Parallel Simplex Algorithm for Circuit Optimisation

Arpad Bűrmen, Janez Puhan, Tadej Tuma, Iztok Fajfar, Andrej Nussdorfer
Faculty of Electrical Engineering
University of Ljubljana
Tržaška cesta 25, 1001 Ljubljana, Slovenija
arpadb@fides.fe.uni-lj.si
http://fides.fe.uni-lj.si/spice

Abstract

Probably the largest obstacle to overcome when optimisation is applied to real-world circuits is the long time it takes to complete an optimisation run. The most intuitive solution is the use of parallel algorithms. One of the algorithms that have given good results in conjunction with the SPICE circuit simulator is the constrained simplex algorithm. The research in this paper focuses on its parallel implementation tailored for use on heterogeneous networks of workstations connected by a local area network (LAN). Such architectures are readily available in every design community and are a low cost solution. The major downside of such approach is the relatively large overhead of LANs that are typically used in circuit design groups. A parallel implementation of the constrained simplex algorithm in SPICE OPUS is presented. The algorithm is tested on *2* test functions and *3* circuit optimisation problems. The performance of the algorithm is examined in terms of speed and computational efficiency. The results are compared to the ones obtained by the original constrained simplex algorithm.

1. Introduction

As the computational power of a desktop workstation grew, circuit simulation evolved from an aid available only to designers with high-end computer hardware into a widely used tool.

Circuit optimisation requires many cost function evaluations and thus many simulations of the circuit, which makes it a computationally intensive task. It is performed by a circuit designer on a daily basis in the form of parameter tuning based on designer's past experience. Automated approaches are rarely used, mostly due to the long time needed for an optimisation run to complete.

One way of removing this obstacle is making the underlying algorithms computationally more efficient. In the past, a lot of effort in the aforementioned direction was invested in making typical circuit designer's jobs like Monte Carlo analysis or design centering more efficient and thus faster.

Developments in the direction of algorithm parallelisation occurred in the area of global optimisation with methods like genetic algorithms or simulated annealing, which by their nature bear a large computational cost. Much less effort has been observed in the area of direct optimisation methods (e.g. simplex algorithm).

Despite the fact that simplex methods have been around since 1960s they are still widely used for many optimisation problems. The key to their popularity probably lies in the fact that the algorithm itself is simple, requires no derivatives of the cost function, and mostly avoids getting trapped in local minima.

This article focuses on the parallel implementation of the constrained simplex algorithm (CSA) optimised for networks of workstations connected by a LAN (typically 10/100Mbit Ethernet). The state of the art in the field of parallel optimisation with direct methods is given. The discussed algorithm for parallel constrained optimisation is presented. The problems in the test suite are described and the results of the application of the parallel CSA are presented. Finally the conclusions are summarised and suggestions for future research are given.

2. State of the Art

The constrained simplex (CSA) algorithm [2] is an extension of the Nelder-Mead simplex algorithm (NMSA) [1]. Both of these algorithms search for the minimum of the cost function in N-dimensional space. For NMSA the number of points in the simplex is usually $N+1$ although the algorithm can handle simplexes consisting of more than $N+1$ points. A larger number of points generally means better search capability and more cost function evaluations.

In the past there has been some work on the parallelisation of the NMSA. Experiments by Dennis and Torczon [3] focused on doing simplex operations on several worst points in parallel. The results were mixed. The parallel implementation exhibited increased performance in some cases. On the other hand cases were demonstrated where the method performed worse than the original NMSA.

Much better results were obtained by the method proposed by Coetzee and Botha [4]. A slightly modified version of NMSA was implemented on a shared memory parallel machine. The algorithm showed improved performance over the original NMSA both in terms of cost function evaluations and quality of results.

Up to now we haven't found a parallel implementation of the CSA in the literature. CSA is simpler than NMSA since it uses only mirroring and contraction. In order to sample multiple regions of space delimited by implicit constraints CSA manipulates a simplex with more than $N+1$ points. Empirical results by Box suggested $2N$ for the simplex size. Again the search capability and number of cost function evaluations increase with the number of points in the simplex.

The parallel implementation of the constrained simplex algorithm has been tested with SPICE OPUS [5], [6]. The Windows95/98/NT, LINUX and SOLARIS release can be downloaded from http://fides.fe.uni-lj.si/spice.

