An empirical investigation of OpenMP based implementation of Simplex Algorithm

Arkaprabha Banerjee[1], Pratvi Shah[1], Shivani Nandani[1], Shantanu Tyagi[1], Sidharth Kumar[2], and Bhaskar Chaudhury[1]

[1] Group in Computational Science & High Performance Computing, DA-IICT, India
[2] University of Alabama at Birmingham, USA

Abstract. This paper presents a shared-memory based parallel implementation of the standard simplex algorithm. The simplex algorithm is a popular technique for linear programming used to solve minimization and maximization problems that are subject to linear constraints. The simplex algorithm reduces the optimization problem to a series of iterative matrix operations. In this paper we perform an empirical analysis of our algorithm and also study the impact of the density of the underlying matrix on the overall performance. We observed a maximum speedup of 10.2 at 16 threads and also demonstrated that our proposed parallel algorithm scales well over a range of matrix densities. We also make an important observation that the effect of increasing the number of constraints is more significant than the effect of varying the number of variables.

Keywords: large-scale problems· linear programming· Simplex · parallel computing· scalable algorithms· OpenMP

1 Introduction

Linear programming, also known as linear optimization, is a method to achieve the optimal outcome for a minimization or maximization problem, subject to a set of linear relationships. Among the various methods available for solving linear programming problems, simplex is the most widely used algorithm, both commercially and academically [1]. The storage and computational overhead make the standard simplex method an expensive approach for solving large linear programming problems. Apart from the computational cost, the standard simplex algorithm also requires the previous iteration to be completed before the new solution can be computed, thus restricting the scope of parallelization [2].

Even though the simplex algorithm is primarily sequential, several attempts have been made during the last decades to parallelize it. There are broadly two implementations for the Simplex Algorithm: The Standard Simplex Method and The Revised Simplex Method (RSM). The Standard Simplex method refers to the original algorithm proposed by George Dantzig [1]. RSM, on the other hand, seeks to efficiently implement the Standard Simplex algorithm by employing a host of Matrix operations specifically built to exploit the sparsity of matrices [3,4]. This means that the RSM is mathematically equivalent to the standard simplex method but differs in implementation. In RSM, instead of having to compute and store the full table in each iteration, it is only necessary to keep track of some of the information, reducing the redundancy, and use the matrix operations directly on the relevant data. This matrix-oriented approach allows for greater computational efficiency, as it exploits, the sparsity of the matrix using matrix inversion techniques optimized for sparse matrices. Apart from certain GPU-based implementations, RSM's optimizations strategies fall short for denser matrices. Furthermore, the steps employed have a limited scope for parallelization [1].

As we moved to the multi processor era, the importance shifted to running the standard algorithm on multiple cores. In 2000, Maros and Mitra presented a cooperative parallel version of the sparse implementation of the revised simplex method for linear programs on distributed memory multiprocessors [5]. Ploskas presented a parallel implementation of the standard simplex algorithm using a personal computer with two cores. Due to dense matrices and heavy communication, the ratio of computation to communication is extremely low and Ploskas' computational results show that a linear speedup is hard to achieve even with carefully selected communication optimization [3,5]. Later many other optimization strategies were proposed [6] such as using certain linear algebraic techniques necessary to exploit the dual block-angular structure of the problem or parallelizing the matrix inverse step based upon GPU implementations. Most of these improvements were primarily done on the revised simplex algorithm [5]. A parallel implementation based on combination of CPU and GPU was also proposed in 2016 [7].

In this paper, we analyze the performance gains of an efficient parallel implementation of the Standard Simplex Algorithm over the sequential version, using a shared-memory architecture. The standard simplex version has been chosen over the revised method owing to known issues of scalability for denser matrices. Here, we define the density of a matrix as the ratio of non-zero elements to the total elements in the matrix. Our study revolves around the following parameters which will help in effective understanding of the parallel implementation of the Standard Simplex Algorithm proposed in this paper:

- Explore the scalability of the algorithm over a range of densities
- Explore the effect on varying the number of constraints
- Explore the effect on varying the number of variables
- Effectively exploiting the SIMD units to update the matrix.

We have analyzed the most important aspects of the algorithm based on the above parameters and the underlying hardware architecture, which to the best of our knowledge, have not been explored and reported in the existing literature. The state-of-the-art implementation does not talk about the performance of the algorithm on varying the density of the matrices. The code has been run on shared-memory architecture systems. All processors share a single view of data and the communication between them can be as fast as memory accesses to a particular location with a lot of intra-node parallelisms to exploit and our implementation is designed specifically for this purpose.

2 Serial Algorithm

Any linear programming (LP) problem can be modelled into the following standard form:

$$\text{maximize } \mathbf{Z} = \mathbf{CX}$$
$$\text{subject to } \mathbf{AX} = \mathbf{B} \qquad \text{where } X \geq 0$$

where A is the constraint matrix, B forms the constant vector matrix and C corresponds to the objective function coefficients. The objective function is the function whose value is to be either minimized or maximized subject to the given set of constraints given by Z. The X vector is the required solution to the LP problem. After the initial modifications, the problems are formulated in the above representation for solving via the Simplex Algorithm.

