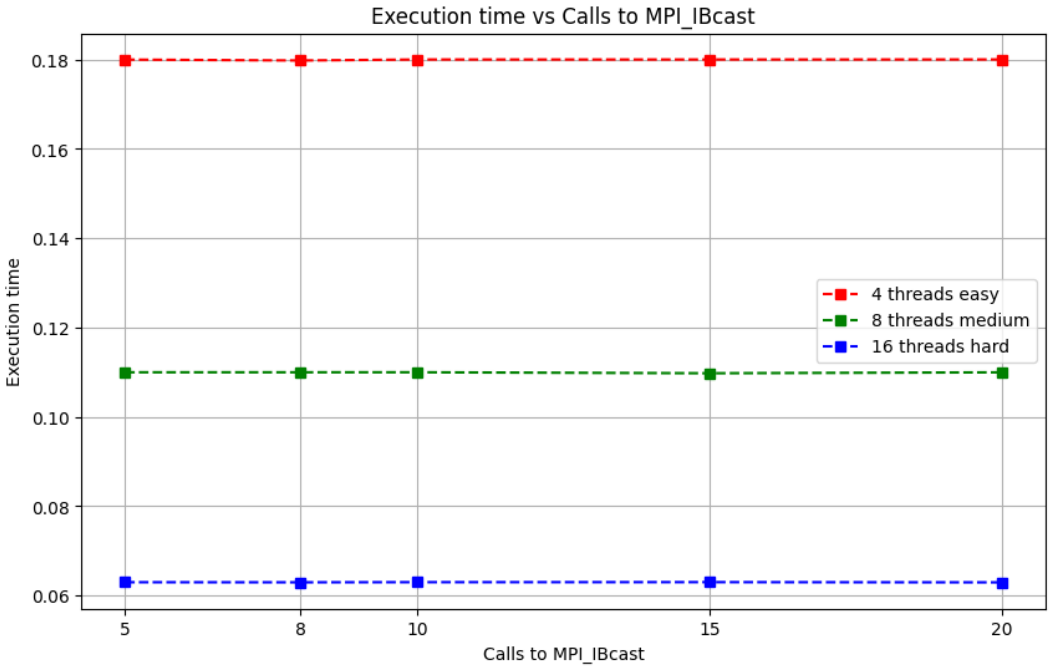

Figure 5: Different number of calls to MPI_IBcast vs. Execution time

2. Both **Optimization 2** and **Optimization 6** provides obvious cache improvement. Recall that **Optimization 2** optimizes cache locality and message passing overhead because all memory of the big tableau is contiguous.

   From Figure 6 we can observe a stable 10% improvement in cache performance. This is almost expected, because for input variable dimension 40, contiguous memory space ideally can provide $8/40 = 20\%$ cache improvement.

3. Figure 7 illustrates the runtime difference between row-wise (our final version) and column-wise approaches. We only run experiments up to thread count 8, because our input data has variable size 40. It doesn't make sense to experiment with higher process counts for the column-wise partition version, because then each process will get $2-3$ columns, which clearly will perform very bad because of high communication overhead.

   In retrospect, the nature of linear programming having much more constraints than variables make the row-wise partition ideal as each collective operation consumes less interconnect traffic; and most importantly most communication are carried row-wise, which gives row-partition the advantage of carrying out a lot of calculations locally without the need to communicate.

