Parallel Linear Programming Solver Using MPI

Yuan (Alex) Guan, Yixuan (Ivan) Zheng

Abstract

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.

Background

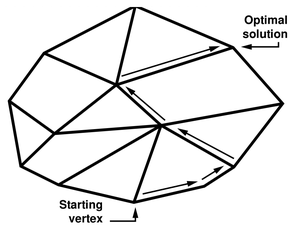
In standard form, an LP can be written as:

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

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

$$x \geq 0$$

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 until we reach a position without a better neighbor.

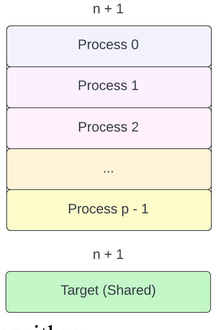


Methodology

Path Parallel: 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.

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

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 process is responsible for updating their portion of the tableau only. All communications are via MPI collective functions.



Major functions of parallel simplex algorithm:
- Eliminate: Modify the tableau with updated value of pivot vertex
- Feasible: Check if LP has valid solution with a call to Eliminate
- Find_Pivot: Find next vertex in the polytope to call Eliminate on

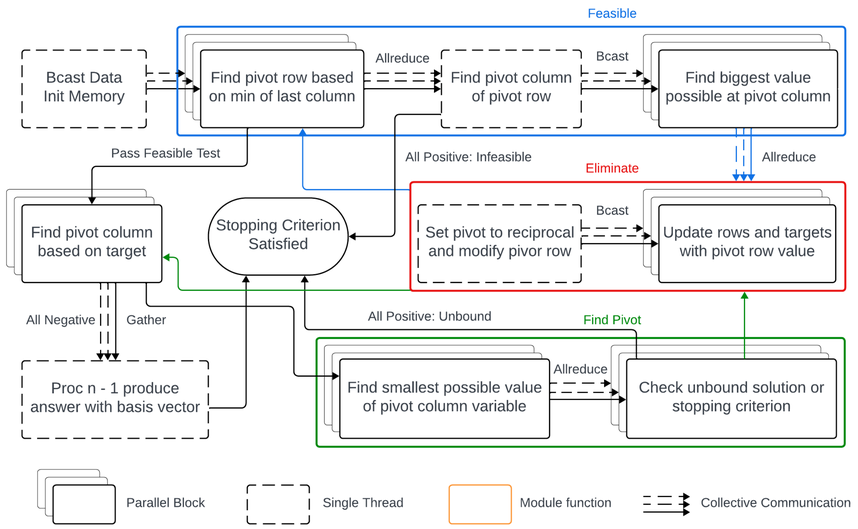


Figure 3: Flowchart of modified simplex algorithm

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

6 major optimizations and how they reduce communication / memory latency:
- Opt1: Store an asymptotically smaller tableau by maintaining a basis vector
  - Reduce memory requirement of each thread by 90% test cases
- Opt2: Store 2D tableau array contiguously by 1 data and 1 interface array
  - Decrease number of MPI calls required; decrease cache miss by 10%

- Opt3: Apply granularity for more async MPI calls with less traffic per call
  - Pipeline parallel messaging passing with computation associated
- Opt4: Store an identical copy of Target array every process
  - Reduce MPI calls of Find_Pivot by 66% with linear memory overhead
- Opt5: Column-wise data partition of tableau
  - Our final row-wise partition approach makes less and efficient MPI calls
- Opt6: Pre-allocate a buffer for sending and receiving message
  - Avoids extra memory allocation per Eliminate call; reduce cache misses

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

Experiments

We measured performance in terms of wall-clock time and speedup compared to running the parallel version on one single process. 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).

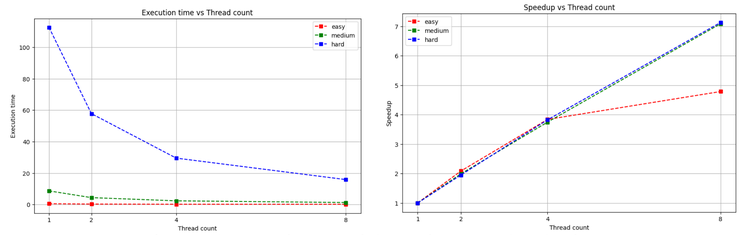


Figure 4: Runtime and Speed up on GHC clusters

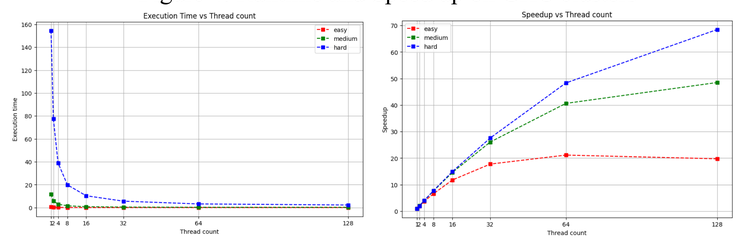


Figure 5: Runtime and Speed up on PSC

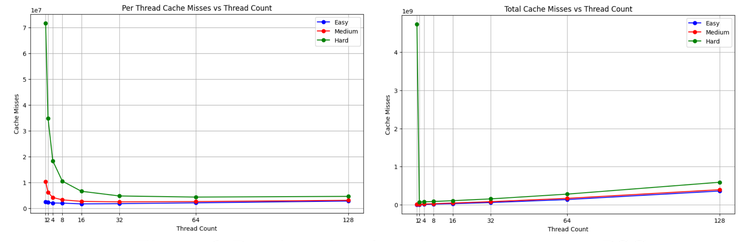


Figure 6: Per thread and total cache misses on PSC

- Fig 7 demonstrate the effectiveness of our optimizations in reducing communication overhead. 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.
- Our optimized code reduces comm-computation ratio by over 2, performing 1 MPI call per 1000 operations. Speedup still gets limited as comm time occupies 70% total runtime on 128 threads.

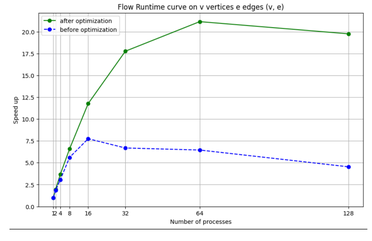


Fig 7: Scaling before / after optim

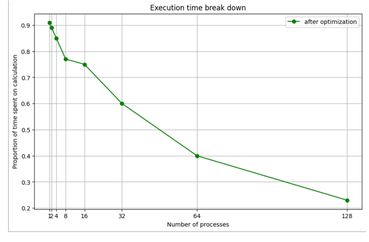


Fig 8: Runtime break down