The simplex method is an iterative procedure for getting the most feasible solution. In this method, we keep transforming the value of basic variables to get the maximum value for the objective function. Within the current context the following assumptions have been made for the linear inequality problems:

– All problems are maximization problems. In the event of a minimization problem, the objective function is multiplied by -1.
– All problems are to be initially considered in the form of less than or equal ($\leq$) inequalities.

$$AX \leq B$$

The following steps illustrate the working of the serial simplex algorithm.

1. Introduce Slack Variables to convert inequality to equality constraints ($AX = B$). The slack variables are known as basic variables and the original ones as non-basic. All slack variables have a zero coefficient in the objective function.
2. Create an initial table[3] consisting of n non-basic variables and m-basic variables. The table consists of the coefficient of the linear constraint variables and the coefficients of the objective function. The slack variables form the initial basis.

		C_j	C_1	C_2	$\cdots$	0	0			
C_B	X_B	B	x_1	x_2	$\cdots$	x_{n+1}	x_{n+2}	$\cdots$	Min Ratio	Row Operations
C_{n+1}	X_{n+1}	B_1	A_{11}	A_{12}	$\cdots$	1	0	$\cdots$		
C_{n+2}	X_{n+2}	B_2	A_{21}	A_{22}	$\cdots$	0	1	$\cdots$		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		
$Z_j - C_j$		ans	$-C_1$	$-C_2$	$\cdots$	$\cdots$	0	0		

[3] **Note:** The table mentioned in Step 2 is not in the block-structured notation. The sample table has been employed in order to effectively explain the serial algorithm.

3. The value of the objective function with respect to every variable (Z_j) at that instant can be calculated by summing up the product of the objective function coefficients of the variable in the basis and the coefficient associated to other variables in the same row of the table.
4. Calculate $Z_j - C_j$ for all the variables. C_j represents the coefficient of the variable in contention in the objective function. The column with the maximum value (< 0) represents the entering variable for the basis in the next iteration. This column is known as the pivot column. If all values of reduced cost (Z_j - C_j) are ≥ 0 then the optimal solution has been reached.
5. Calculate the ratio of the elements of B with the corresponding coefficients of the pivot column. The row representing the minimum positive value of the ratio represents the variable that will leave the basis in the next iteration. This row is known as the pivot row. If all the values of the replacement ratio are either negative or infinite, then it represents a case of unbounded solution.
6. The intersection value of the Pivot row and Pivot column gives the value of the pivot coefficient. Divide the pivot row with the pivot coefficient. Subtract all the other rows from the new modified pivot row by a multiplier such that all the other values in the pivot column apart from the pivot coefficient become zero. In order to prepare for the next iteration, swap the entering and leaving variables along with all the other associated values.
7. Go to step 3. Repeat until the algorithm ends.

An example of the above-mentioned algorithm can be seen in Appendix A.

3 Parallel Algorithm

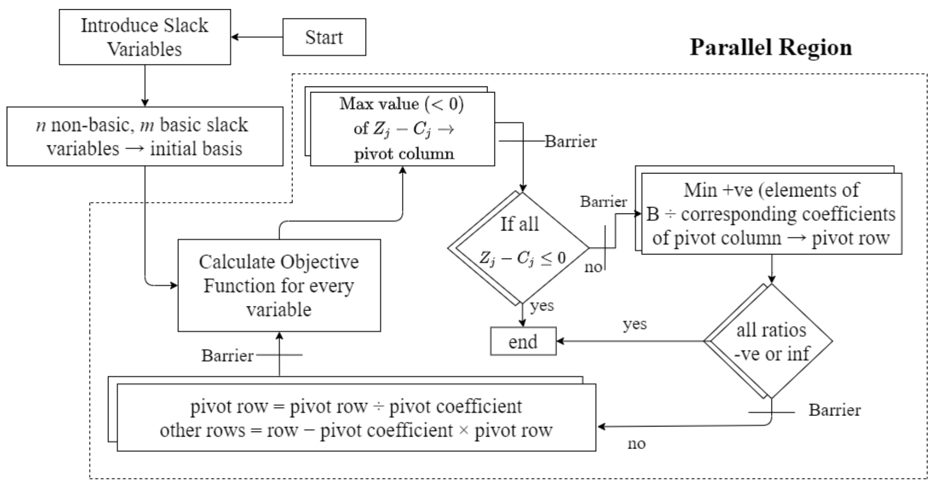

Start
Introduce Slack Variables
n non-basic, *m* basic slack variables → initial basis
Parallel Region
Max value (< 0) of $Z_j - C_j$ → pivot column
Barrier
If all $Z_j - C_j \leq 0$
no
yes
end
Min +ve (elements of B ÷ corresponding coefficients of pivot column → pivot row
all ratios -ve or inf
Calculate Objective Function for every variable
pivot row = pivot row ÷ pivot coefficient
other rows = row − pivot coefficient × pivot row

Fig. 1. Flowchart for the parallel implementation. Nodes with an overlying boundary in the background represent regions where multiple threads work concurrently.