3. The Method

The following steps are taken for an *N*-dimensional optimisation run:

Initialise simplex:

1. Initial point must satisfy all implicit constraints.
2. Choose $k-1 \geq N$ points randomly. If a point doesn't satisfy some implicit constraint, repeat contraction towards initial point until all implicit constraints are satisfied:
$$P_{chosen}^{(i+1)} = \tfrac{1}{2}\left(P_{chosen}^{(i)} + P_{initial}\right) \quad (1)$$

The CSA performs the following sequence of steps on *k* simplex points:

1. Sort the points by their descending cost function value so that $E(P_1) \geq E(P_2) \geq \ldots \geq E(P_k)$. Calculate the centroid (P_C) of points $P_2,\ldots,P_k$:
$$P_C = \tfrac{1}{k-1}\sum_{i=2}^{k} P_i \quad (2)$$
2. Mirror P_1 over the centroid with factor $\alpha > 1$
$$P_1'^{(1)} = (1-\alpha)P_1 + \alpha P_C \quad (3)$$
3. While $P_1'^{(i)}$
   - violates implicit constraints or (4a)
   - $E(P_1') \geq E(P_1)$ (4b)

   contract it towards the centroid:
$$P_1'^{(i+1)} = \tfrac{1}{2}\left(P_1'^{(i)} + P_C\right) \quad (5)$$
   If the contracted point comes closer to the centroid than the desired final relative simplex size, start over with $P_1'^{(1)}$ and begin contracting it towards the point with the lowest cost function value (P_k):
$$P_1'^{(i+1)} = \tfrac{1}{2}\left(P_1'^{(i)} + P_k\right) \quad (6)$$
   In case contraction towards P_k fails (contracted point is closer to P_k than the desired final relative simplex size, and conditions (4a) and (4b) are not fulfilled), P_1 becomes P_k.
4. Repeat steps 1-3 until simplex is small enough.

It can be seen from the algorithm that at least 1 cost function evaluation is required for one pass through steps 1-3.

One way to parallelise the algorithm is to mirror multiple worst points at the same time. Instead of executing steps 2-3 only for the worst point, they can be executed for *m* worst points. All of the worst points that were mirrored are then replaced by the outcomes of step 4. The centroid in step 1 is calculated using only the *k-m* best points. Every sequence of steps 2-3 can be executed by one slave processor. The algorithm requires *m* parallel slave processors and one master processor. For every iteration a slave processor gets the calculated centroid, the best point, and one of the *m* worst points. After step 3 is completed, the slave processor returns the outcome (new point and its cost function value) to the master processor, which keeps track of all points in the simplex and assigns data for steps 2-3 to slave processors. Since the master processor's work is simple and completed in a fairly short amount of time, the algorithm can be executed on *m* workstations where one workstation runs the master process as well as one of the slave processes.

4. Implementation

The communication among processes on workstations was implemented using TCP/IP. The SPICE OPUS executable was modified to provide two modes of operation: master and slave. In master mode the simulator runs as a normal simulator. By issuing a 'drone' command in SPICE master prompt, a request for a slave process can be sent to a particular workstation. The workstation must run a 'Drone Controller' program, which awaits such requests and upon receiving one starts a slave process. The SPICE slave process (SPICE OPUS executable in slave mode) is instructed to connect to the master process.

The master process broadcasts all of the commands and netlists it interprets to the slave processes. Exceptions are commands like 'plot' and of course the 'optimize' command. This way the slaves do exactly the same things as the master thus creating the same state of data and circuit as in the master process. When an 'optimize' command is reached by the master the behaviour changes. The master starts dividing jobs among slaves. The slaves await jobs and return results after job execution. The master collects the results from slaves and checks for convergence. If convergence is reached, the slaves are notified and a summary of results is sent to all slaves so that the state of memory for all slaves synchronises with the master. From that point on master continues to broadcasts commands it executes to slaves as it did before the 'optimize' command.

By utilising such methods of communication and synchronisation, existing netlists with their .control blocks can be simulated and optimised in a parallel manner without any major changes in them. Two things have to be done prior to running a circuit through the simulator:

1. a sequence of drone commands that create slave processes on workstations has to be issued,
2. the parallel simplex method has to be selected ('optimize method ...' command).

5. Examples and Results

Since CSA performance depends on the set of initial points that are chosen randomly, all optimization times and numbers of CF evaluations are averaged over multiple runs. The average network overhead is calculated by:

$$\overline{T}_{overhead} = \overline{T}_L - \frac{\overline{T}_1}{L} \quad (7)$$

Where L stands for the number of parallel workstations and $\overline{T}_L$ stands for the average time per CF evaluation when L parallel workstations are used.

