Multicore Scalability and Efficiency Analysis of the standard simplex algorithm

Demétrios A. M. Coutinho
Departamento de Computação e Automação – Universidade Federal do Rio Grande do Norte
Natal, RN – Brazil
demetrios@dca.ufrn.br

Samuel Souza
Departamento de Computação e Automação – Universidade Federal do Rio Grande do Norte
Natal, RN – Brazil
samuel@dca.ufrn.br

ABSTRACT

This paper presents a scalability and efficiency analysis for a parallel implementation of the standard simplex algorithm for multicore processors for solving large-scale linear programming problems. We present a general scheme explaining how we parallelize each step of the standard simplex algorithm pointing out important spots of our parallel implementation. We choose to implement the standard simplex algorithm rather than the revised method for being more suitable for parallelization. We verify the scalability of different amounts of constraints and variables for large-scale problems, finding evidence that the standard simplex algorithm has better parallel efficiency for problems with more variables than constraints. One of the disadvantages of the standard simplex algorithm is being less efficient for problems with more variables that constraints, however our parallel version proved to be more efficient for this case. To support our claims, we present the results of several experiments on a 24 cores share memory machine.

KEYWORDS. Simplex, efficiency, scalability, PM,OC.

1. Introduction

There has been many studies with parallel versions of the simplex method. Many of those have focused on the revised simplex method due to the advantage of solving sparse problems and being most effective where the number of variables is much larger than the amount of restrictions (Yarmish and Slyke, 2003). However, The revised method is not well suited for parallelization (Hall, 2007). Eckstein et. al (Eckstein et al., 1995) describe three simplex parallel implementations, including a revised method, for dense LP problems on the Connection Machine CM-2. They showed that CM-2 can yield better execution times than a workstation from the same era and processing power of the CM-2. Thomadakis and Liu (Thomadakis and Liu, 1996) worked on parallelizing the Standard and Dual simplex algorithm based on steepest-edge, comparing it on two versions of Maspar (MP-1 and MP-2). Their results show that as the problems size increases the speedup obtained by MP-1 and MP-2 is 100 times and 1000 times, respectively, over sequential steepest-edge. Hall and McKinnon (Hall and McKinnon, 1998) studied an asynchronous variant of the revised simplex method on the Cray T3D machine. They present different ways for improving the performance of this algorithm on Cray, using distincts T3D routines. They also explain the potential of this variant for shared memory processors, instead of the parallel distributed implementation on the Cray T3D.

Since mid 2000's, we have been facing a paradigm change where the computers are not being produced with only one processing core. This tendency has being called Era Multicore, detailed by Kock and Borkar (Koch, 2005; Borkar, 2007), which the idea includes the principle of doubling the number of processing cores in a single chip with each technology generation. Thus, it is important to validate parallel algorithms for scalable performance. This means that the program should able to use progressively greater number of processors in an efficiency way.

In 2009, Yarmish and Slyke (Yarmish and Slyke, 2009) presents a scalable simplex implementation for large-scale problems of linear programming using 7 workstations connected by Ethernet. Their parallel standard algorithm is more efficient than the revised method, which was validated by an analytical model.In another work produced in the same year, N. Ploskas et al. (Ploskas et al., 2009) showed a parallel implementation of the standard simplex algorithm using a personal computer with two cores. Due to heavy communication, the computational results show that it is hard to achieve a linear speed-up even with carefully selected partitioning patterns and communication optimization. The standard Simplex is effective for dense problems and when the relation of variables and constraints is not very high (Yarmish, 2006). However, it is more suitable for parallel implement with this relation is not a serious problem (Yarmish and Slyke, 2009).

This paper presents a scalability and efficiency analysis for a parallel implementation of the standard simplex algorithm for multicore processors for solving large-scale linear programming problems. We choose the standard simplex algorithm for being more suitable for parallelization.We verify the scalability of different amounts of constraints and variables for large-scale problems, finding evidence that the standard simplex algorithm has better parallel efficiency for problems with more variables than constraints. One of the disadvantages of the standard algorithm is being less efficient for problems with more variables that constraints, however our parallel version proved to be more efficient. To support our conclusions, we present the results of several experiments on a 24 cores share memory machine.