Researchers till now have stated that the serial time complexity of the simplex algorithm is generally a polynomial, but for the worst case, the time complexity tends to increase exponentially [6]. The execution time depends on two factors: time taken for each iteration and secondly, the number of iterations which will be equal to the number of pivots that need to be traversed in order to reach the optimal point in the n-dimensional space which satisfies the given constraints [8].The latter factor makes the evaluation of time complexity a highly involved task, as every problem depending on the density and the structure of the constraint matrix gives rise to a novel situation.

Efforts to effectively modify the algorithm, by evaluating the different cases, are being carried out in order to generalize the time complexity expression [6]. From a generic standpoint, one can assume that the time complexity increases with the number of equations or the number of variables, however, this is not guaranteed. For certain cases, it tends to the worst exponential time complexity, leading to an exceptionally large number of iterations to obtain a solution, even for relatively smaller problem sizes [9].

In order to understand how the serial algorithm can be parallelized, the time distribution among various steps of the naive serial version (Section 2) is analyzed. The results show that Step 6 constitutes the major portion of the run time with more than 99.5% contribution followed by Steps 3 and 4 (or Step 7), with Step 5 taking the least time.

3.1 Implementation

In this Section we provide a methodology for parallel implementation using OpenMP which optimizes its serial counterpart mentioned in Section 2. Fig. 1, gives an overview of the steps involved in the implementation.

Steps 1 and 2 are the same as the serial implementation of the standard simplex algorithm given in Section 2, since it is essential to perform these steps in a sequential format before solving the linear programming problem. After these steps, we create a parallel region and define the required number of threads and move to OpenMP implementation.

Step 3 and 4 now uses all the threads that were initialized. Every thread works among a defined set of columns to find the index of the column holding the maximum absolute value among the negative elements in the objective row. A *user-defined class* consisting of the values of the maximum data entry and the corresponding index are stored and evaluated via the *reduction* clause. This class notation and reduction clause together form a powerful tool to evaluate maximum or minimum elements in a data structure, as compared to traditional explicit comparison mechanisms.

For Step 5, we again make use of the *reduction* clause with the user-defined class to find the leaving variable by evaluating the Minimum Ratio row-wise. Finally, a single thread with a *nowait* clause checks if the solution is unbounded or not present, and exit the parallel region if the solution satisfies the unbounded constraints.

In Step 6, the pivot row is now updated concurrently among the threads. Once the pivot row has been updated, all the remaining rows are updated. The rows are evaluated via the *pragma for* construct, while the iterations among columns has also been vectorized via the *simd* construct. Simultaneous modifications are possible because there are no dependencies among columns or rows.

Steps 3 and 4 (as Step 7), are again performed in the parallel region with the mechanism mentioned above. A single thread finally updates all iteration-specific local variables. Finally, the loop continues until a solution reached, or an exit clause is triggered for 'Unbounded Solution/No Possible solution'.

The above steps are implemented using a block-structured matrix notation having dimensions $(m+1) \times (n+m+1)$ (where m is the number of constraints and n, the number of variables) as proposed in the case of the Standard Simplex method of Dantzig [1]. The same is given as $\begin{bmatrix} A & B \\ -C & 0 \end{bmatrix}$ where A is an mxn matrix, B is an mx1 vector, and C is an 1xn matrix.

Algorithm 1 Parallel Implementation of Simplex Algorithm

```
//Get the dimension of the table
Rows(R)= m + 1,
Columns(C) = m + n + 1
 Initialize  & load the data in table
Algorithm time starts from here

# pragma omp parallel <shared variables>
{
//Step 3 & 4
# pragma omp for <schedule> <reduction>
for(int j=0;j<C;j++){
    find max negative value (max_value)
    from reduced cost row
}

Corresponding index=max_index
& column=Pivot Col

#pragma omp single nowait
max_value=0

do{
     // Step 5
     #pragma omp for <schedule><reduction>
     for(int j=0;j<R;j++) {
          find the min value
          (min_value) from Min Ratio column
          do count++ for negative values
     }

     #pragma omp single nowait
     {
          if count == R
               there is unbounded/no solution
               flag=False break
     else
          Row corresponding to
          min_value is the
          Pivot row with index= min_index
     }

     // Step 6
     pivot = table[min_index][max_index]
     #pragma omp barrier

     #pragma omp parallel for
     for(int i=0;i<C;i++)
          update Pivot Row
     #pragma omp parallel for
     for(int i=0;i<C;i++)
          #pragma omp simd
          for(int i=0;i<R;i++)
                update all elements except
                that of Pivot row
     }

     //Step 3 & 4 repeated
     #pragma omp for <schedule> <reduction>
          for(int i=0;i<C;i++)
             find the new reduced cost values
             for updated table
             countNegative++ for negative values

     #pragma omp single
            Update the initial conditions
     }while(countNegative and flag)
}

//Algorithm time ends here
Solution = table[Rows][Columns]
```

3.2 Optimization Strategies

After analyzing the time bifurcation and identifying the steps which take significant amount of time we conducted experiments to optimize the code. The following methodologies were explored to make the algorithm more efficient.