5.1 Test Functions

The parallel algorithm was tested on two test functions: Rosenbrock function (*2* dimensional) and Woods function (*4* dimensional). Both of them are de-facto standard test functions for optimisation algorithms. Both of them have long curved valleys that present a major problem for most algorithms. The number of points in the simplex was set to *8* for the Rosenbrock function and *11* for the Woods function. Optimisation was stopped when the relative simplex size was smaller than *0.001*. The tests were conducted using a network of *3* workstations connected by *10Mbit* Ethernet. Since the number of CF evaluations and consequently the run time depends on the initial simplex configuration (which is random), the results were averaged over *10* runs. The results are summarized in Table 1 and Table 2.

Mean CF evaluations			
Workstations	**1**	**2**	**3**
Rosenbrock	753	859	981
Woods	1254	1346	1476

Table 1: Comparison of number of CF evaluations for various number of parallel workstations

As it can be seen from Table 1, the number of CF evaluations increases with the number of parallel workstations. This reduces the overall acceleration that can be achieved by parallelisation.

Mean optimisation time (s)			
Workstations	**1**	**2**	**3**
Rosenbrock	1.47	1.11	1.10
Woods	5.11	3.34	2.60

Table 2: Comparison of time needed for the method to converge for various number of parallel workstations.

The mean optimisation time for both test functions shows a slight, but not significant acceleration for the Rosenbrock test function. This is something one would expect since CF evaluation takes a fairly small amount of time (same order of magnitude as it takes for a message to be transported between two workstations). Therefore the network overhead determines the overall performance of the algorithm.

The situation changes in favour of the CF evaluation time for the Woods test function, where significant acceleration can be achieved by using *2* workstations.

5.2 Test Circuits

Three cases were tested: linearisation of a BJT amplifier, Schmitt trigger optimisation and robust design of a low-pass filter [7], [8]. Table 3 gives detailed information on every test circuit.

	BJT amplifier	**Schmitt**	**Low-pass**
Parameters	2	2	5
k	8	8	10
Constraints	1	1	0
Cases averaged	10	10	5

Table 3: Number of optimised parameters, number of points in the simplex and number of implicit constraints.

The robust filter design case is new and is derived from the 5th order low-pass filter test case. The goal is to design a low-pass filter with less than *2dB* ripple between *0.1Hz* and *85Hz*, and more than *70dB* attenuation at *350Hz*. The capacitors in the circuit have *2%* tolerances. The cost function increases linearly from *0* if any of the design requirements is not fulfilled. Worst case results are obtained by doing *20* iterations of Monte Carlo analysis for each examined point in design space.

The results for the three test circuits are given in Table 4 and Table 5.

Mean CF evaluations			
Workstations	**1**	**2**	**3**
BJT Amplifier	245	244	239
Schmitt trigger	407	416	442
Low-pass filter	250	167	191

Table 4: Comparison of number of CF evaluations for various number of parallel workstations

All cases show significant acceleration when optimisation is performed in parallel (Table 5).

Mean optimisation time (s)			
Workstations	**1**	**2**	**3**
BJT amplifier	1.57	0.84	0.48
Schmitt trigger	50.07	24.96	18.14
Low-pass filter	270.01	93.11	72.43

Table 5: Comparison of time needed for the method to converge for various number of parallel workstations.

The comparison of Table 6 and Table 8 confirms the assumption that the performance is limited by network overhead. It can be seen that the algorithm performs well when one CF evaluation takes at least *5ms* (see numbers for *1* workstation in Table 6).

Mean time/CF evaluation (ms)			
Workstations	**1**	**2**	**3**
Rosenbrock	1.59	1.29	1.11
Woods	4.07	2.48	1.76
BJT amplifier	6.41	3.44	2.00
Schmitt trigger	123	60	41
Low-pass filter	927	556	379

Table 6: mean time per CF evaluation.

To further illustrate the effect of network overhead Table 7 summarises the results obtained using (7). As it can be seen, the network overhead affects the performance significantly only for the first two cases.

Mean overhead/CF evaluation (ms)

Workstations	2	3
Rosenbrock	0.31 (24%)	0.34 (31%)
Woods	0.44 (18%)	0.40 (23%)
BJT amplifier	0.24 (7.0%)	0.13 (6.5%)
Schmitt trigger	1.52 (2.5%)	1.08 (2.6%)
Low-pass filter	17.2 (3.1%)	19.4 (5.1%)

Table 7: mean overhead per CF evaluation (percent values are relative to mean CF evaluation time).