This paper is organized in the following way: in Section 2 we present the metrics that are necessary to make the scalabily analysis; in Section 3 we present a general scheme for a parallel implementation; in Section 4 we detail our parallel implementation; in Section 5 we show the results and finally in the Conclusion we make our final considerations about our main contributions.

2. Scalability and Efficiency

Among the many challenges of the the multicore era, one of the most important is to check if the algorithms have performance scalability problems, i.e., whether the program is able to use a progressively greater number of processors. Therefore, it is important to accomplish scalable parallelism, which is adaptable to a wide range of parallel platforms.

For scalability analysis, speedup and efficiency are considered the main metrics. The speedup S is defined as the ratio between sequential time[1], T_S and parallel time T_P.

$$S = \frac{T_S}{t_P} \quad (1)$$

The speedup tells how many times the parallel algorithm is faster than the serial algorithm. A linear speedup is that where the algorithm has linear acceleration, i.e., by doubling the number of processors it is obtained twice the speed. Very often real speedups tends to saturate for an increased number of processors. This occurs because the size of the problem is fixed while the number of processors is increased, which implies the reduction of the amount of work per processor. With less work per processor, overhead[2] cost may becomes more significant, so that the relation between the serial time and parallel time does not improve linearly.

The efficiency E is the division of speedup S by the amount of processors P used in the computation.

$$E = \frac{S}{P} \quad (2)$$

Efficiency is a normalized measure of speedup that indicates how effectively each processor is employed. A speedup with value equals to P, have an efficiency equal 1, i.e., all processors are effectively used. However, efficiency is typically less than 1, due to performance loss, and always decreases as the number of processors is increased.

Although efficiency can show how effectively is the processors are being used for a specific problem size, the serial portion of the code can be much larger for small problems. Therefore, to have better insights about the scalability of a parallel algorithm, it is necessary to vary the size of the problem and observe the evolution of the efficiency. For scalable algorithms, it is expected that efficiency scales with an increasing problem size, because the parallel portion increases more than the serial.

[1] Serial time is the execution time of the algorithm implemented sequentially.

[2] Overhead is any processing or storage in excess, whether of computing time, memory, bandwidth or other resource that is required to be used or spent to perform a certain task.

3. General Scheme

For a better understanding of this paper, we explain three important concepts of parallelism present in this article: thread, barrier and critical region.

Thread is a way for a process to divide itself into two or more tasks that can be performed concurrently. It can exist within the same process and share resources such as memory.

Barrier allows synchronization of several threads at a specific point in the code. It is used in applications where all threads must complete one stage before moving all together for the next phase. A barrier for a group of threads means that any thread should stop at this point and can not proceed until all other threads have reached the barrier.

Critical section is a region of the code where only one thread can enter at a time, otherwise the execution becomes inconsistency.

The parallel scheme is composed of 6 steps. The first is to make the initial simplex table with all constraints and variables. The second selects the column with the minimum negative value of the objective row. The third step selects the row with minimum ratio

$$\frac{table[i][LAST_COL]}{table[i][PIVOT_COL]} \tag{3}$$

, where $table[i][LAST_COL]$ is the independent value, $table$ are the constraints, i is the row index and $PIVOT_COL$ is the index of the pivot column. Step IV updates the pivot row. Step V updates the remaining constraint rows and the last step does three operations in the same loop. Those steps can be viewed on Figure 1.