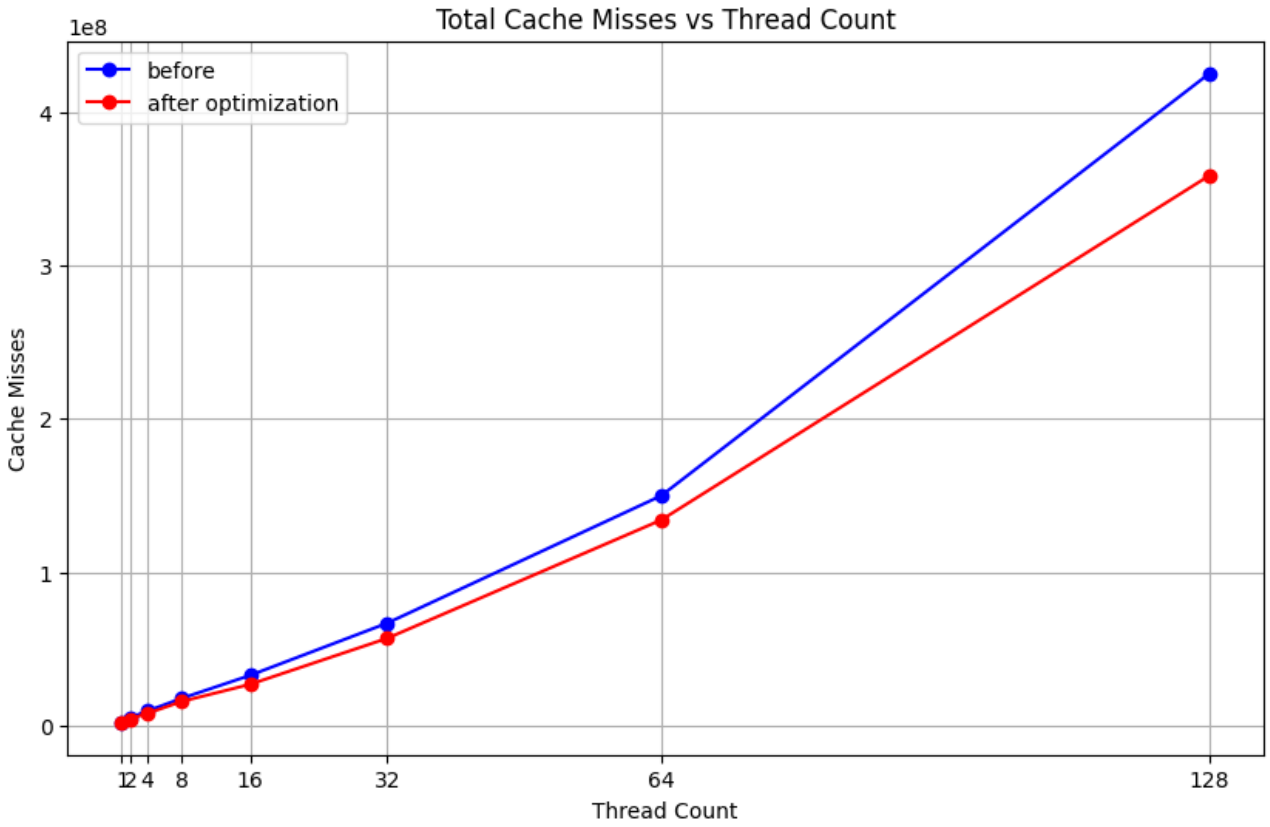

Figure 6: Cache Optimization due to contiguous memory and pre-allocated buffer

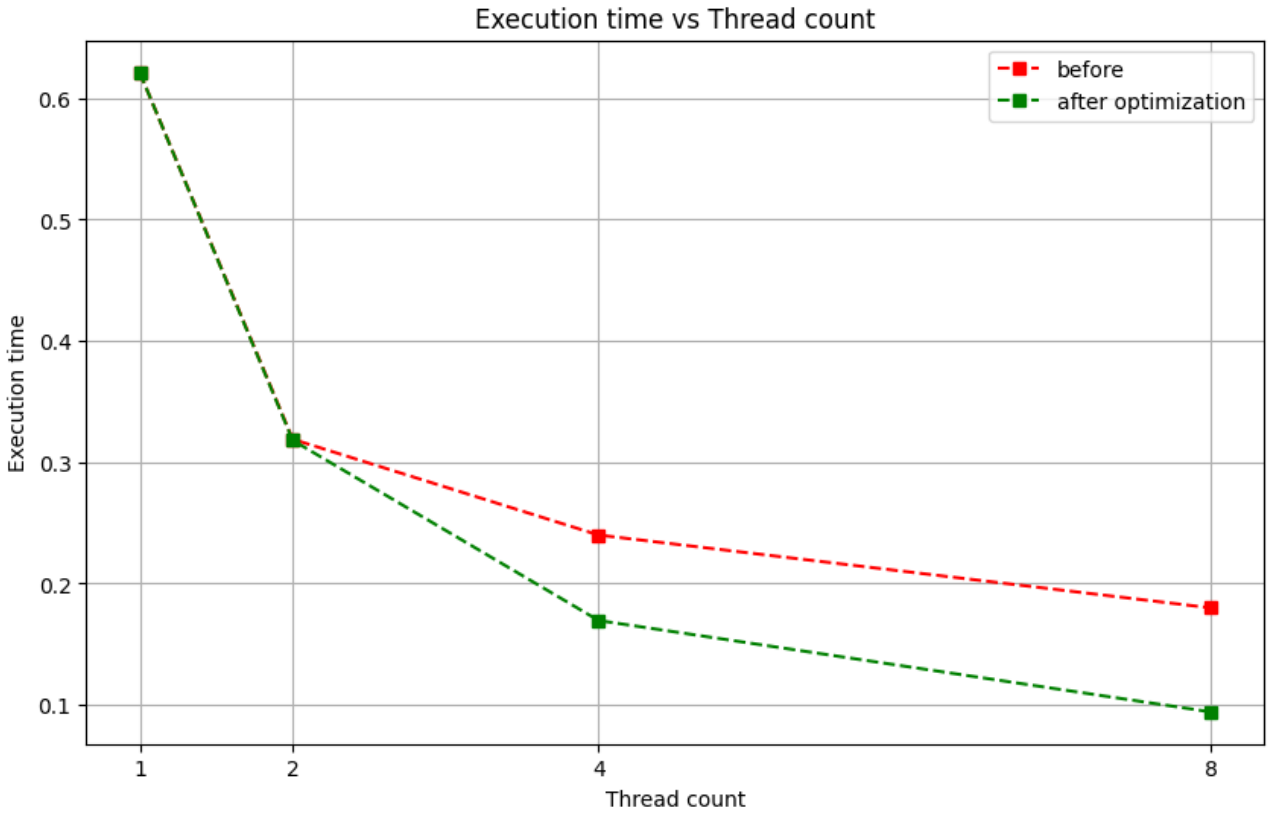

Figure 7: Different number of calls to MPI_Bcast vs. Execution time

4 Results

- We measured performance in terms of wall-clock time and speedup compared to running the parallel version on one single processor. Our experimental setup involves different sizes of flow input and different thread counts (we tested high thread counts on PSC). The "Easy" input contains 600 edges, the "medium" input contains 1500 edges, and the "difficult" input contains 4000 edges (each edge has a corresponding constraint). All edges are directed, and each edge is generated by uniformly choosing 2 *vertices* at random. We also do not exclude self-loops or multi-edges to make the input more generalizable. For detailed test case generation script please refer to our repository, under **tests/test_data/flow/generate_tc.cpp**.

- Speed & Execution Results

  Our baseline is the parallel implementation for a single CPU (after all above optimizations). In other words, all optimizations are relative to running our parallel program with **mpirun -np 1** on our final algorithm.

  We don't compare our speed-up with serialized implementation because all our optimizations involving cache and general algorithm make our final parallel version with 1 core 2x faster than the serialized code. For reference, the serialized code takes 234s on input **flow_hard.in**, while the parallel version runs 115s with 1 process.

  Experiments on GHC clusters (Figure 8 and Figure 9):

