MemComputing Integer Linear Programming

Fabio L. Traversa*
MemComputing, Inc., San Diego, CA, 92130 CA

Massimiliano Di Ventra†
Department of Physics, University of California, San Diego, La Jolla, CA 92093

(Dated: August 31, 2018)

Integer linear programming (ILP) encompasses a very important class of optimization problems that are of great interest to both academia and industry. Several algorithms are available that attempt to explore the solution space of this class efficiently, while requiring a reasonable compute time. However, although these algorithms have reached various degrees of success over the years, they still face considerable challenges when confronted with particularly hard problem instances, such as those of the MIPLIB 2010 library. In this work we propose a radically different *non-algorithmic* approach to ILP based on a novel physics-inspired computing paradigm: Memcomputing. This paradigm is based on *digital* (hence scalable) machines represented by appropriate electrical circuits with memory. These machines can be either built in hardware or, as we do here, their equations of motion can be efficiently simulated on our traditional computers. We first describe a new circuit architecture of memcomputing machines specifically designed to solve for the linear inequalities representing a general ILP problem. We call these *self-organizing algebraic circuits*, since they self-organize dynamically to satisfy the correct (algebraic) linear inequalities. We then show simulations of these machines using MATLAB running on a single core of a Xeon processor for several ILP benchmark problems taken from the MIPLIB 2010 library, and compare our results against a renowned commercial solver. We show that our approach is very efficient when dealing with these hard problems. In particular, we find within minutes feasible solutions for one of these hard problems (f2000 from MIPLIB 2010) whose feasibility, to the best of our knowledge, has remained unknown for the past eight years.

I. INTRODUCTION

Integer programming represents an important tool to describe a variety of optimization problems that appear both in industry and academia [1]. The general format of integer programming consists of an objective function to be minimized over a set of variables and subjected to a set of constraints defined by linear inequalities among variables. In addition, the variables are constrained to be integer values. If the objective function is linear, then we properly refer to Integer Linear Programming (ILP), which is the problem class we consider in this paper.

Due to its fundamental and practical importance, ILP is still extensively studied in both academia and industry. Several general-purpose open source [2–6] and commercial solvers [7–10] have been developed together with specialized solvers specifically optimized for ILP, with additional structures usually developed for some specific projects [11–14].

Solutions to ILP can be approached via different algorithms, both heuristics [13, 15, 16] and exhaustive [1, 17–19]. Quite often, exhaustive methods are also called "complete algorithms". The complete algorithm for ILP that is most commonly employed is a combination of cutting planes and branch-and-bound, also known as the branch-and-cut algorithm [1]. All these solvers have demonstrated several degrees of success on a variety of ILP problems [20–24], but they still struggle when faced with particularly hard problems such as those within the MIPLIB 2010 library [25].

In this paper, we present a novel and general purpose *non-algorithmic* approach to the solution of ILPs based on the *memcomputing paradigm* previously introduced by two of us (FLT and MD) [26]. This new approach, which stands for computing *with* and *in* memory [27], cannot be classified as stochastic search, since it does not use a probabilistic scheme, nor a trial and error strategy. In addition, memcomputing does not employ educated guesses and known structures of the problem to define a set of instructions for a program to find solutions to the problem at hand. In other words, the memcomputing approach is not algorithmic [28].

On the contrary, a given problem is embedded into an electronic circuit (a possible realization of Memcomputing Machines (MM) [26, 28, 29]) whose time evolution ultimately relaxes to a steady state (equilibrium) that expresses the solution of the original problem [28, 29]). If these circuits are properly designed to satisfy several mathematical properties (see [28, 29]), they efficiently converge to the solution of the given problem, and chaos or periodic orbits can be avoided [30, 31]. However, for the case of optimization problems, the method does not provide proof of optimality for a given solution, nor does it detect the infeasibility of a problem.

In order to approach ILP we have first designed novel MMs based on the concept of *self-organizing algebraic gates* (SOAGs) we introduce in Section III. SOAGs are

* email: ftraversa@memcpu.com
† email: diventra@physics.ucsd.edu

the building blocks of the MM to solve ILP and are designed so that their dynamics self-organize towards the equilibrium that represents the solution satisfying the constraints of the ILP.

We have used this approach to find solutions for a selection of hard benchmark problems from the MIPLIB 2010 library [25]. In Section II we introduce the basic nomenclature for ILP. In Section III we discuss the memcomputing approach for this class of problems, and in Section IV we provide and discuss numerical results by simulating the corresponding MMs to solve several open problems from the MIPLIB 2010 library. We then compare our performances against a renowned commercial solver (Gurobi) and show the efficiency of our approach, which has been able to find within minutes feasible solutions for one of these hard problems (f2000 from MIPLIB 2010) whose feasibility, to the best of our knowledge, has remained unknown for the past eight years.