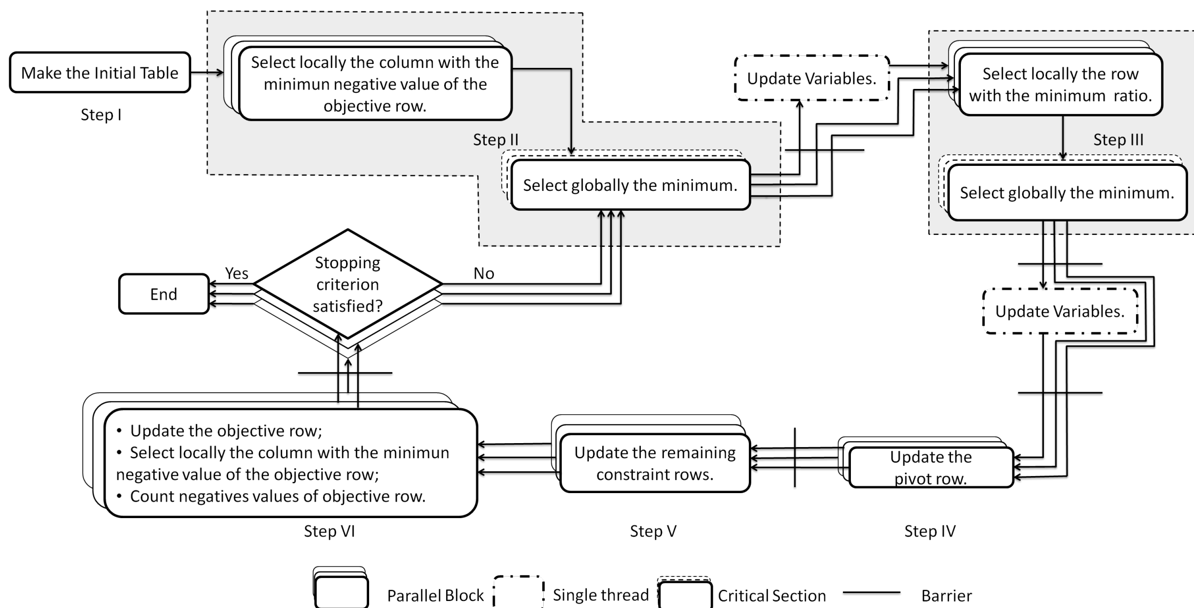


Figure 1. Flowchart of the Simplex algorithm

Step II contains a parallelized loop, i.e., each thread is responsible for a number of columns. They select the minimum negative value of the objective row. After this, it is

necessary to select the threads with the lowest negative value. For this, all threads enter a critical section, as they finish their tasks, comparing their local minimum value with a global variable that, at the end, will hold minimum negative value. In sequence, there is a barrier to make sure that all threads have the same global column.

We use a single thread to update variables and in step III. Then, we locally select the row with the minimum ratio and use a critical region to select the global row, in the same way it happens on step II. A barrier synchronizes the global row among the threads. Another barrier before the step IV is necessary for getting the correct pivot with the row and column selected.

The parallel loop of step IV updates the pivot row. To proceed, it is necessary that this step completes because the values of this row are used in the next step. Therefore, it needs a barrier.

The Step V is a parallelized loop for updating the remaining constraint rows without a barrier. A barrier is not necessary because the next step is independent of this step.

Step VI is an algorithm improvement. Using the same parallel loop, three operations are performed: update the the objective row; select locally the column with the minimum negative value of the objective row; and count the negatives values of the objective row. We have a barrier to ensure the counting is correct, because the stop criterion is satisfied when there is no more negative values in the objective row. If the stop criterion is not reached, the loop restarts in the critical section of the second step to select the global column.

Observe the synchronization between the steps, as they are dependent on each other. For example, step III can only be accomplished if the column index is set. Step IV needs the pivot, i.e., the column selected in step II and the row selected in step III. And the step V needs the pivot row updated. The next section we detail our parallel implementation.

4. Parallel Implementation

The parallelization was done in C++ using OpenMP (Open Multi-Processing), which is an API (Application Programming Interface) platform using shared memory multiprocessing. It consists of a set of directives to the compiler, library functions, and environment variables that specify the implementation of a parallel program (OpenMP, 2013).

One of the key constructions of openMP is the directive **omp parallel** that specifies a parallel section. When a parallel section is encountered, threads are triggered as needed, and they all start running parallel code within that section.

The pseudo-code of our standard simplex parallel implementation can be viewed on Figure 2. Note that the threads are triggered only once to avoid the overhead of creating and destroying threads. For the purpose of code optimization, the fist part of step II runs before the start of the parallel section (lines 4-7). Each thread works locally on a set of columns, finding the index of the column with the minimum negative value in the objective row.

The directive **omp for** (line 4), which divides the interval iterations of the loop between the threads automatically. At the end of a loop preceded with directive **omp for** there is an implicit barrier that synchronizes all threads, i.e., only after all threads perform their jobs is that they will continue to the rest of the program code. However, you can add

the clause **nowait** to the directive in order to remove this barrier since , in the case, there is no need for synchronization after completion of tasks of each thread.

