Parallelizing A Linear Programming Solver using OpenMP

Authors: Charlie Phillips, Raghav Gupta

Summary:

We implemented a parallelized version of the Simplex Linear Programming algorithm using OpenMP. We ran our code on the GHC Clusters, PSC supercomputers, and the M1 Pro.

Background:

Linear Programs are a way to represent multiple algorithms, such as maximum flow or scheduling. Linear programs can be used to represent optimization problems and are an important topic in the field of operations research and logistics. As a result, an efficient linear programming solver is highly useful for this field. There exist multiple open-source solvers (e.g. GNU Linear Programming Kit, HiGHS) and closed-source solvers from companies such as Gurobi.

A linear program is specified with inputs:

- n real-valued variables $x_1, \ldots, x_n$
- A linear objective function using these variables as inputs.
- m linear inequalities (not strict) in these variables that act as constraints on our variables.

A linear programming solver algorithm outputs n values (one for each real-valued variable x_i) that maximize the objective function while satisfying these constraints.

We store our inputs in standard form. The data structures used are 1D and 2D arrays. The m constraints as a two-dimensional array, A. The values of the n variables are stored in a one-dimensional array.

At a high level, the Simplex linear programming algorithm starts at a valid vertex. It then explores its neighboring vertices and chooses the neighbor with best value. Since the n variables form a convex n-dimensional polytope, repeatedly choosing the local optimum vertex yields the solution to the linear program that maximizes the objective function while being valid.

The Simplex linear programming implementation first checks whether the linear program can be solved or not. This choosing the strictest constraint (the constraint that must be less than or equal to the smallest value). We choose a variable x_j to increase and pivot to a new vertex in the n-dimensional polytope. We see whether one of the constraints becomes impossible to satisfy after pivoting between constraints and attempting to satisfy them. If there is an impossible to satisfy constraint, this linear program can't be solved; it is infeasible.

If a linear program can be solved, we attempt to solve it. We start at the origin. Since the inputs are in standard form, each variable $x_i \geq 0$, so the origin (all variables are 0) is valid. Each iteration of the simplex algorithm, we find the constraint x_i that increases the objective function the most; this x_i has the largest constant c_i where c_i is the constant of x_i in our linear objective function. We then maximize the value of this variable given the constraints. Once we have maximized the value of this constraint, we pivot to a new vertex that is created by increasing the value of x_i from our original one. After pivoting, we do a change of basis that transforms our new vertex into our origin by shifting our coordinate system so our new vertex is the origin each iteration. Once we cannot increase the objective value, we return the values of each variable we currently have.

The data structures we use are the same 1D and 2D arrays for output and input. The simplex algorithm needs to be able to rapidly determine which input variable out of n variables can increase the optimum the most at the current feasible solution it is exploring. The simplex algorithm also needs to rapidly edit constraints as it searches for the optimal solution and the values of the variables change. We kept the 1D and 2D array based implementation out of the need to edit multiple elements in our constraint array A and find the maximum element quickly. Since changing the coordinates after pivoting can involve $O(mn)$ array updates, we use an array to avoid adding a log(n) factor with the updates.

Each iteration of the simplex algorithm depends on the new vertex in the n-dimensional polytope found in the previous iteration. This is the main dependency, so we need to parallelize across work done in each iteration.

The computationally expensive part is the work done in each iteration of the algorithm. Given the large number of constraints and variables, A can be large. Editing A while pivoting to make the new vertex the origin is expensive. Finding the variable to change and how much we can change the variable by are computationally expensive can be well. We can use data parallelism to compute and edit the new values of A in parallel. However, we have synchronization issues with finding the best neigbhoring vertex since if we split the neighboring vertices up across the cores, we need to synchronize.

Finding the best vertex to move to is expensive via SIMD. This involves finding one element in each vector. There isn't an operator for max in a vector. SIMD is good for updating the new values of A in parallel, as scaling constraints by the same factor can be parallelized by a vector instruction.

Locality is an issue. Each variables x_i is represented as a column in our matrix A. Each constraint is a row in A. Finding the best vertex to move to (which depends on the best variable to increase) and then modifying the constraints will involve memory accesses that have poor spatial locality since we are accessing elements across a column in a 2D matrix.

The Approach:

We decided that our first approach to the problem would be shared-memory parallelism through OpenMP.

