J Supercomput (2009) 49: 373–381
DOI 10.1007/s11227-008-0253-6

A distributed, scaleable simplex method

Gavriel Yarmish · Richard Van Slyke

Published online: 4 February 2009

Abstract We present a simple, scaleable, distributed simplex implementation for large linear programs. It is designed for coarse-grained computation, particularly, readily available networks of workstations. Scalability is achieved by using the standard form of the simplex rather than the revised method. Virtually all serious implementations are based on the revised method because it is much faster for sparse LPs, which are most common. However, there are advantages to the standard method as well. First, the standard method is effective for dense problems. Although dense problems are uncommon in general, they occur frequently in some important applications such as wavelet decomposition, digital filter design, text categorization, and image processing. Second, the standard method can be easily and effectively extended to a coarse grained, distributed algorithm. Such an implementation is presented here. The effectiveness of the approach is supported by experiment and analysis.

Keywords Linear programming · Standard simplex method · Dense matrices · Distributed computing · Parallel optimization

1 Introduction

We present a simple, scaleable, distributed simplex implementation for large linear programs. It is designed for coarse-grained computation, particularly readily avail-

G. Yarmish (✉)
Computer Information Science, Brooklyn College, 2900 Bedford Ave Brooklyn, New York, NY 11210, USA
e-mail: yarmish@sci.brooklyn.cuny.edu

R. Van Slyke
Computer Science, Polytechnic University, 6 Metrotech Center Brooklyn, New York, NY 11201, USA
e-mail: rvslyke@poly.edu

able networks of workstations. Scalability is achieved by using the standard form of the simplex rather than the revised method.

Most research is focused on the revised method since it takes advantage of the sparsity that is inherent in most linear programming applications. The revised method is also advantageous for problems with a high aspect ratio; that is, for problems with many more columns than rows. On the other hand, there are not many parallel or distributed implementations of the revised method that scale well [5]. Earlier work focused on more complex, and more tightly coupled, networking structures. Hall and McKinnon [4] and Shu and Wu [11] worked on parallel revised methods. Thomadakis and Liu [10] worked on the standard method utilizing the MP-1 and MP-2 MasPar. Eckstein et al. [2] showed in the context of the parallel connection machine CM-2 that the iteration time for parallel revised tended to be significantly higher than for parallel tableau even when the revised method is implemented very carefully. Stunkel [9] found a way to parallelize both the revised and standard methods so that both obtained a similar advantage in the context of the parallel Intel iPSC hypercube.

The standard method can be easily and effectively extended to a coarse-grained, distributed algorithm. We look at distributed linear programming especially optimized for loosely coupled workstations. Yarmish [13] describes such a coarse-grained distributed simplex method, dpLP, in greater detail. This implementation was successful in solving all LP problems in the Netlib repository. For more details on the corresponding serial implementation, see [12].

It should also be noted that although dense problems are uncommon in general, they do occur frequently in some important applications within linear programming [2]. Included among those are wavelet decomposition, image processing [1, 7, 14] and digital filter design [3, 6, 8]. All these problem groups are well suited to the standard simplex method. Moreover, when the standard simplex method is distributed, aspect ratio becomes less of an issue. We simply divide the extra columns among more processors.

Below we model the speedup and scalability achievable with our method. We then show speedup from actual runs on 7 machines to validate the model.

2 General scheme

We assume that the reader has basic familiarity with the simplex method.

The simplex method consists of three basic steps:

High-level serial algorithm

a. Column choice.
b. Row choice.
c. Pivot.

A relatively straightforward parallelization scheme within the standard simplex method involves dividing up the columns amongst many processors. Instead of three basic steps, we would then have five basic steps:

High-level parallel algorithm

a. Column choice—each processor will "price out" its columns and choose a locally best column (Computation).

b. Communication amongst the processors of the local best columns. All that is sent is the pricing value (a number) of the processor's best column. At the end of this step, each processor will know which processor is the "winner" and has the global column choice (Communication).
c. Row choice by the winning column (Computation).
d. A broadcast of the winning processor's winning column and choice of row (Communication).
e. A simultaneous pivot by all processors on their columns (Computation).

3 Models and analysis

Let p be the number of homogeneous processors.

Let m and n, respectively, refer to number of rows and columns of the linear program to be solved.

Let *mult* and *add* refer to the time it takes to do one multiplication/division and one addition/subtraction respectively by each processor.

Let s and g refer to the communication latency (startup time) and the throughput (items/sec), respectively.