The overall acceleration factor is listed in Table 8. For the linearisation of a BJT amplifier and robust filter design the acceleration factors exceed the number of parallel workstations. This could be attributed to increased efficiency of the parallel CSA due to parallel exploration of multiple directions. The assumption is confirmed in Table 4 (the parallel algorithm indeed needs less CF evaluations when multiple parallel workstations are used).

Acceleration factor		
Workstations	**2**	**3**
Rosenbrock	1.32	1.34
Woods	1.53	1.97
BJT amplifier	1.87	3.27
Schmitt trigger	2.01	2.76
Low-pass filter	2.90	3.73

Table 8: Acceleration factor for various test cases and numbers of workstations.

6. Conclusions

A parallel method for constrained optimisation was presented. The method was verified on several test cases. A significant increase in speed roughly proportional to the number of parallel workstations was demonstrated for cases where one CF evaluation takes more than *5ms*.

An iteration of CSA steps 2-3 that are executed by one workstation can require *1* or more CF evaluations. The upper limit of these numbers is determined by the relative simplex size where the algorithm terminates. In case this size is smaller, more CF evaluations can happen in steps 2-3. Since not all workstation need the same number of CF evaluations some of them have to wait for others to finish before they get new data to process from the master. The same can happen in an environment where workstations differ in speed. In the latter case the performance would be bounded from below by the slowest workstation.

Both of the aforementioned disadvantages could be at least partially solved by researching in the following two directions:

- dynamic load balancing among workstations
- dividing the work from steps 2-3 more equally among workstations in a more predictable manner

The algorithm itself also requires more evaluation on real-world circuit design problems. Especially robust circuit design (which is performed more or less manually in the ASIC design community) could be a potential area where the presented parallel algorithm and its derivatives could be of great benefit. There are several reasons for this:

- most of the ASIC design simulations take a significant amount of time which makes the use of any optimisation algorithm impractical,
- the network infrastructure required by the algorithm is cheap and readily available in ASIC design communities,
- by using faster processors or more workstations the speed can be increased without major additional cost which is not the case with dedicated parallel hardware and hardware accelerators.

7. References

[1] J. A. Nelder, R. Mead, “A Simplex Method for Function Minimization”, Computer Journal, Vol. 7, pages 308-313, 1965

[2] M. J. Box, “A New Method of Constrained Optimization and a Comparison with Other Methods”, Computer Journal, Vol. 7, pages 42-52, 1965

[3] J. E. Dennis, Jr., V. Torczon, “Parallel Implementations of the Nelder-Mead Simplex Algorithm for Unconstrained Optimization”, Proceedings of the SPIE, Vol. 880, pages 187-191, 1988

[4] L. Coetzee, E. C. Botha, “The Parallel Downhill Simplex Algorithm for Unconstrained Optimisation”, Concurrency: Practice and Experience, Vol. 10(2), pages 121-137, 1998

[5] T. Quarles, A. R. Newton, D. O. Pederson, A. Sangiovanni-Vincentelli, *“SPICE3 Version 3f4 User’s Manual”*, University of California,Berkeley, California, 1989

[6] J. Puhan, T. Tuma, *"Optimization of analog circuits with SPICE 3f4"*, Proceedings of the ECCTD’97, Volume 1, pages 177 - 180, 1997

[7] J. Puhan, T.Tuma, I. Fajfar, *“Optimisation Methids in SPICE, a Comparison”*, Proceedings of the ECCTD’99, Volume 1, pages xxx-xxx, 1999

[8] “Application notes”, http://fides.fe.uni-lj.si/spice/applications/applications.html, 2001

Povprečno število računanj kriterijske funkcije			
Št. računalnikov	**1**	**2**	**3**
Rosenbrock	753	859	981
Woods	1254	1346	1476
Linearizacija	245	244	239
Schmitt trigger	407	416	442
NP filter	250	167	191

Povprečen čas optimizacije (s)			
Št. računalnikov	**1**	**2**	**3**
Rosenbrock	1.47	1.11	1.10
Woods	5.11	3.34	2.60
Linearizacija	1.57	0.84	0.48
Schmitt trigger	50.07	24.96	18.14
NP filter	270.01	93.11	72.43

Pospešitev		
Št. računalnikov	**2**	**3**
Rosenbrock	1.32	1.34
Woods	1.53	1.97
Linearizacija	1.87	3.27
Schmitt trigger	2.01	2.76
NP filter	2.90	3.73