Optimal Scheduling clause and Load Balancing: As shown in Fig 2, for larger problem sizes, the performance for static scheduling clauses is possibly hampered when some threads take more time to complete their share of work. Even for dynamic scheduling, with a completely random thread allocation, enhanced performance may require guided scheduling. Thus, guided scheduling mechanisms were used to effectively tackle the load balancing problem.

Optimal SIMD Units: In order to find the SIMD units for vectorization in Step 6 above, multiple values of SIMD units ranging from 2 to 8 were considered. The optimal SIMD length was found to be 4 for our implementation, just half of the total number of lanes. We assume that the use of all SIMD lanes generates excessive overhead for using additional SIMD units, and the use of fewer than half the SIMD lanes under-utilizes a SIMD unit's resources.

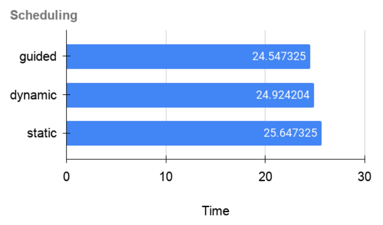


Fig. 2. Scheduling on 512x512 dataset of 0.5 density

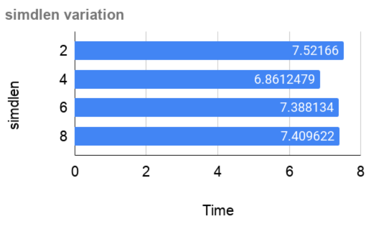


Fig. 3. Variation of simdlen()

3.3 Algorithm Analysis

In this section the algorithm is analyzed to provide a basis for drawing out conclusions.

Cache Miss Analysis Assume that the number of rows is 'r' and the number of columns is 'c', while the matrix is stored in row-major format in the memory. Theoretical analysis of each step led to the following expression: The steps mentioned below are the corresponding steps in the Serial Algorithm mentioned in Section 2.

Steps 3 and 4 access the matrix row wise (considering a cache line of 64 bytes and double data-type): c/8 misses

Step 5 requires us to access the matrix column-wise: r misses

Step 6 update almost the entire matrix : $\frac{r*c}{8}$. Thus Miss ratio in a single iteration of the overall loop would be

$$MissRatio = \frac{\frac{c}{8} + r + \frac{r*c}{8}}{r + c + r*c} \approx \frac{1}{8}$$

In order to verify the above miss ratio, profiling was performed via **Valgrind** using the memcheck and callgrind tools on BENCH2 for a 256x256 problem of density 0.5. The cache simulator simulates a computer with a split L1 cache (separate instruction I1 and data D1), which is backed up by a single second-level cache (L2). This is consistent with the architecture of most modern machines' caches. The reads/writes and respective misses recorded after profiling for L1 data and L2 unified cache are:

	D refs	D1 misses	LLd misses	D1 miss rate	LL refs	LL misses	LL miss rate
serial	39,028,802,003	66,529,666	28,606	0.1705%	66,547,316	31,020	0.0466%
openMP	44,301,959,277	66,980,319	29,457	0.1512%	66,998,585	32,331	0.0483%

Table 1. D refs (Data cache memory reads), D1 misses (D1 cache data misses but found in L2), LLd misses (L2 cache data misses but found outside it), LL refs (Combined L2 cache references), LL misses (Combined L2 cache misses)

The key observations to be made here is that that the D1 cache miss rate has gone down in the parallel (OpenMP) version as compared to the serial version. Furthermore, the cache miss rate is significantly less than what is expected theoretically. This can be explained by the fact that instead of bringing in cache blocks one by one, the compiler automatically optimizes this procedure based on the repetitive access patterns that it finds with every iteration. One such compiler optimization is pre-fetching. Pre-fetches are possible only if the memory addresses can be determined ahead of time. However, for extremely small table sizes, the cache miss rate is much higher on account of ineffective compiler optimizations and follows the standard expected values.

Analyzing the nature of the algorithm The serial implementation of the simplex algorithm, Section 2, was evaluated for understanding the nature of the algorithm, in particular, whether the algorithm is CPU-bound or memory-bound. In our algorithm we consider m as the number of constraints and n as the number of variables. So, the size of the table will be $(m+1) \times (n+m+1)$, and let $a = m+1$ and $b = n+m+1$.

Now, let's consider a single iteration of the ***do*** loop, as this represents the most granular as well as comprehensive segment of this iterative algorithm. An analysis of its steps will provide a basic picture of the algorithmic operation. We obtained the following expression after analyzing the number of computations and memory access counts:

$$\texttt{Computations} = 3a + b + a + 2ab + 6b$$
$$\texttt{Memory Access} = 2a + b + a + 2ab + 2b$$

The above figures will be multiplied by the total number of pivots which is a constant factor, depending upon the problem. So, barring that factor, we can state that our implementation of the simplex algorithm has more computations in comparison to memory access and thus, being CPU-bound it will be more suitable to be parallelized on a shared-memory architecture.