The next step is to find the global column index among the solutions found by each thread. Note that, this index column is a shared variable, i.e., it is in common to all threads. Thus, two or more threads can not read or write that variable at the same time, because if this happens the variable may contain some erroneous value. This is a critical region. To solve this problem, we use the directive **omp critical** (line 11) to ensure access to only one thread at a time. The directive **omp barrier** creates a synchronization barrier to ensure that all threads have the shared variable updated correctly (line 13).

And the directive **omp single** is used to ensure that only one thread execute this section of code which is used to update some control variables (lines 14-15).

```
1  #pragma omp parallel scope_of_the_data
2  {
3    //STEP II
4    #pragma omp for nowait
5    for (j = 0; j <= col_size; j++)
6      /*Select locally the column with the
7       minimun negative value of the objective row.*/
8
9    do{
10
11       #pragma omp critical
12       /*Select globally the minimum (PIVOT_COL).*/
13       #pragma omp barrier
14       #pragma omp single nowait
15       //Update Variables.
16
17       //STEP III
18       #pragma omp for nowait
19       for (i = 0; i < row_size; i++) {
20           ratio =  table[i][LAST_COL] / table[i][PIVOT_COL]
21           /*Select locally the row with the minimum  ratio.*/
22       }
23
24       #pragma omp critical
25       /*Select globally the minimum (PIVOT_ROW). */
26
27       #pragma omp barrier
28       #pragma omp single nowait
29       //Update Variables.
30
31       pivot = table[PIVOT_ROW][PIVOT_COL]
32       #pragma omp barrier
33       //Step IV
34       #pragma omp for
35       for (j = 0; j <= col_size; j++)
36           /*divide each element of the pivot row
37             by the pivot*/
38
39       //STEP V
40       #pragma omp for nowait
41       for (i = 0; i < row_size; i++)
42         if( the row is not the same of the pivot)
43           for (j = 0; j <= col_size; j++)
44             table[i][j] =
45               table[i][j] -
46                 (table[i][PIVOT_COL] * table[PIVOT_ROW][j])
47
48       //STEP VI
49       #pragma omp for reduction(+:count)
50       for (j = 0; j <= col_size; j++){
51           table[LAST_ROW][j] =
52             table[LAST_ROW][j] -
53               (table[LAST_ROW][PIVOT_COL] * table[PIVOT_ROW][j])
54           if(any j elementy of the objective_row is negative){
55               /*Select locally the column with the
56                 minimun negative value of the objective row.*/
57               count = count + 1
58           }
59       }
60
61     } while(count > 0)
62 }
```

Figure 2. Parallel Code

The table of variables is composed by the constraint matrix A, the vector of coefficients of the objective function C and the vector of independent values b, as showed in the table 1. So, the independent values are in the last column and the objectives values are in the last row.

A	b
$-c$	$\mathbf{0}$

Table 1. Initial Table

The third step divides a set of rows for each thread, which find locally the row that has the minimum value of the ratio seen in equation 3 (lines 18-22). Note the use of the directive `critical` (line 24) to find globally the minimum row index. Then all threads are synchronized to get the correct pivot. If there were no barrier (line 27), it could happen that the row index had a wrong value and, consequently, the pivot would be wrong too. The second barrier (line 32) is to ensure that all threads have the same pivot value before proceeding to the next step.

The fourth step parallelizes the pivot row division by the pivot (lines 34-37). Note that there is an implicit barrier after the execution of the loop. This is necessary since the next step needs the normalized pivot row.

The Step V (lines 40-46) is responsible for updating the remaining constraints using the formula presented on lines 44-46. Observe the presence of the clause nowait in the loop directive.

The last step consists of three operations in the same loop: a modification of the objective row using the same formula used in the previous step (lines 49-53); a search per thread for the column with the minimum negative element in the objective row (lines 54-56), i.e., the same operation as the fist part of step II; and an increment of the variable $count$ (line 57).

The clause `reduction` (line 49) makes a local copy of the variable in each thread, but the values of the local copies are summarized (reduced) into a global shared variable at the end of the loop. In our case each thread have a local copy of the variable $count$ and when the threads finish their jobs the local copies of $count$ are added to a shared variable version of $count$. In the end (line 61) is the test for the stop criterion. If there is any negative numbers in the objective row the algorithm continues.