Our OpenMP implementation of the Simplex algorithm is written in C++ and uses OpenMP. We targeted the GHC cluster initially. We parallelize the work inside each iteration of the simplex algorithm. We use OpenMP reductions to find the best variable to change by iterating over the variables and how much we can change that variable by iterating across each constraint. We use data parallelism to calculate the new constraints after pivoting to a new vertex in the n-dimensional polytope. Each OpenMP thread (corresponding to each core) is given a subset of the input variables or constraints. This lets each OpenMP thread find the best variable to update or how much the best variable can be updated given the variables and constraints (respectively) in its region.

We initially attempted to use a Python implementation of the Simplex algorithm we found on the Internet. After finding starter code for this algorithm in 15-451 (a class both members of this group are taking), we switched to using this code since it was written in C++, and had been put through the test of many years of students using the code for their 15-451 programming homework.

Some sections, like the feasible check, were relatively simple to parallelize, since they were a series of reductions over the data. However, other sections, such as finding the best vertex, were complex and required modification, since parallelizing them naiively would introduce race conditions. We solved this by restructing and rewriting portions of the code to avoid these race conditions, at a slight synchronization overhead.

When an OpenMP parallel execution context is defined, any private variables need to be copied to each thread before execution can continue. This copying imposes a significant overhead for large data size objects, so our next optimization pass was to reduce the number of shared variables across the program. An example of this is copying a specific row of the input matrix to a vector, to allow only the needed vector to be copied to each thread. This gave a 23.4% speedup, reducing our runtime from 3010ms to 2303ms.

Our next optimization pass was to restructure our code for vectorization and use Clang vectorization hints. This approach helps with regular OpenMP implementations as well, since OpenMP will utilize vectorization to speed up its individual threads. This was a bit of a learning curve, since we needed to learn about

the scenarios that Clang recognizes for SIMD, as well as some bugs in Clang to avoid. There were quite a few bugs with Clang thinking behavior would be modified by adding vectorization, since it didn't realize we would skip over the problematic section. This might be a limitation of Clang's ability to simulate some execution of the code, or it may cost too much computationally to fully analyze the loop. Either way, we were able to circumvent this by forcing Clang to assume loop vectorization safety.

In order to get Clang to vectorize a portion of the code, we need it to recognize a pattern. To help with this, we enabled compiler remarks regarding vectorization and vectorization hints, and looked at the Clang source code unit tests, to see the correct and incorrect ways to structure our code. The remarks would print out, for every hinted loop, the vectorization parameters, or the reason why it wasn't vectorized. For example, "loop not vectorized: cannot identify array bounds" when the compiler was unable to backtrack and figure out the limitations of the array. This message would also indicate the loop index bounds couldn't be determined at compile time, for example in a code snippet of "A[B[i]]".

Another common message was "value that could not be identified as reduction is used outside the loop," this is for the case of when the same value was modified or checked every loop execution, and used outside of the loop. An example of this is finding the max in an array. There were a few cases where Clang would realize this was a reduction, for example, with just finding the max or finding the sum, but for quite a few of our loops, we were taking argmax, something that Clang wouldn't realize as a reduction. We didn't have time for this, but vectorizing argmax would be a good expansion to Clang.