4 Experimental Results and Observations

We have implemented our algorithm on two systems with the following hardware architecture.

Specifications	**BENCH1**	**BENCH2**
Model Name	Intel(R) Xeon(R) Silver 4214R CPU @ 2.40GHz	Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz
Core(s) per socket	12	8
Socket(s)	2	2
L1d cache	768KB	32KB
L2 cache	24MB	0.256MB
L3 cache	33MB	20.48MB
GNU GCC version	9.3.0	10.2.0

The primary motivation for using two different hardware architectures was to understand the performance of this algorithm on different hardware cores with large and small L1 caches and large and small L2 caches per core. The above data represents the total cache of the system in consideration. BENCH1 has 1MB L2 cache per core while BENCH2 has 256KB of L2 cache per core. L3 cache is shared in both cases.

We have compared certain selected results with the current state-of-the-art algorithm implementation which was implemented on a system with four AMD Opteron 6376 processors with 16 cores, totalizing 64 cores, 768KB L1 and 16MB L2 individual caches per core, and 16Mbytes L3 caches per socket, running Ubuntu 16.04.2 LTS [5].

For reproducibility, we have made use of the standard NETLIB LP dataset[4], which comes in the specific mps format, consisting of all the necessary variables and their respective coefficients. In addition to the Netlib dataset, we made use of a computationally generated dataset of specific dimensions and density [10]. The generated datasets do not guarantee a finite solution, and hence, some anomalies might arise in the analysis of those datasets, but they are not relevant to the behavior of the algorithm. The results have been verified using standard reference codes, and the answer is consistent over multiple thread configurations.

Keeping into consideration the configuration of our machines and the fact that in the worst case the Simplex algorithm can take exponential time to solve, we limit our observations to the maximum number of variables to 4096 and the number of constraints as 512 in the primal formulation. We have also analyzed their dual counterpart. For most of the cases we have either fixed variables to 256 and varied the number of constraints or vice versa, as this allowed us to efficiently exploit the different levels of the underlying memory hierarchy of the system.

[4] http://www.netlib.org/

All mean execution times have been measured in seconds. All standard graphs have been plotted with respect to the run time on the BENCH1 unless mentioned otherwise.

4.1 NETLIB Dataset

In this section we evaluated the standard Netlib dataset using our serial and parallel code.

From Fig. 4 we observed that the speedup for all the datasets remained within the linear upperbound and we could also see that for smaller datasets the speedup for large number of threads was very low due to the synchronisation overhead. The decrease in speedup after certain problem size is due to fetching of the data from L3 cache as we have 1MB per core L2 cache and the size of dataset exceeds that limit.

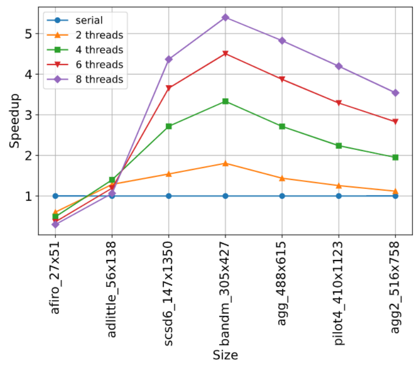


Fig. 4. Speedup for standard netlib datasets (performed on BENCH 1)

4.2 Variation of the Number of Variables

In this section, we examine LP problems of 256 constraints and variables varying from 256 to 4096 with a density of 0.5. We can verify from the hardware architecture that the data with this large problem size would be fetched from the L2 or L3 cache. This lead to an increase in fetching time and overhead incurred due to the necessity to maintain consistent copies of the data across all the processors.

In Fig. 5 (256 constraints) it is observed that the speedup for each thread size increases (up to a point at mid-size) and then decreases. Peak values for large thread counts occur at larger problem sizes. At the largest problem size, the reduction in the peak performance is greatest for small thread counts (1, 2 and 4) and smallest for the largest thread counts (8,12 and 16).

We could observe superlinear speedup in the case of 2 and 4 threads for certain problem sizes. Due to *pragma omp for*, there is coarse parallelization, whereas the use of *simd* enables finer parallelization within each thread and specifying the vector length in *simdlen*() can give us control over the extent of parallelization needed/supported by the system. Thus, the superlinear speedup can be attributed to this increased parallelization within each thread.

From Fig. 6 we see that the speedup achieved in the BENCH2 is similar to the one achieved in the BENCH1, till the problem size fits in the L1 and L2 cache of the respective systems. After that, we witness a drop in the speedup. In the case of BENCH2, when the work allocated per thread (in terms of the size of the table) exceeds the L2 cache size and results in data being continuously fetched from the L3 cache, an increase in the mean execution time occurs. BENCH1 had larger L1 and L2 cache sizes leading to higher speedup for its simulation.

Our implementation observes a maximum speedup of 10.2 with 16 threads for 256x2048 which is comparable to the maximum speedup of ≈ 10 for a problem size of 256x4096 with 16 threads in a state-of-the-art implementation, shown in [5]. Secondly, we see that in the state-of-the-art implementation, although the relative trend is similar, the speedup increases till 16 threads for all problem sizes but in our case, it starts decreasing from 4 to 8 threads for smaller problem sizes owing to the difference in the hardware architecture.