Using these terms, we may provide an expression for the amount of iteration time for both the serial single-processor standard simplex algorithm and for our parallel scheme for the standard simplex algorithm. Assume we are using the classical column choice rule within the standard simplex method. We can approximate the time per iteration of the serial algorithm by

$$\begin{aligned} T_{\text{serial}} &= n^{*}add;\ (columnchoice) \\ &\quad + m^{*}mult;\ (rowchoice) \\ &\quad + nm^{*}mult;\ (parallel\ pivot) \\ &= > \\ T_{\text{serial}} &= n(add + m^{*}mult) + m^{*}mult. \end{aligned} \tag{1}$$

The time per iteration as a function of p can then be approximated by

$$\begin{aligned} T_{\text{parallel}} = f(p) &= \frac{n}{p}add;\ (columnchoice) \\ &\quad + p(s+g);\ (Communication) \\ &\quad + m^{*}mult;\ (rowchoice\ of\ winning\ p) \\ &\quad + (s+mg);\ (columnbroadcast\ by\ winning\ p) \\ &\quad + \frac{nm}{p}mult;\ (parallelpivot) \\ &= > \\ T_{\text{parallel}} = f(p) &= \frac{n}{p}add + p(s+g) + m^{*}mult + (s+mg) + \frac{nm}{p}mult. \end{aligned}$$

Each of the five terms of T_{parallel} corresponds to one of the 5 basic steps of the algorithm given in Sect. 2. Combining terms yields

$$f(p) = \frac{n}{p}(add + m^*mult) + p(s+g) + m^*mult + (s+mg). \tag{2}$$

We calculate the optimal number of processors to use for a given problem: take the derivative of the timing function T_{parallel} with respect to p, set it to zero, and solve for the optimum number of processors p^*. p^* is not necessarily an integer, but for simplicity in the exposition we use fractional p^*'s.

$$\begin{aligned} f'(p) &= -\frac{n}{p^2}(add + m^*mult) + (s+g) = 0 \\ &= > \\ (p^*)^2 &= \frac{n(add + m^*mult)}{s+g}, \qquad (3) \\ &= > \\ p^* &= \sqrt{\frac{n^*add + m^*n^*mult}{s+g}}. \qquad (4) \end{aligned}$$

The time per iteration reaches a minimum at p^* as can be seen from the second derivative which is positive:

$$f''(p) = \frac{2n}{p^3}(add + m^*mult) > 0 \quad \text{for all } p > 0.$$

Next, we calculate the optimal time per iteration (T_{opt}) assuming use of the optimum number of processors p^*. First, multiply both numerator and denominator of the first two terms of (1) by p to yield

$$f(p) = \frac{pn(add + m^*mult)}{p^2} + \frac{p^2(s+g)}{p} + m^*mult + (s+mg).$$

Next substitute $(p^*)^2$ for p^2 (3) and p^* for p (4) to get the time per iteration when using the optimal number of processors p^*.

$$\begin{aligned} T_{\text{opt}} = f(p^*) &= \frac{pn(add + m^*mult)}{[\frac{n(add+m^*mult)}{s+g}]} + \frac{n(add + m^*mult)(s+g)}{p(s+g)} \\ &\quad + m^*mult + (s+mg) && \text{(substitute } (p^*)^2\text{)} \\ &= p(s+g) + \frac{n(add + m^*mult)}{p} + m^*mult + (s+mg) && \text{(substitute } p^*\text{)} \\ &= \frac{\sqrt{n(add + m^*mult)}}{\sqrt{s+g}}(s+g) + \frac{n(add + m^*mult)}{[\frac{\sqrt{n(add+m^*mult)}}{\sqrt{s+g}}]} + m^*mult + (s+mg) \end{aligned}$$

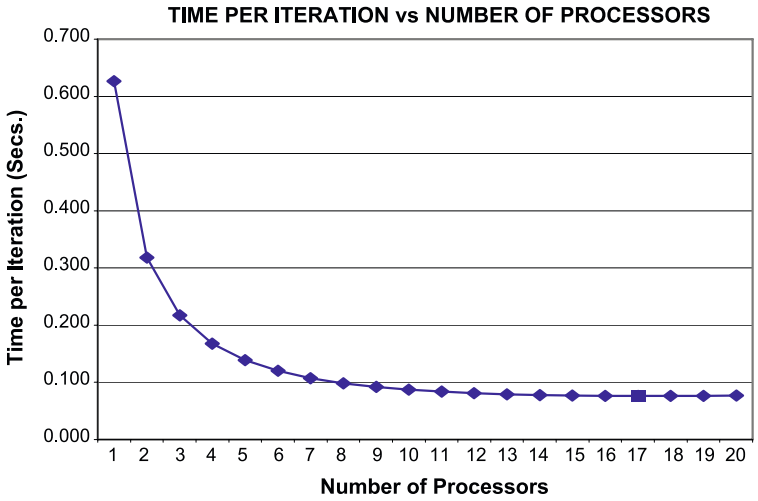

Fig. 1 Iteration time vs. number of processors

$$= \sqrt{n(add + m^*mult)}\sqrt{s+g} + \sqrt{n(add + m^*mult)}\sqrt{s+g} + m^*mult + (s + mg)$$

$$=>$$

$$T_{\text{opt}} = 2\sqrt{n(add + m^*mult)}\sqrt{s+g} + m^*mult + (s + mg). \tag{5}$$