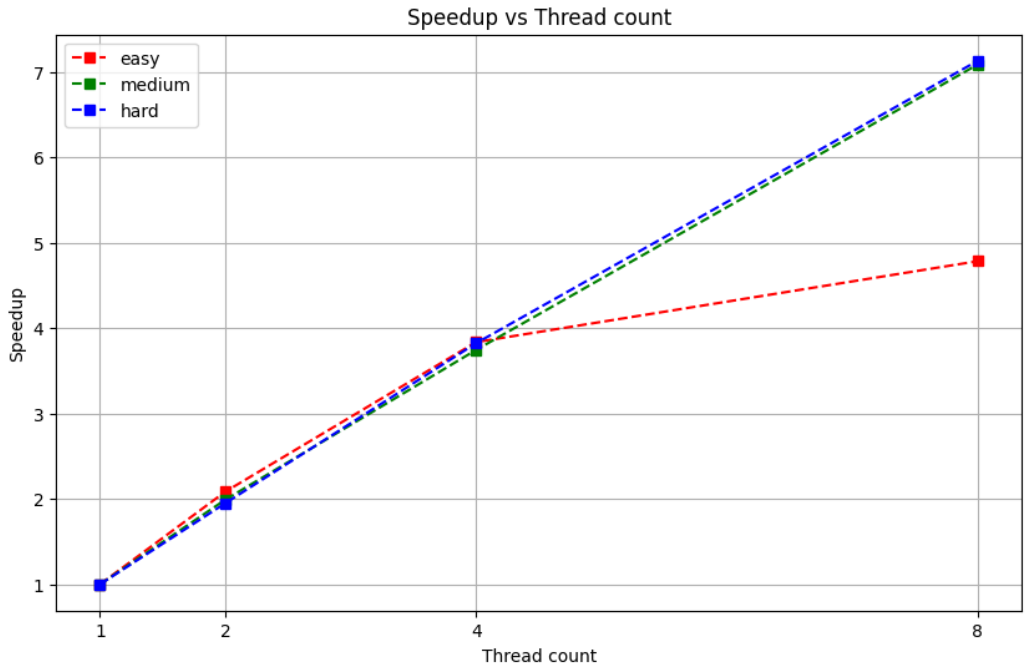

Figure 8: GHC machine results: speed up graph

  Experiments on PSC (Figure 10 and Figure 11):

  Generally, for relatively smaller thread counts, we observe a similar pattern in the speedup curves measured on the PSC machines to the GHC results. We mentioned above that our initial approach already scales