II. INTEGER PROGRAMMING BASICS

Let us consider a restricted version of ILP for which we have only binary variables. In this case, the problem can be formalized as follows:

$$\min_{\{x_j\}} \sum_j f_j x_j \tag{1a}$$

$$A_{eq} x = b_{eq} \tag{1b}$$

$$A_{ineq} x \leq b_{ineq} \tag{1c}$$

$$x_j \in \mathbb{Z}_2 \text{ for each } j \tag{1d}$$

where $x = \{x_1, ..., x_n\}$ with $x_j \in \mathbb{Z}_2$ for any $j = 1, ..., n$, $f_j \in \mathbb{R}$, $A_{eq} \in \mathbb{R}^{m_{eq} \times n}$, $b_{eq} \in \mathbb{R}^{m_{eq}}$, $A_{ineq} \in \mathbb{R}^{m_{ineq} \times n}$ and $b_{ineq} \in \mathbb{R}^{m_{ineq}}$ with m_{eq} and $m_{ineq} \in \mathbb{N}$. This is also known as *0-1 linear programming* and it is one of the Karp's 21 NP-complete problems [32].

A solution $\bar{x}$ of the ILP (1) is an assignment to x such that all constraints (1b)–(1d) are satisfied. The solution $\bar{x}$ is said to be sub-optimal if $\sum_j f_j \bar{x}_j \geq \min_{\{x\}} \sum_j f_j x_j$ where x satisfies (1b)–(1d). We also define the *objective*, O, of the problem (1) as $O = \min_{\{x\}} \sum_j f_j x_j$.

A lower bound for the ILP objective can be calculated efficiently in most cases by solving the relaxation problem obtained by replacing the constraint (1d) with

$$x_j \in [0, 1] \text{ for each } j. \tag{1d'}$$

It is easy to prove that the objective O_{LP} of the linear programming problem (1a)–(1c), (1d') satisfies $O_{LP} \leq O$.

Finally, given a lower bound O_{lb} of (1) a gap from optimality can be defined. Some commercial solvers estimate this gap as

$$\text{gap} = \frac{O_{best} - O_{lb}}{O_{best}} \tag{2}$$

where O_{best} is the best objective found so far while O_{lb} is the best lower bound to the objective found so far.

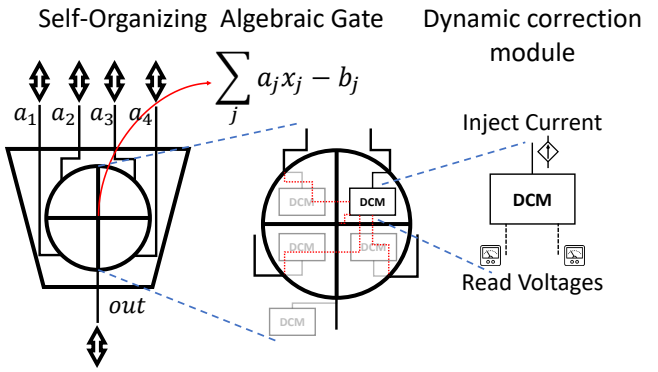

FIG. 1. Sketch of a Self-Organizing Algebraic Gate. All terminals allow a superposition of incoming and outgoing signals from the surrounding circuit. The central unit processes the signals in order to satisfy a liner algebraic relation consistent with the requirement of the "out" terminal. The self-organization is enforced by the Dynamic Correction Modules that read voltages from all terminals and inject a current to the appropriate terminal as long as the algebraic relation is not satisfied.

Therefore in many cases O_{LP} can be used as a lower bound for the gap if no better lower bound is available.

III. MEMCOMPUTING STRATEGY

The memcomputing approach to ILP problems is based on the concept of Self-Organizing Algebraic Gates (SOAGs). SOAG is a novel circuit design developed at MemComputing, Inc. [33] by one of the authors (FT). It is inspired by the previous work on Self-Organizing Logic Gates (SOLGs) [29, 34]. Both SOLGs and SOAGs are building blocks for practical realizations of Universal Memcomputing Machines (UMM) [26, 28, 35], in particular their *digital* (hence scalable) sub-set: digital memcomputing machines (DMMs) [29].

The main properties of SOLGs have been recently investigated and it has been proved that a proper design leads to Self-Organizing Logic Circuits (SOLCs) that demonstrate long-range order and topological robustness [36, 37]. Moreover, SOLCs can be designed in such a way that persistent chaotic and oscillatory behavior can be avoided [30, 31]. SOLCs have also been proved to be very efficient in a variety of combinatorial optimization problems such as maximum satisfiability (MAXSAT) [38, 39], quadratic unconstrained binary optimization (QUBO), spin-glasses [40], and pre-training of deep-belief networks [41].