The speedup of the parallel scheme relative to the serial algorithm is

$$Speedup = \frac{T_{\text{serial}}}{T_{\text{opt}}} = \frac{n(add + m^*mult) + m^*mult}{2\sqrt{n(add + m^*mult)}\sqrt{s+g} + m^*mult + (s + mg)}.$$

Figure 1 graphically demonstrates our timing model T_{parallel} (2), the time per iteration as a function of p, for a problem with $m = 1{,}000$ rows, and $n = 5{,}000$ columns. The workstations and Ethernet used for the experiments had measurements: $mult = 1.24$ E-7, $add = 3.73$ E-8, $s = 2.10$ E-3 and $g = 1.76$ E-6. These are the parameters used for T_{parallel} depicted in Fig. 1.

From the graph, we see that addition of processors causes great initial speedup. As more processors are added, the amount of speedup begins to level off. The reason for this is the ratio of computational speed to communication speed. Initially, for every additional processor, there is a very large computation savings and a very low communication cost. As processors are added, the computation gain lessens while communication costs rise.

From the figure, it is hard to determine precisely the optimum number of processors. Using the formula for p^*, we determine that $p^* = 17$, and the optimal speedup relative to serial processing is about 8.

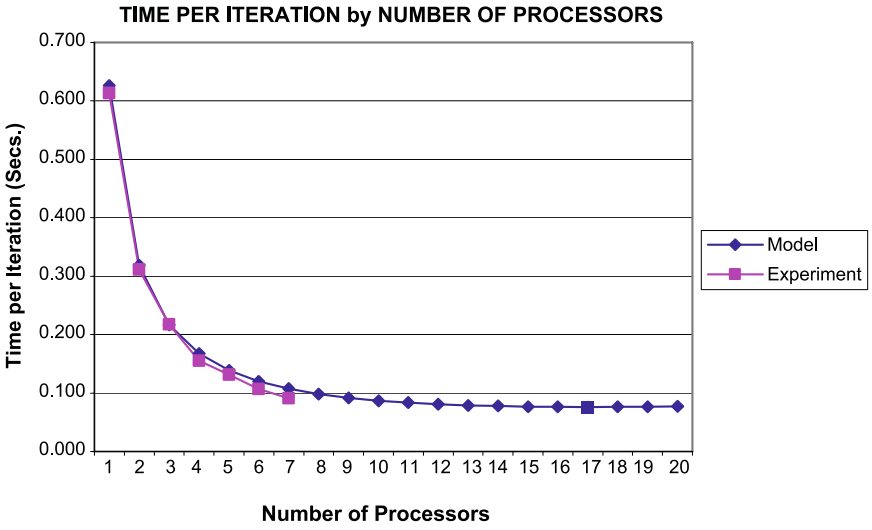

Fig. 2 Iteration time: model and using dpLP

As problems get larger, the number of processors that may prove useful rises, as does the speedup. For example, for $m = 10{,}000$, and $n = 50{,}000$, p^* would be 172, and the speedup relative to the serial processor would be 83.

4 Experimental results

We have conducted experiments on many problems using our implementation of the parallelized version of the standard simplex method. In particular, we report on a problem with the dimensions ($1{,}000 \times 5{,}000$) used in our analysis of Sect. 3. We used a lab that had seven independent workstations connected by an Ethernet.

4.1 Verification of model

In order to verify our model, developed in Sect. 3, we ran numerous problems using our implementation of the parallel simplex (dpLP). Figure 2 compares the time per iteration with utilization of one processor, with utilization of two processors and so on, through utilization of all seven processors. One can see how close the actual timing is to the time predicted by the model.

4.2 Comparison with the revised simplex

As noted in the Introduction, one of the main incentives for our focus on the standard simplex method as opposed to the revised method is the ability to have a scaleable parallel algorithm for the simplex method. We also know that the density of the linear programming matrix is a major factor affecting the efficiency of the revised simplex method, since the revised method is quite sensitive to density. Although problems

Table 1 Time per iteration (secs) vs. number of processors

Processors	Parallel-dpLP
1	0.61328
2	0.31150
3	0.21724
4	0.15496
5	0.13114
6	0.10658
7	0.09128

Table 2 Time per iteration (secs) vs. density for MINOS

Density	Revised-MINOS
5%	0.04848
10%	0.08726
20%	0.16463
40%	0.29643
50%	0.38814
60%	0.48544
70%	0.57012
80%	0.64688
90%	0.70477
100%	0.79544

with a sparse matrix are more common, there do exist applications with dense matrices; several are referenced in our Introduction. We demonstrate the interaction of both factors, parallelization and density, to provide an understanding of the advantages of our method.