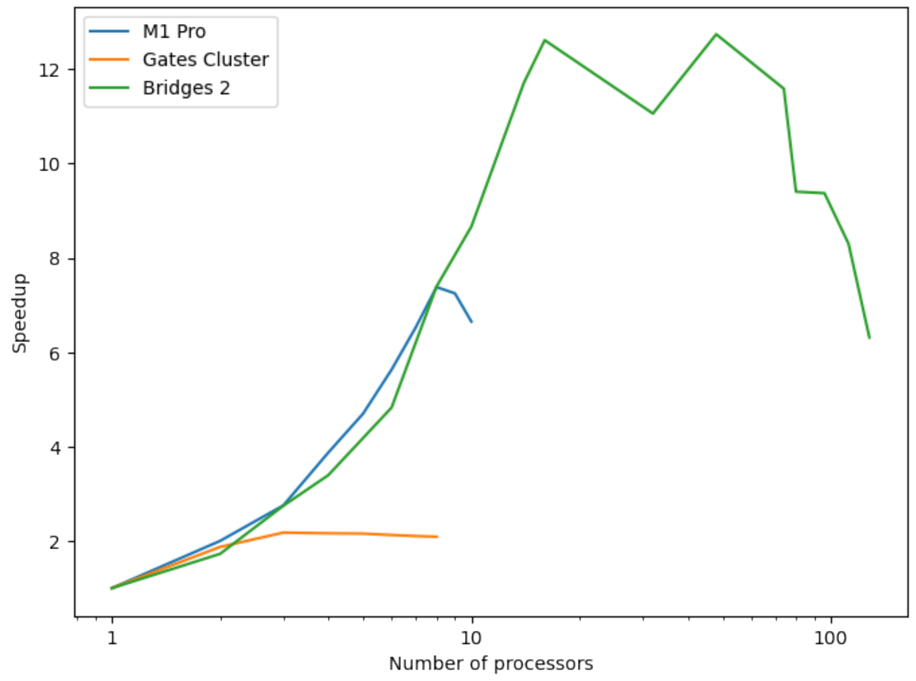
When a loop is vectorized, we would see a message like: "vectorized loop (vectorization width: 2)", meaning that a SIMD width of 2 was used to vectorize the loop. Replacing some of our existing OpenMP routines with vectorization took our runtime down to 2063ms for 20k input size, a 10.4% improvement over our previous implementation of 2303ms.

Our next pass was to use more Clang hints, but for partial unrolling of some loops. Partially unrolling a loop has the goal of reducing time used when processing the loop control variables, such as the counter and checking the loop condition. Adding unroll pragmas to some non-vectorizable loops didn't yield any performance benefit, so we removed the unroll pragmas. This is possibly due to speculative execution having a very high hit rate for a loop, since the loop condition only fails once. Having such a high hit rate allows for the processor to continue partial execution before the loop condition is entirely checked.

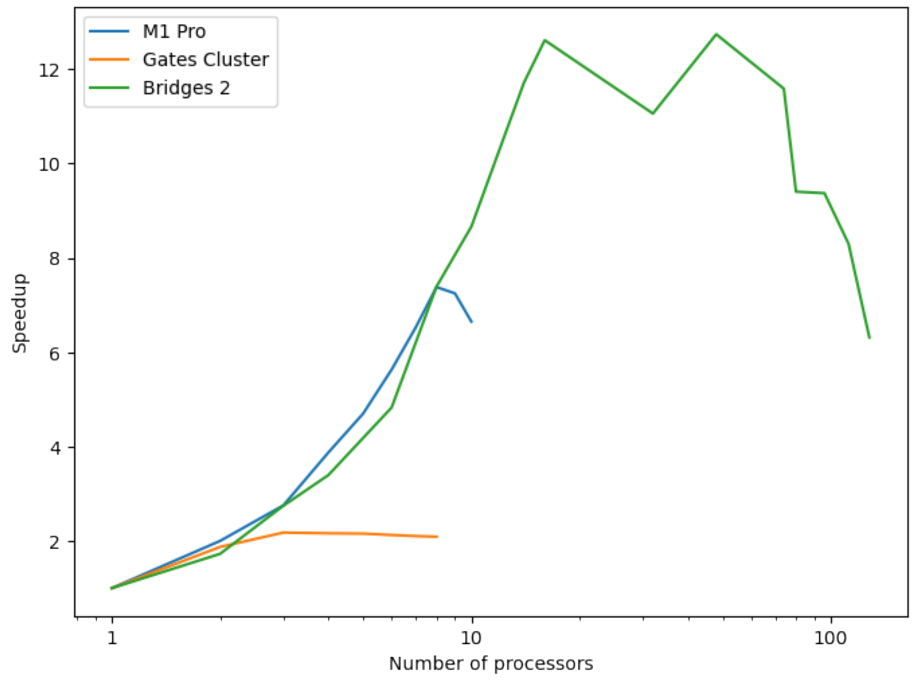
Results:

We measured performance in terms of speedup from our parallel implementation with one core.