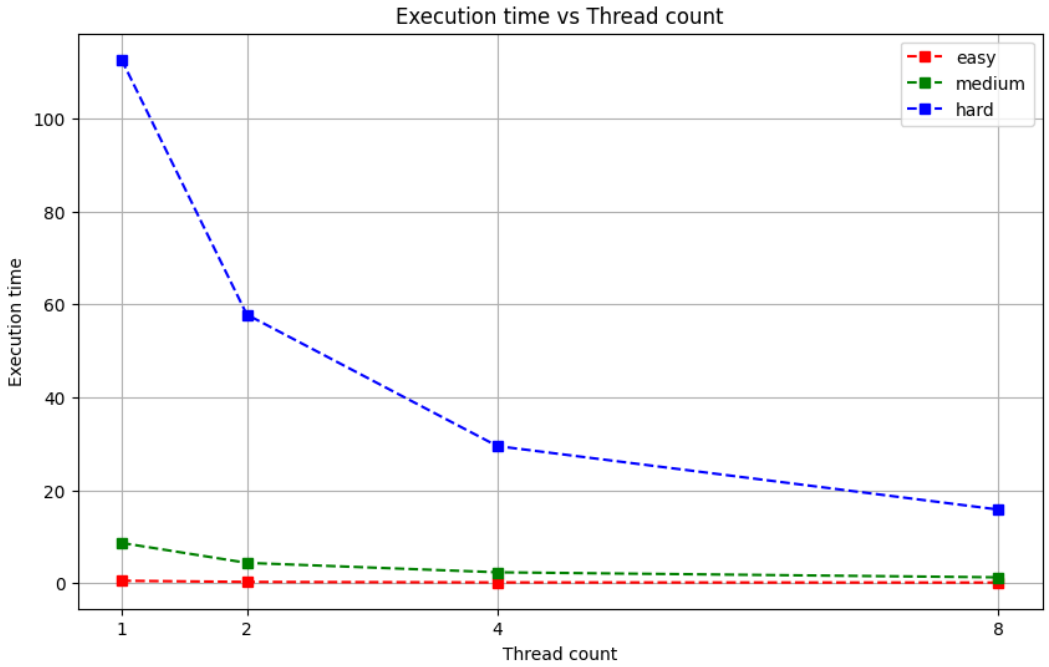

Figure 9: GHC machine results: runtime graph

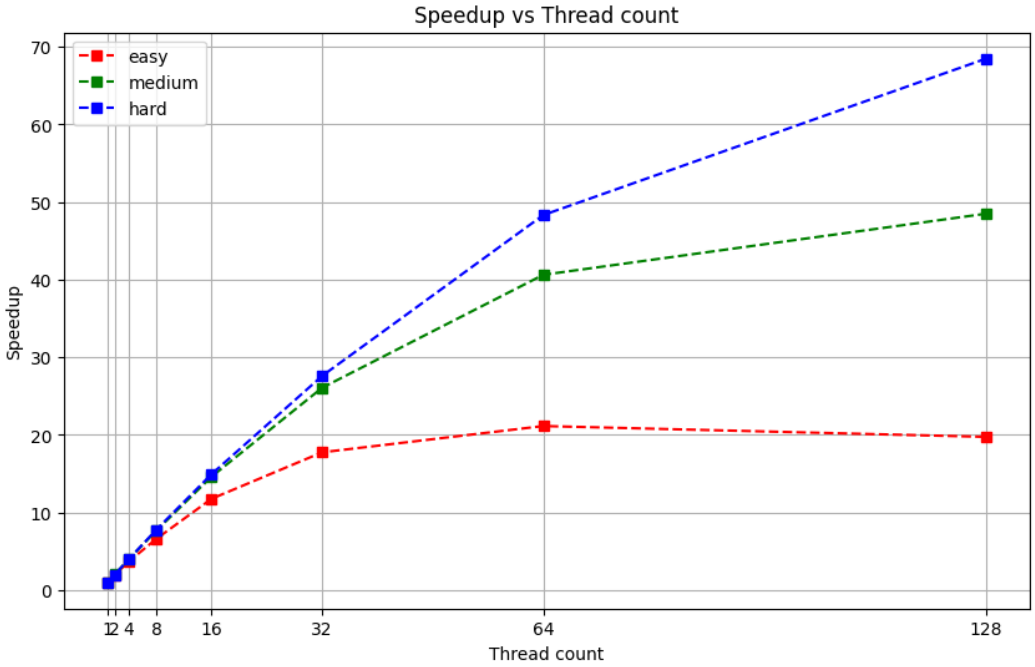

Figure 10: PSC machine results: runtime graph

to thread count 16, so this is not surprising. However, the results from PSC further reveal how the performance increases less fiercely with high thread counts.

- Sensitivity tests

  It is important to report results for different problem sizes for our project. For each thread count we compare their performance on different problem sizes. Since higher thread counts increase the communication overhead, we want to particularly compare the performance under different input sizes with higher thread counts.

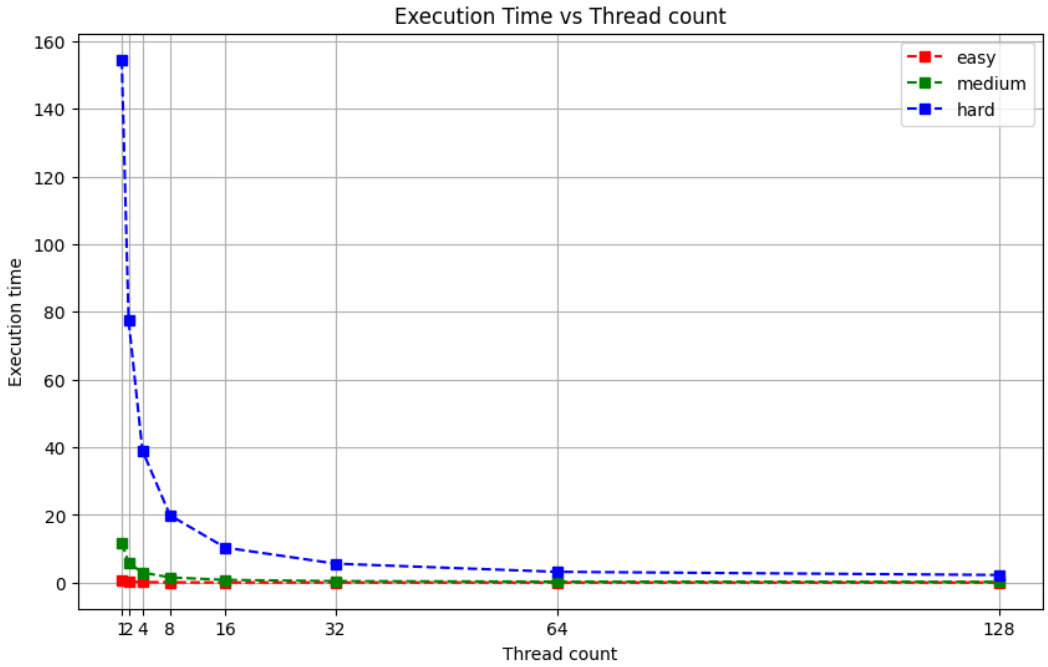

Figure 11: PSC machine results: runtime graph

We have 3 carefully chosen input sizes. For larger input (**flow_medium / flow_hard**), more expensive computation is involved, which reduces communication-computation ratio. The results do show that our program exhibit better scaling with a larger input size.

We are more concerned with the test case **flow_easy**, because that's how we demonstrate the effectiveness of our optimizations in reducing communication overhead. For this test case, each iteration on average does $\frac{5 \cdot 10^5}{p}$ computation and makes 3 calls to MPI functions. For thread count 128 that is almost 1 MPI function call per 1000 arithmetic operation.