5. Results

For the experiments, the computer used has two AMD Opteron 6172 with 12 cores 2.1 Ghz, 16 GB DDR3 RAM, running Ubuntu 12.04.1 LTS.

To generate the set of LP problems used to test the standard simplex algorithm is necessary create the initial table which is represented by table 1. In order to generate the test problems we implemented in Octave[3] a method in which the values of A, C and b were generated with random numbers. The dimensions of the problems were the following: 256, 384, 512, 768, 1024, 1536, 2048, 3072, 4096. The number of rows and columns were defined as combinations of numbers from this. Each problem was generated 5 times with different data, thereby generating 405 (9 * 9 * 5) problems. For example, the problem 256x256 has 5 different instances: 256x256_1, 256x256_2, 256x256_3, 256x256_4, 256x256 _5. All instances have problems with different values.

All problems were solved using the serial standard simplex algorithm and the proposed parallel implementation with 2, 4, 8, 16 and 24 threads. The execution time considered was normalized per iteration and averaged among the execution times of the 5 problems of the same size.

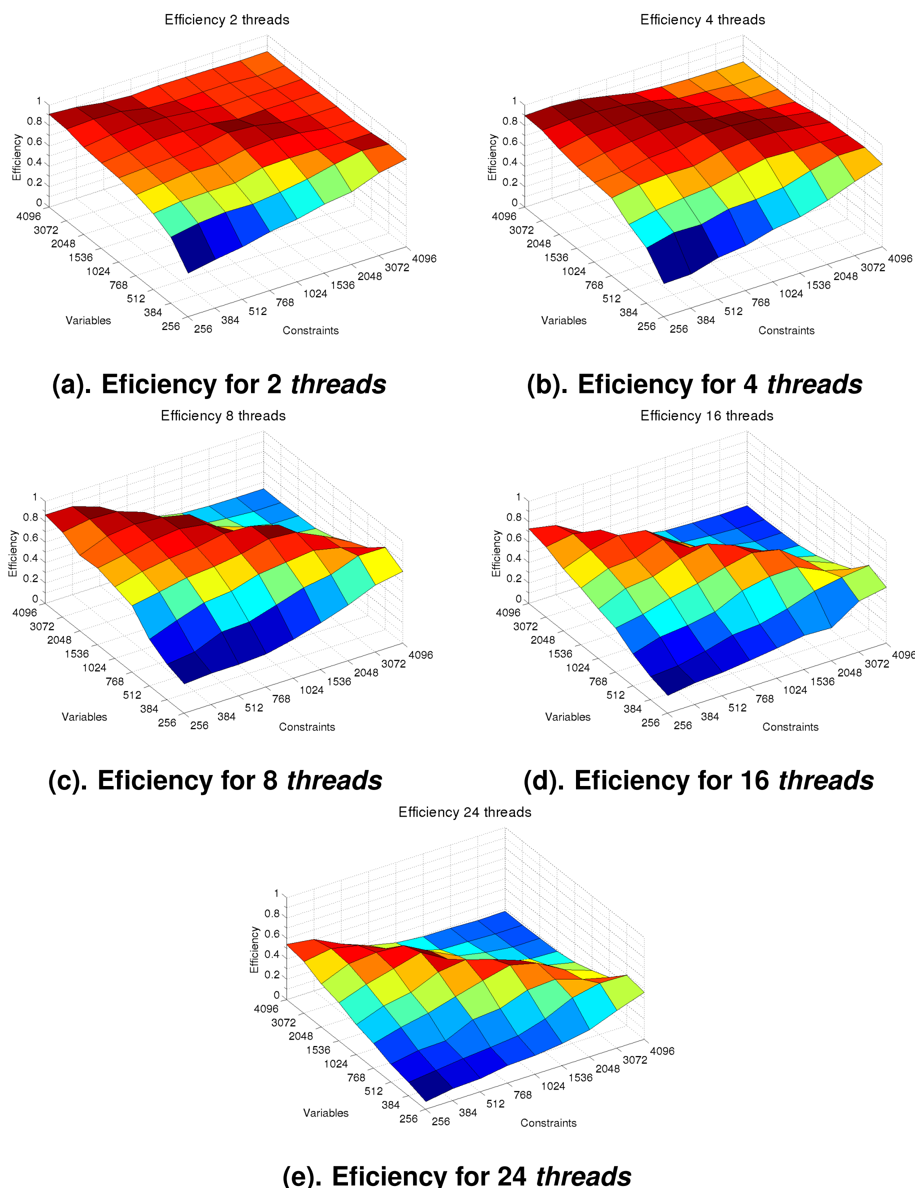
The figure 3 shows the graphics of efficiency for 2, 4, 8, 16 and 24 threads for a varying number of constraints and variables. Note that the rows represent the constraints of the problem, whereas the columns represent the variables.