In the second experiment, in addition to timing our problem within the standard method, we also timed the problem on MINOS, a well-known implementation of the revised simplex. MINOS is commonly used in analysis due to the availability of its source code.

Table 1 shows the average running time per iteration for dpLP running the standard simplex algorithm repeatedly as the number of processors used varies from 1 to 7.

Table 2 shows the time per iteration taken by MINOS as a function of density. Performance for our standard implementation was unchanged with changes of densities.

Examination of these two tables reveals the effects of both the number of processors and the density. For densities of 20%, 4 processors are enough to make the full tableau standard method more efficient than the revised method. When we used 7 processors, the standard method outperformed the revised method at density slightly above 10%. According to our model, if we were to have 17 processors for this $1{,}000 \times 5{,}000$ size problem, it should take about 0.0762 seconds per iteration, which would make the full tableau method more efficient than the revised method for a density well below 10%.

5 Summary

In summary, the original standard simplex method has two advantages over the revised method.

1. It is possible to build a scaleable parallel version of the standard method whereas the revised method is difficult to parallelize.
2. The standard method is not affected by problem density.

The combination of these two factors allows our parallel algorithm to be useful for a significant number of applications.

Our model was both studied and implemented in the context of off-the-shelf independent workstations. The advantage of that is that tightly-coupled Massively Parallel Processors (MPP) are becoming less popular as off-the shelf small processors are becoming more powerful. We have demonstrated that large problems can be solved using any underutilized lab of workstations. These networks are extremely common.

We further have done an analysis showing the optimal number of processors (workstations) that should be used. This analysis can be repeated for any network of workstations to find the network-specific optimal.

We believe that as more people see the feasibility of solving dense problems on networks of workstations the original tableau would be used in a more prominent fashion in the solving of such problems.

Acknowledgements This paper has been supported in part by a grant from PSC-CUNY.

References

1. Chen SS, Donoho DL, Saunders MA (1998) Atomic decomposition by basis pursuit. SIAM J Sci Comput 20(1):33–61
2. Eckstein J, Boduroglu I, Polymenakos L, Goldfarb D (1995) Data-parallel implementations of dense simplex methods on the connection machine CM-2. ORSA J Comput 7(4):402–416
3. Gislason E et al (1993) Three different criteria for the design of two-dimensional zero phase FIR digital filters. IEEE Trans Signal Processing 41(10):3070–3074
4. Hall JAJ, McKinnon KIM (1998) ASYNPLEX an asynchronous parallel revised simplex algorithm. Ann Oper Res 81:27–50
5. Hall JAJ (2007) Towards a practical parallelisation of the simplex method. Optimization Online. http://www.optimization-online.org/DB_FILE/2005/02/1061.ps
6. Hu JV, Rabiner LR (1972) Design techniques for two-dimensional digital filters. IEEE Trans Audio Electroacoust AU-20:249–257
7. Selesnick IW, Van Slyke R, Guleryuz OG (2004) Pixel recovery via l_1 minimization in the wavelet domain. In: IEEE international conference on image processing (ICIP) 2004, Singapore
8. Steiglitz K, Parks TW, Kaiser JF (1992) METEOR: a constraint-based FIR filter design program. IEEE Trans Signal Proc 40(8):1901–1909
9. Stunkel CB (1988) Linear optimization via message-based parallel processing. In: ICPP: international conference on parallel processing, 1988, vol III, pp 264–271
10. Thomadakis ME, Liu J-C (1996) An efficient steepest-edge simplex algorithm for SIMD computers. In: Proc of the international conference on super-computing, ICS '96, May 1996, pp 286–293
11. Shu W, Wu M-Y (1993) Sparse implementation of revised simplex algorithms on parallel computers. In: 6th SIAM conference in parallel processing for scientific computing, March 1993, pp 501–509
12. Yarmish G (2001) A distributed implementation of the simplex method. Ph.D. dissertation, Polytechnic University, Brooklyn, NY

13. Yarmish G (2007) Wavelet decomposition via the standard tableau simplex method of linear programming. WSEAS Trans Math 6:170–177. http://www.wseas.us/e-library/conferences/2006dallas/papers/519-254.pdf
14. Yarmish G, Van Slyke R (2006) A description of and motivation for retrolp, an implementation of the standard simplex method. In: Proceedings of the 5th annual Hawaii international conference on statistics, mathematics and related fields, January 16–18, 2006, pp 1704–1714

Gavriel Yarmish is a professor of Computer Science at Brooklyn College of the City University of New York. His research interests include distributed and parallel optimization methods, and optimization of large mathematical programs.

Richard Van Slyke is Professor Emeritus of Computer Science at the Polytechnic Institute of NYU. His research interests are in applications of, algorithms for, and computer architecture applied to optimization.