From Figure 12, we see a significant boost by running on PSC machine with input **flow_easy**. Before optimization, we get almost 8x speed from 16 cores and no further speedup regardless. After all the optimizations we gone through, our program scales well until 64 cores with speedup of over 20.

- Cache Analysis (Figure 13 and Figure 14)

  We see that there are more total cache misses, and fewer per-thread cache misses, as we run with more threads, which supports the pattern of the speedup curves. As the thread count increases, the cache experiences more misses due to more demand for communication among threads. Having more threads also leads to more frequent contention. Therefore, the total cache misses increase as we increase the thread counts. As mentioned above, we observe a slight drop in the derivative of speedup at higher thread counts due to communication overhead, which is consistent with this pattern of cache misses.

- Speedup Limit

  First of all, the Simplex algorithm is quite sequential in nature as each iteration depends on the vertex determined previously to proceed. Therefore, we are not able to perform parallelization among iterations

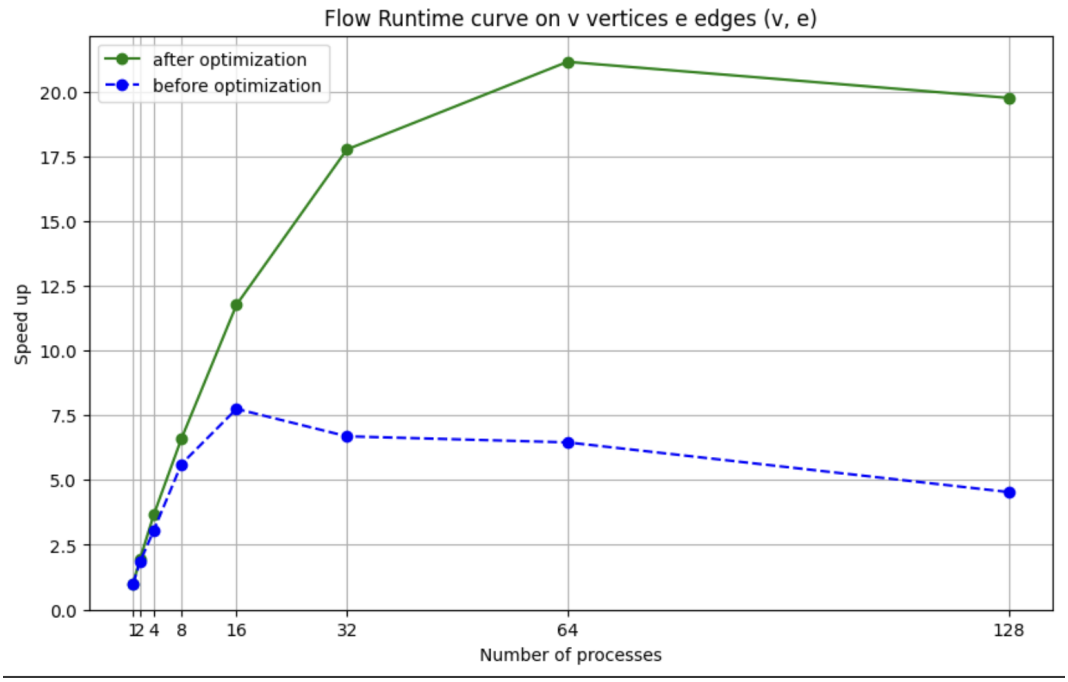

Figure 12: Speed up on PSC machines before and after optimization

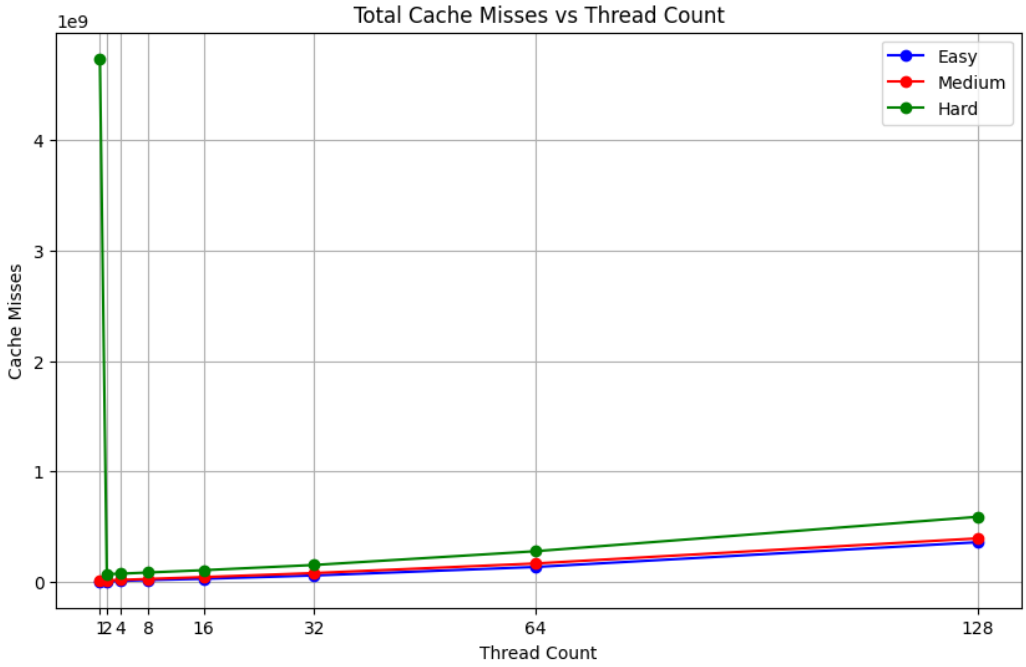

Figure 13: PSC machine results: total cache miss

of vertex searching due to this dependency. Theoretically, not following the heuristic given in classic Simplex algorithm might lead to infinite loop when updating the whole tableau, so we stick with the classic approach and do not parallel over iterations in the second approach.

Besides, as we are implementing the parallelization using message passing, communication and synchronization overhead limits our speedup. As revealed by the experiments we performed for cache analysis, the total cache misses keep increasing as we increase the thread count. Please refer to the last sections of **Approach** for a more detailed analysis for how we experiment and optimize communication cost. All

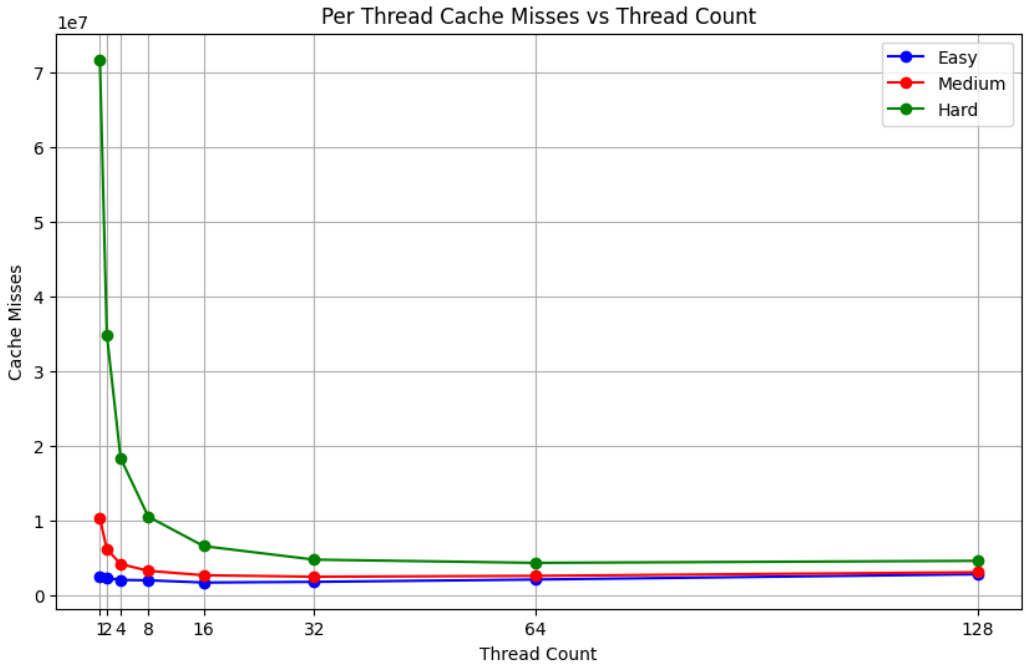

Figure 14: PSC machine results: per process cache miss

the optimizations we made are targeted toward less, and more efficient, MPI collective communication function calls.