First, note that as the number of threads increases the efficiency decreases which is common to all parallel systems. Note that for 2, 4, 8 threads the values of efficiency are close to 1. indicating a good use of the processors.

Note that for all threads, the efficiency value scales with the increasing magnitude of the problems up to a certain amount of constraints and variables. Thus, we can say that

[3]GNU Octave is a high-level interpreted language, primarily for numerical calculations. The Octave language is quite similar to Matlab® making scripts and tools often compatible.

the algorithm is scalable to these problems. However, when we further increase the size of the problem, observe a decline in the value of efficiency. This effect can be blamed on the limited amount of cache memory, which may be insufficient to accommodate all the values of large problem sizes, causing the data to be fetched from RAM memory, and thus degenerating the performance.



(a). Eficiency for 2 *threads*

(b). Eficiency for 4 *threads*

(c). Eficiency for 8 *threads*

(d). Eficiency for 16 *threads*

(e). Eficiency for 24 *threads*

Figure 3. Efficiency for varing proble sizes and varing number of threads.

Observe in Figure 3 that setting the number of constraint to 256 and varying the number of variables we see that the efficiency is higher than setting the number of variables 256 and varying the number of constraints. In other words, it shows that for problems with more variables the parallel standard simplex had better efficiency than the problems with more constraints. It happens in all other numbers of the figure 3.

Figure 3 supports what we suspected: the parallel standard simplex algorithm solves problems more efficiently with more variables than constraints. We can say that the

simplex algorithm is more scalable for problems with more variables than constraints. This also shows that in the parallel world it is better to solve problems with more variables, while in the sequential world the knowledge is thtat it was better to solve problems with more constraints.

6. conclusion

One of the main points approached in this work is the importance of the program being able to use progressively greater number of processors in an efficiency way. It is necessary to analyze the scalability of parallel algorithms. We presented a general scheme explaining how we parallelize each step of the standard simplex algorithm detailing important spots of our parallel implementation.

Our parallel simplex algorithm demonstrated good performance and good scalability to various combinations of variables and constraints, showing good efficiency independently of the relation between variables and constraints. Although sequential standard simplex algorithm has difficulties solving problems where the number of variables is higher than the constraints, our parallel implementation proved to be more efficient for these type of problems

Referências

Borkar, S. (2007). Thousand core chips-a technology perspective. *44th ACM/IEEE Design Automation Conference*.

Eckstein, J., Boduroglu, I., Polymenakos, L. C., and Goldfarb, D. (1995). Data-parallel implementations of dense simplex methods on the connection machine cm-2.

Hall, J. A. J. (2007). Towards a practical parallelisation of the simplex method. *Computational Management Science*.

Hall, J. A. J. and McKinnon, K. (1998). Asynplex, an asynchronous parallel revised simplex algorithm. *APMOD95 Conference*.

Koch, G. (2005). Discovering multi-core: Extending the benefits of moore?s law. Technical report, Intel Corporation.

OpenMP (2013). Pthe openmp api specification for parallel programming.

Ploskas, N., Samaras, N., and Sifaleras, A., editors (2009). *A parallel implementation of an exterior point algorithm for linear programming problems*. University of Macedonia.

Thomadakis, M. E. and Liu, J.-C., editors (1996). *An Efficient Steepest-Edge SImplex Algorithm for SIMD Computers*. Departament of Computer Science Texas.

Yarmish, G., editor (2006). *The Simplex Method Applied to Wavelet Decomposition*. Computer and Information Science Brooklyn College.

Yarmish, G. and Slyke, R. V. (2003). retrolp, an implementation of the standard simplex method. Technical report, Departament of Computer and Information Science.

Yarmish, G. and Slyke, R. V. (2009). A distributed, scaleable simplex method. *Supercomputing*.