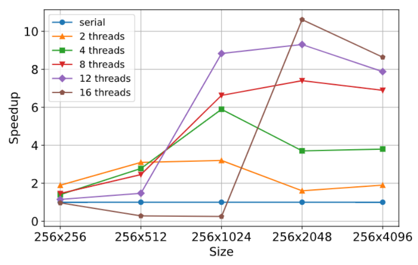


Fig. 5. Speedup (256 constraints)

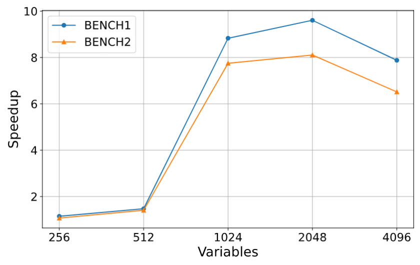


Fig. 6. Speedup (256 constraints and 12 threads)

4.3 Variation of the Number of Constraints

In this section, we examine LP problems for 256 variables and constraints varying from 256 to 4096 with a density of 0.5.

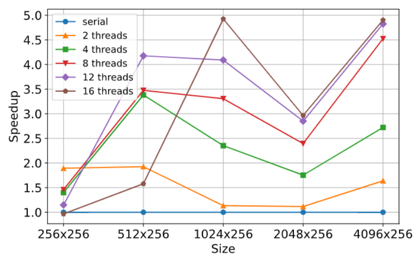


Fig. 7. Speedup (256 variables)

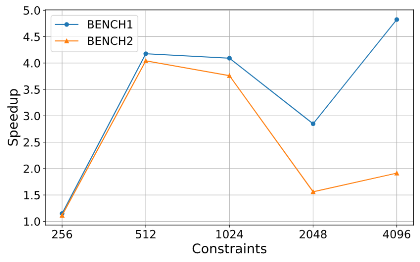


Fig. 8. Speedup (256 variables and 12 threads)

On increasing the constraints for 256 variables, the speedup increases faster (because of the number of iterations increase), as compared to when the number of variables increased. As a result of this, the drop in speedup on increasing threads, which occurred in the previous section for 256 constraints(Fig. 5), now happens at a lower threshold and is evident from Fig. 7 and Fig. 8. For 2048 constraints with 256 variables the problem size is about 36MB which exceeds the L3 cache limit for the BENCH1. Thus we observe a temporary increase for 4096x256 since the problem has no solution and has a very low execution time.

As compared to the state-of-the-art implementation, for problems with 256 variables, we see that in both the implementations, the speedup increases for larger problem size as threads increase to 16 and vice versa is seen for smaller problem sizes where the speedup first increases and then decreases as the threads increase to 16. We see smaller speedup values, in general, as compared to the state-of-the-art implementations. This can be attributed to the smaller cache architecture for L2 and L3 levels.

4.4 Variation in Matrix Density

The standard Simplex Algorithm for OpenMP was initially proposed in [5] primarily for dense matrices. In this section, we attempt to explore its scalability to lower densities. We have considered 512x512 matrices with densities varying from 0.1 to 1 in steps of 0.1. These experiments were performed on BENCH1.

Sparse problems often take fewer number of iterations to be solved, as compared to dense matrices, owing to their inherent matrix structure and the number of manipulations involved. Therefore, the synchronization overhead has a greater precedence, and speedup is reduced. Hence from a generic standpoint, sparse matrices may have a slightly less speedup as compared to dense matrices in this algorithm. The final result however depends on the actual problem structure. We can see from Fig. 9,that the speedup in all the cases remains almost constant or increases a little when the density of the matrices increases. Hence, the parallel algorithm is scalable in that nature.

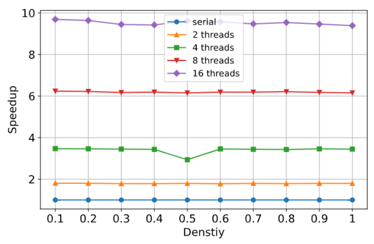


Fig. 9. Speedup vs Matrix Density

4.5 Discussion

As the number of threads increase, the problem partitioning also increases. Since every iteration needs to modify the entire table, using more threads increases the synchronization overhead, while using a lower number of threads reduces the parallelization. We achieve an optimal limit on the number of threads somewhere in between. We could also conclude that there exists a critical problem size for each thread where the nature of the speedup changes from increasing to decreasing on either side of that critical number. This critical value is achieved at a larger problem size when using a larger thread count.

We observed that smaller problems performed better with a lower number of threads as the overhead associated with a larger number of threads significantly increases the run-time. This overhead is mainly attributed to two factors: synchronisation overhead amongst the threads, and false sharing when there are multiple threads working on the same cache line(primarily in step 6). However, on increasing the problem size, the synchronization overhead takes less precedence as the scope for parallelization increases leading to higher speedups.