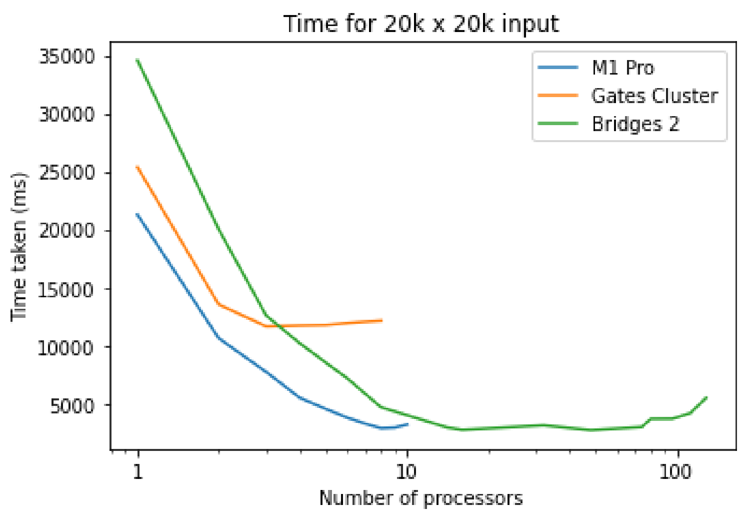
Number of Threads used vs Speedup Compared to Single Threaded Non-Vectorized OpenMP Implementation:



Number of Threads used vs Speedup Compared to Single Threaded Vectorized OpenMP Implementation:



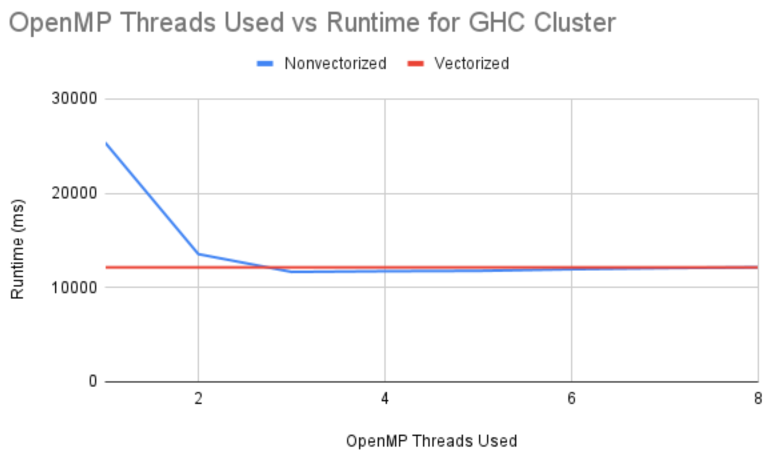
Time used to run the program for each computer:

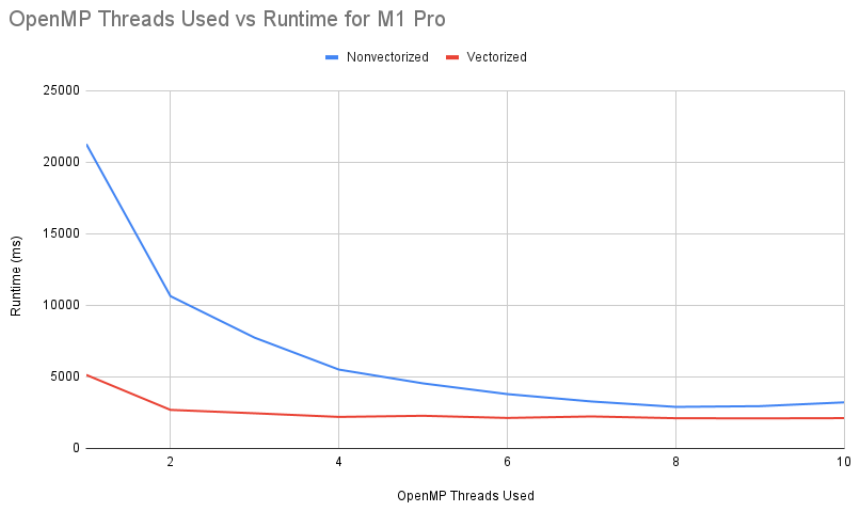


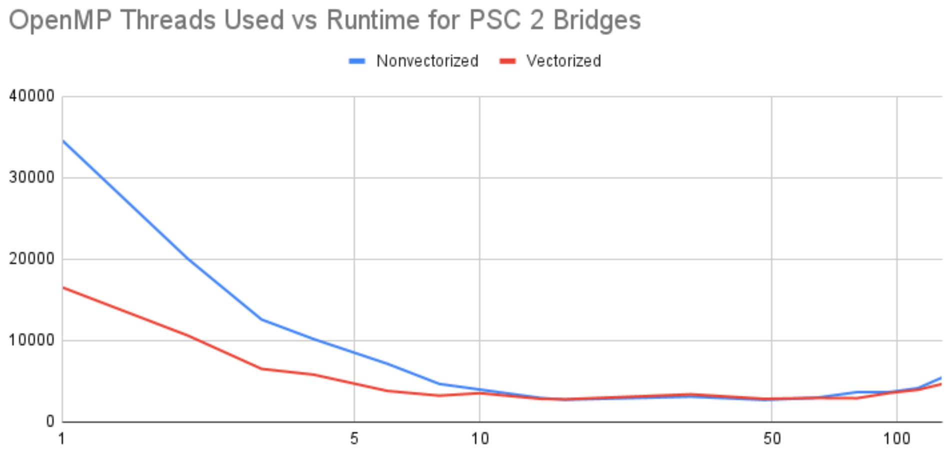
We got these graphs using 20,000 constraints and 20,000 variables randomly generated with our experimental setup.

Our initial experimental setup involved reading sample linear programming problems we found online when making our project proposal. Since many of these linear programs are not in standard form and our baseline code only solves linear programs in standard form, we wrote a parser that converts arbitrary inputs into standard form. This involved a few operations on the inputs to ensure all variables are ≥ 0 and requiring all constraints are of the less than or equal to form. Additionally, our goal function has been changed to a maximum goal. Our parser for input is written in Python and saves the standard form to a text file that is piped to our C++ solver as input.We wrote a basic parser and converter in Python. However, due to the large size of some inputs (with uncompressed versions reaching the hundreds of megabytes), we found our old setup to be inefficient on 11/29. Later that week, we switched to generating random test cases with 20,000 rules and 20,000 constraints in C++. This made testing our Simplex algorithm implementations faster.

Number of Threads used vs Runtime for our devices:

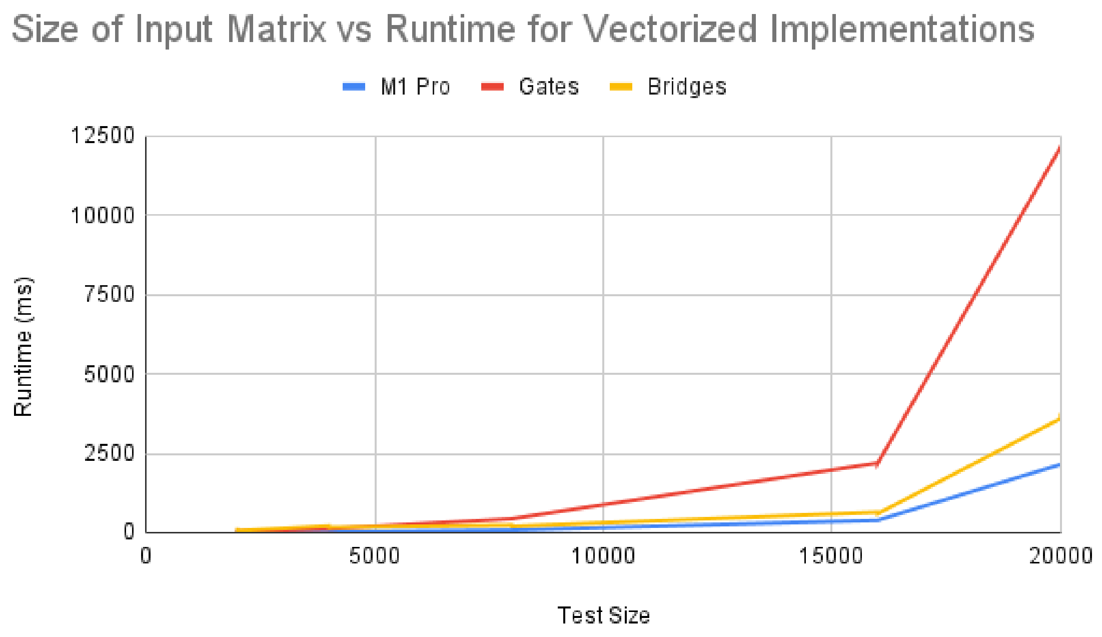






We got these graphs using 20,000 constraints and 20,000 variables randomly generated with our experimental setup.