- Deeper Analysis

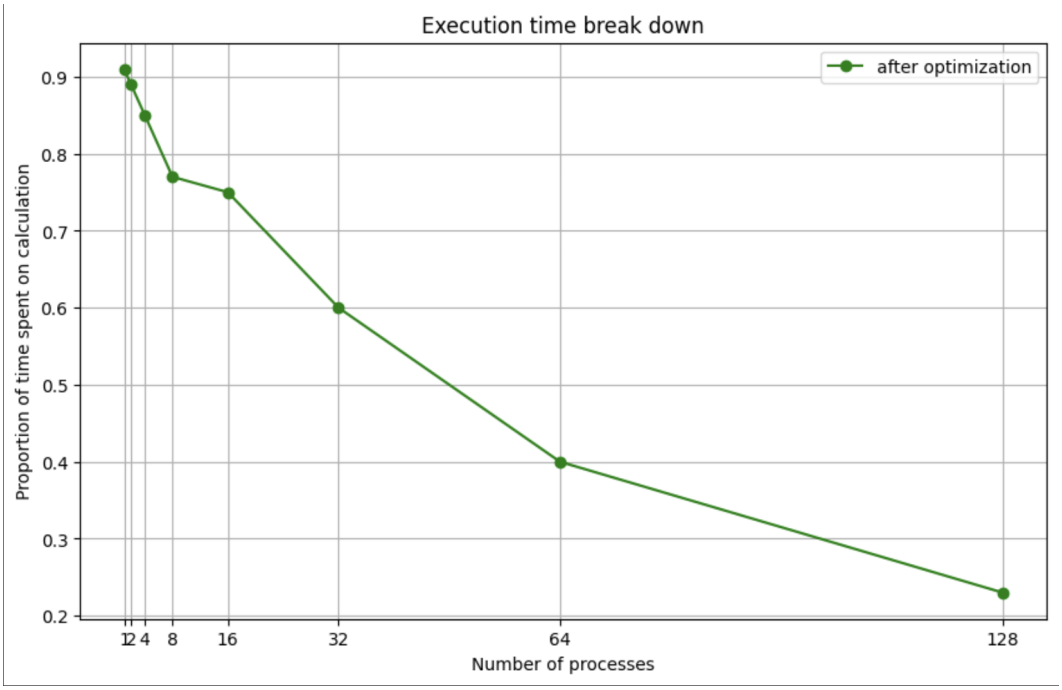

Figure 15: Runtime break down on PSC performance of easy test case

Figure 15 shows the proportion of pure computation time of different process over input **flow_easy**. This computation time only counts the major chunks of computation, therefore even for nproc = 1 we get

90% due to initialization overhead, and some trivial calculations not involved. We can observe the time spent on calculations decreases monotonically as the number of processes increases. This is probably due to much more communication overhead as we get more threads. With previous experiments we already observe roughly the same total cache misses over different processes, which means we don't spend more time on cache misses or memory latency. The only possibility is the communication cost that comprises the majority of runtime other than computation time.

With all our optimizations, our program still spends 70% time communicating on **flow_easy** with 128 threads. Compared to Figure 7 which makes the name number of MPI collective communication function calls and our current approach is 2x faster, we can conclude the major issue is that we are still making too many function calls. The message size of each call doesn't change the data significantly, meaning the interconnect traffic is not at its maximum. As mentioned above, 3 calls to MPI functions per 1000 calculations is still too high. For future optimizations, we should strive for a even more clever data partition scheme (if exists) so reduce the communication-calculation ratio even further for better scaling performance.

- Machine Target

  We chose CPUs instead of GPUs. One thing is that we focus more on optimizing communication across cores and threads instead of mapping repetitive computation to SIMD. Therefore, CPUs are more suitable for our direction of investigation. On the other hand, each thread in our algorithm is responsible for its own search of a local optimal, which does not fit the characteristics of SIMD utilization that maps small and repetitive tasks to processors. Therefore, our choice of machine target is sound.

5 References

Daniel Aloise Samuel Xavier-de-Souza Demetrios A. M. Coutinho, Felipe O. Lins e Silva. A scalable shared-memory parallel simplex for large-scale linear programming. 2019. 19

Tar, P., Stágel, B. & Maros, I. Parallel search paths for the simplex algorithm. Cent Eur J Oper Res 25, 967–984 (2017). https://doi.org/10.1007/s10100-016-0452-9

Notes from 15-451 which introduces linear programming and the Simplex algorithm: https://www.cs.cmu.edu/ 15451-s24/schedule.html

Notes from 21-292 which introduces simplex algorithm and its proof: Notes from F22 by Xinyue Yang and Michael Cui