In general, efficiency decreases with an increase in the number of threads. At large problem sizes with high thread counts, even though the absolute speedup is high, the efficiency is quite low. This can be verified from Fig. 10. For larger problems with higher iteration counts, we need to maintain synchronization even among a single iteration, highlighted by *pragma omp barrier* constructs in the parallel implementation. This is why the synchronization overhead plays a major role.

As compared to the state-of-the-art implementation, for problems with 256 constraints, we see similarity for smaller number of variables where efficiency decreases as threads increases for a given number of variables. The comparison for 2 threads is not valid due to their assumption that the time for serial implementation is double that of using 2 threads while we did not make that assumption.

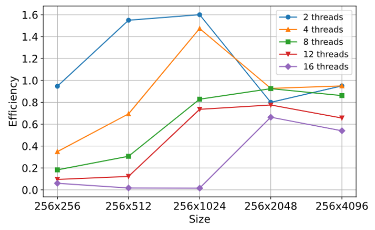


Fig. 10. Efficiency (256 constraints)

5 Conclusion

The theoretical understanding of the standard simplex algorithm supported by the experimental observations from our OpenMP based parallel implementation on two different architectures for a variety of problem sizes enabled us to critically analyze the problem. In our CPU based parallel implementation, vectorization contributes significantly towards improving the performance, however, this is constrained by the hardware properties of the system as well as the problem structure. Our parallel algorithm proved to be fairly scalable, in terms of relative speedup, for the matrices of varying densities (in range of 0.1 to 1).

We could also conclude that the number of constraints has a greater factor of proportionality while determining speedup in comparison to the number of variables. This is because the problem size or the number of computations increases more with the increase in the number of constraints in comparison to the number of variables. This can also be explained using two types of overhead, synchronization overhead and/or the overhead due to completely filled cache leading to delayed memory access. Increasing the number of constraints for the chosen problem sizes lead to cache fulfillment hence the drop in the performance, whereas on increasing variables, the synchronisation overhead incurred due to false sharing, dominates and in a bid to maintain consistent values, in the shared L3 cache and higher memory units , we incurred a drop in the performance. The source code pertaining to this work is being made publicly available under a permissive open source licence at Github https://github.com/arkaprabha10/Simplex-Algorithm.

References

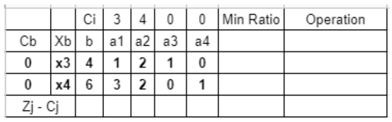
1. George B. Dantzig. 1990. Origins of the simplex method. A history of scientific computing. Association for Computing Machinery, New York, NY, USA, 141–151.DOI: https://doi.org//10.1145/87252.88081
2. Karl Heinz Borgwardt. 1986. A probabilistic analysis of the simplex method. Springer-Verlag, Berlin, Heidelberg.
3. Ploskas N., Samaras N., Margaritis K. (2013) A Parallel Implementation of the Revised Simplex Algorithm Using OpenMP: Some Preliminary Results. In: Migdalas A., Sifaleras A., Georgiadis C., Papathanasiou J., Stiakakis E. (eds) Optimization Theory, Decision Making, and Operations Research Applications. Springer Proceedings in Mathematics & Statistics, vol 31. Springer, New York, NY. DOI:https://doi.org//10.1007/978-1-4614-5134-1_11
4. Harvey M. Wagner. 1957. A Comparison of the Original and Revised Simplex Methods. Oper. Res. 5, 3 (June 1957), 361–369. DOI:https://doi.org//10.1287/opre.5.3.361
5. Coutinho, D., Souza, S.X., & Aloise, D. (2018). A Scalable Shared-Memory Parallel Simplex for Large-Scale Linear Programming. ArXiv, abs/1804.04737 `https://arxiv.org/pdf/1804.04737v1.pdf`
6. Goldfarb D. (1994) On the Complexity of the Simplex Method. In: Gomez S., Hennart JP. (eds) Advances in Optimization and Numerical Analysis. Mathematics and Its Applications, vol 275. Springer, Dordrecht. DOI:https://doi.org//10.1007/978-94-015-8330-5_2
7. Basilis Mamalis and Marios Perlitis. 2016. A Hybrid Parallelization Scheme for Standard Simplex Method based on CPU/GPU Collaboration. In Proceedings of the 20th Pan-Hellenic Conference on Informatics, 2016 (PCI '16). Association for Computing Machinery, New York, NY, USA, Article 12, 1–6. DOI:https://doi.org//10.1145/3003733.3003757
8. John Fearnley and Rahul Savani. 2015. The Complexity of the Simplex Method. In Proceedings of the forty-seventh annual ACM symposium on Theory of Computing (STOC '15). Association for Computing Machinery, New York, NY, USA, 201–208. DOI: https://doi.org//10.1145/2746539.2746558
9. Ed Klotz, Alexandra M. Newman, Practical guidelines for solving difficult linear programs, Surveys in Operations Research and Management Science, Volume 18, Issues 1–2, 2013, Pages 1-17, ISSN 1876-7354, DOI:https://doi.org//10.1016/j.sorms.2012.11.001
10. Ketabchi, S., Moosaei, H., Sahleh, H. and Hedayati, M., 2012. New Methods for Solving Large Scale Linear Programming Problems in the Windows and Linux computer operating systems. DOI: https://doi.org//10.12785/amis/070440

A Appendix: Serial Algorithm - Working Example

This example illustrates the standard simplex algorithm steps mentioned in Section 2. Steps 1 to 7 are the basic steps of the algorithm, whereas steps 8 and onward are for a second iteration.