Input Size vs Performance for our devices:

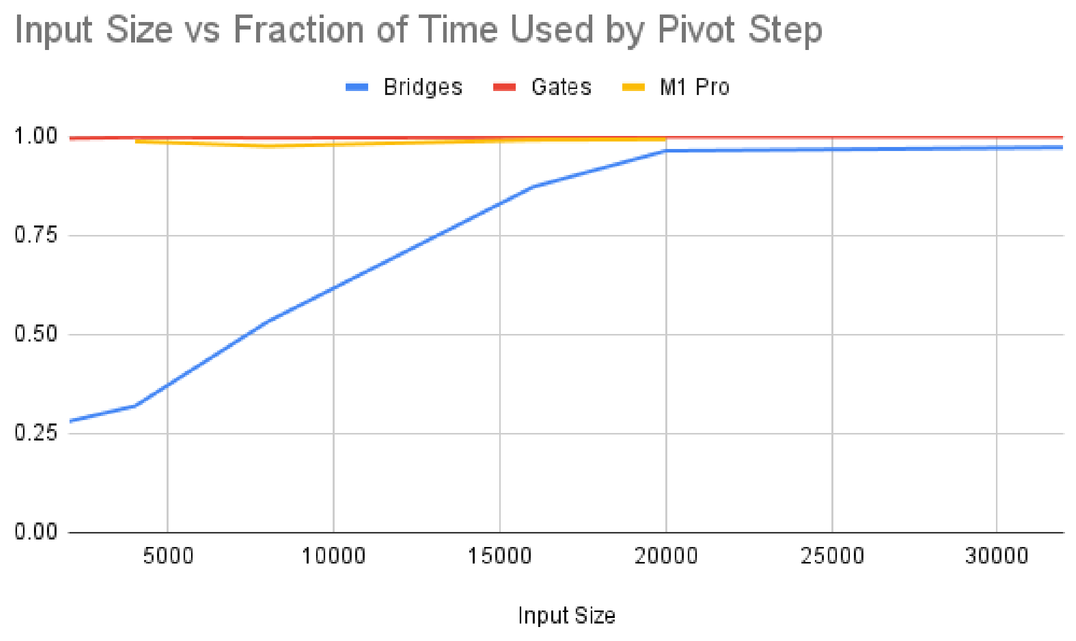


We got these graphs using 10 threads for M1 Macbook Pro, 8 threads for the Gates cluster, and 64 threads on the PSC system.

It is important to report results for different problem sizes. Different problem sizes have different execution behavior on different platforms. The runtime worsens significantly after 16,000 elements. This is most noticeable on the Gates platform. Different workloads don't exhibit different execution behavior. We did not make versions of our code that support a sparse matrix representation to speed up sparser input matrices.

Algorithm's time per region of code: For the M1 Pro and the GHC cluster, most of the time our algorithm takes is from pivoting to a new point on the n-dimensional polytope and updating the constraints and objective function to represent the new pivot. For the PSC-2 Bridges computer, the percent of time taken by pivoting increases as the input size increases. Time taken to find whether an input is feasible and find the constraint to optimize takes the remaining time.

Graphs per time in pivot step code for M1 Pro, GHC Cluster, and PSC-2 Bridges:



We were only able to run perf, a command that we used in lab 3 to give us information about cache misses, on GHC. As we increased the number of threads from 4 to 5 to 6 to 7 to 8, perent of cache hits for the 16k by 16k input matrix increased. The percentages for cache hits were $56.5, 58.5, 86.5, 87.5, 83.7$ percent. As we increased the number of threads from 4 to 5 to 6 to 7 to 8, perent of cache hits for the 16k by 16k input matrix increased. The percentages for cache hits were $56.5, 58.5, 86.5, 87.5, 83.7$ percent. As we increased the number of threads from 4 to 5 to 6 to 7 to 8, perent of cache hits for the 16k by 16k input matrix increased. The percentages for cache hits were $52.7, 53.9, 57.2, 57.3, 56.8$ percent.

We initially suspected that on the Gates machine, false sharing caused performance to decrease as the number of threads increased. This does not seem to be the case since cache hits increased in the 16k case as the number of threads increased.

Speedup analysis:

We conclude that the speedup limits were largely memory throughput limited due to performance across the different devices we tested our code on. As stated earlier, the Simplex algorithm has multiple memory-intensive operations, without much computational intensity. Changing the constraint array when pivoting to a new vertex in the n-dimensional polytope involves editing the entire constraint array. As seen above, the region of code for pivoting between different vertices in the n-dimensional polytope takes almost the entire runtime. Room to improve consists of improving time taken to edit the constraint array during each pivot.

Our fist inclination towards thinking of memory throughput limitations was the severe difference in performance between the M1 Pro and Gates, and even the Bridges computers. The M1 Pro has a higher memory bandwidth than the Gates computer, with the Gates i7-9700 processor having a maximum throughput of about 41GB/s, and the M1 Pro of about 200GB/s. We suspected that given the frequent need to load data from memory to each core's cache with OpenMP during the pivot step, along with the relatively low computational intensity, a faster memory bandwidth for the M1 Pro would make this step faster. This is the case, as the M1 Pro and the Gates machine spend the same fraction of time on this memory-intensive task but the M1 Pro is faster.

Each iteration of the simplex algorithm depends on the new vertex in the n-dimensional polytope found in the previous iteration. We couldn't parallelize across iterations of the Simplex algorithm due to this dependency. We have synchronization issues with finding the best neigbhoring vertex since if we split the neighboring vertices up across the cores, we need to synchronize to find the best vertex to visit. Synchronization costs also are present when finding how much we can change the variable that leads us to the best neigbhoring vertex, as we need to synchronize all OpenMP threads to find the constraint that minimizes this variable the most.

Finding the next variable to move requires taking an argmax an entire column of an array, and this is difficult or impossible to vectorize, since the elements we're checking aren't in contiguous memory locations. Thus, we had to settle for parallelizing this portion with OpenMP reductions.

Data transfer between cores is an issue that limits speedup. Each uses the entirety of A (our constraint matrix) every iteration of the algorithm, which limits the ability of temporal caching to increase performance. There is a slight help with spatial locality of accessing contiguous memory locations, but even this is hampered by having to perform several operations over the columns of a row-major matrix. The input for much of our testing was in size 20,000 by 20,000 doubles, making it about 3.2 Gigabytes. This is much too large for any CPU cache to temporally speed up.

Our choice of target machine was sound. CPUs have a larger L2 cache than GPUs. For example, the i7 9700s on the GHC computers have a 12MB L2 cache and the M1 Pro has a 28MB L2 cache. In contrast, the RTX 2080 on the GHC computers have a 4MB L2 cache. There are 2944 CUDA cores. Given the small size of the L2 cache, a GPU would have more accesses to slower memory. Therefore, using a CPU was the right choice.

References:

- A 15-451 lecture that introduced linear programming. Both members of this group are currently taking 15-451. https://www.cs.cmu.edu/ 15451-f22/lectures/lec15-lp1.pdf
- A paper by one of the creators of HiGHS, a high performance linear programming solver library titled "Towards a practical parallelisation of the simplex method": https://link.springer.com/content/p 008-0080-5.pdf on optimizing simplex
- An online seminar by Gurobi: https://www.gurobi.com/events/how-to-exploit-parallelism-in-linear-and-mixed-integer-programming/
- A set of linear programming test cases at https://netlib.org/lp/data/index.html . We plan on using this if possible to avoid having to write our own test case generator.

Work Distribution:

- Write Proposal: Charlie, Raghav (equal)

- Set up repo & sequential algorithm: Charlie, Raghav (equal)
- Benchmark Sequential algorithm: Charlie, Raghav (equal)
- Write Midpoint: Charlie, Raghav (equal)
- MPS Parser: Charlie, Raghav (equal)
- Simple OpenMP parallelism: Raghav (minimal work)
- Advanced OpenMP parallelism: Charlie, Raghav (equal)
- Optimize OpenMP Memory Sharing & Usage: Charlie
- Clang Vectorization Pragmas: Charlie
- Benchmark OpenMP Bottlenecks: Raghav, Charlie (equal)
- Investigate Parallelism Bottlenecks: Raghav, Charlie (equal)
- Write Final: Charlie, Raghav (mostly Raghav)
- Write Poster: Charlie, Raghav (mostly Raghav)

We request that the grade be split 50/50.