Suppose, $Z = 3x_1 + 4x_2$
Subject to,
$x_1 + 2x_2 \leq 4$
$3x_1 + 2x_2 \leq 6$
$x_1, x_2 \geq 0$

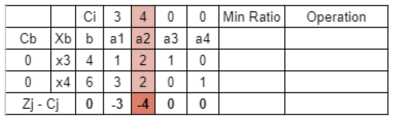
1. Introduce slack variables to get,
   $Z = 3x_1 + 4x_2 + 0x_3 + 0x_4$
   Subject to,
   $x_1 + 2x_2 + x_3 + 0x_4 = 4$
   $3x_1 + 2x_2 + 0x_3 + x_4 = 6$
   $x_1, x_2 \geq 0$
2. Table of coefficients is made with slack variables as basic variables.



		Ci	3	4	0	0	Min Ratio	Operation
Cb	Xb	b	a1	a2	a3	a4		
0	x3	4	1	2	1	0		
0	x4	6	3	2	0	1		
Zj - Cj								

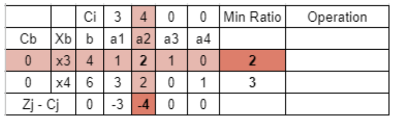
3. The $Z_j - C_j$ differences are evaluated.

4. The smallest value (-4 here) for x_2 is determined. It becomes the new entering variable and the corresponding column becomes the pivot column.



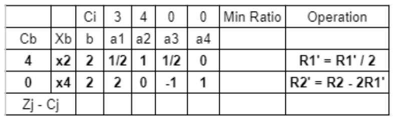
		Ci	3	4	0	0	Min Ratio	Operation
Cb	Xb	b	a1	a2	a3	a4		
0	x3	4	1	2	1	0		
0	x4	6	3	2	0	1		
Zj - Cj		0	-3	-4	0	0		

5. The min ratios are determined and the smallest value (2) is set for x_3 variable, which becomes the leaving variable, and the corresponding row becomes the pivot row.



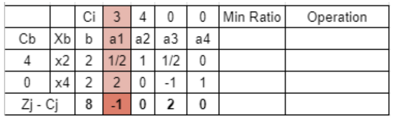
		Ci	3	4	0	0	Min Ratio	Operation
Cb	Xb	b	a1	a2	a3	a4		
0	x3	4	1	2	1	0	2	
0	x4	6	3	2	0	1	3	
Zj - Cj		0	-3	-4	0	0		

6. The pivot row is divided by the pivot coefficient (2).



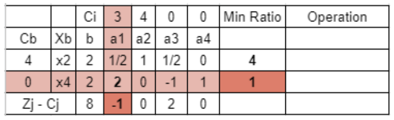
		Ci	3	4	0	0	Min Ratio	Operation
Cb	Xb	b	a1	a2	a3	a4		
4	x2	2	1/2	1	1/2	0		R1' = R1' / 2
0	x4	2	2	0	-1	1		R2' = R2 - 2R1'
Zj - Cj								

7. Now we have the new basis variables as x_2 and x_4. We again evaluate $Z_j - C_j$ values.
8. The smallest value for differences is -1, and is set in x_1, which is the new entering variable, with the corresponding column set as the pivot column.



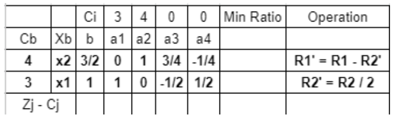
		Ci	3	4	0	0	Min Ratio	Operation
Cb	Xb	b	a1	a2	a3	a4		
4	x2	2	1/2	1	1/2	0		
0	x4	2	2	0	-1	1		
Zj - Cj		8	-1	0	2	0		

9. The min ratios are determined and the smallest value (1) is set in x_4, which becomes the leaving variable, and the corresponding row becomes the pivot row.



		Ci	3	4	0	0	Min Ratio	Operation
Cb	Xb	b	a1	a2	a3	a4		
4	x2	2	1/2	1	1/2	0	4	
0	x4	2	2	0	-1	1	1	
Zj - Cj		8	-1	0	2	0		

10. The pivot row is divided by the pivot coefficient (2).



		Ci	3	4	0	0	Min Ratio	Operation
Cb	Xb	b	a1	a2	a3	a4		
4	x2	3/2	0	1	3/4	-1/4		R1' = R1 - R2'
3	x1	1	1	0	-1/2	1/2		R2' = R2 / 2
Zj - Cj								

11. Now we have the new basis variables as x_2 and x_1. We again evaluate all $Z_j - C_j$ values. All values are ≥ 0 and we terminate the algorithm with: $x_1 = 1$, $x_2 = \frac{3}{2}$, and $Z = 9$