SOLCs can be classified as *digital* realizations of UMMs because they accept inputs and return outputs that are digital in nature. Input and output are related to the circuit realization by associating logical 0s or 1s to voltages that are below or above a threshold, respectively. In this way, the required precision in writing inputs and reading outputs is *finite* and *independent* of the size of

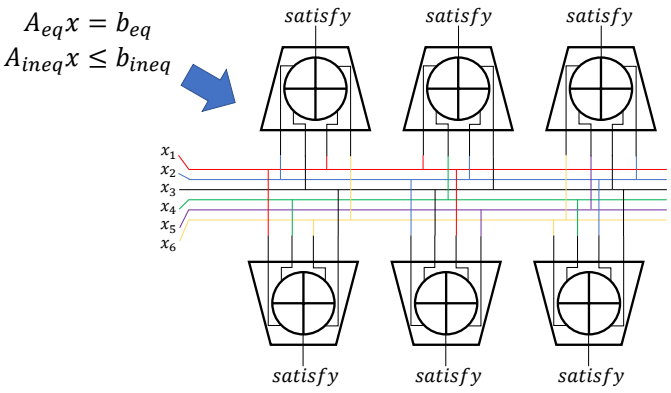

FIG. 2. Sketch of a Self-Organizing Algebraic Circuit (SOAC). SOAGs are connected together in an architecture that directly maps the ILP into the SOAC.

the problem at hand. However, the transition function of these machines (namely the function that maps input to output) is physical (analog) and takes full advantage of the *collective state* of the system to process information [29, 35, 42]. We reiterate though that, despite the physical nature of the transition function, DMMs can easily *scale* because they *do not* require precision that increases with the size of the problem. Rather, they can handle, ideally, unbounded problem sizes [39].

SOAGs share the same principles and scalability advantages of SOLGs but their circuit is designed to self-organize toward an *algebraic relation* rather than a boolean relation as for SOLGs. In this work, the SOAGs have been designed to satisfy linear relations between boolean variables as a particular case of algebraic relations (see Fig. 1). Further extensions of this design will include mixed integer and continuous variables, as well as nonlinear algebraic relations.

By connecting together SOAGs, we then assemble a Self-Organizing Algebraic Circuit (SOAC), see Fig. 2. The SOAC collectively self-organizes in order to satisfy the relations embedded in the gates. In this way, it is trivial to embed the problem (1) directly into the SOAC. Each one of the equations in (1b) and (1c) is mapped directly into a SOAG, while (1a) can be easily reformulated as an extra linear inequality

$$\sum_j f_j x_j \leq \tilde{b}. \tag{1a'}$$

where $\tilde{b}$ is an extra parameter that can be dynamically changed in the circuit in order to find solutions of increasing quality, each time closer to the global optimum. Finally, (1d) is naturally embedded in the circuit since inputs and outputs are digital.

Like the SOLCs, the ultimate *physical* electrical circuit representing a SOAC contains active and passive elements, with and without memory (internal state variables) [28, 29]. The corresponding electrical circuit can be built with available complementary metaloxide semiconductor (CMOS) technology. However, since its components are *non-quantum*, the ordinary differential equations describing it can be efficiently simulated on our modern computers. These equations are of the type

$$\dot{y} = F(y), \quad y(t=0) = y_0, \tag{3}$$

with y a vector describing all voltages/currents and internal state variables of the system, and F the flow vector field describing its dynamics [28, 29]. These nonlinear differential equations are then integrated numerically in time from a given (random) initial condition $y(t=0) = y_0$ up to a time out (TO) we set at the outset. Finally, the voltages of the state $y(t = \text{TO})$ represent the solution $\bar{x}$ of the ILP (1) at hand. Below, we present the results of these numerical simulations.

IV. NUMERICAL RESULTS

We consider as benchmark instances, problems from the MIPLIB 2010 library [25]. In particular, we consider the class of *open problems*, and we focus on the 0-1 programming ones only. This represents a set of 24 benchmark problems. The open problems are classified as problems for which either the optimal solution has never been found, or proved to be optimal, or, in some cases, the feasibility of the problem is unknown. This benchmark is a unique collection of problems from several industries or competitions for solvers. It represents a standard benchmark that developers use to test their solvers.

Parameter Name	Parameter Value
TimeLimit	3380
Presolve	2
Method	3
MIPGapAbs	0
MIPGap	0

TABLE I. Gurobi 8.0 parameters used in the tests. The same values for Gurobi solver parameters were used across all models. The *TimeLimit* parameter was set to 3380 seconds to allow a 220 second buffer for cloud server booting and shutdown while staying within one hour of total machine time. The *Presolve* parameter was set to 2 (maximum) in expectation that, more often than not, any additional presolve time would be offset by finding improved solutions earlier, given the known difficulty of the models. The *Method* parameter was set to 3 (concurrent) to allow maximum use of the 16 threads at the root node of the model in addition to using all 16 threads by default beyond the root node. The *MIPGapAbs* and *MIPGap* parameters were both set to 0 to ensure the Gurobi solver would not terminate at a suboptimal solution prior to the time limit.

In order to compare the performance of the MemComputing ILP solver (we refer to it as "MemCPU"), we have run these problems also using the Gurobi 8.0

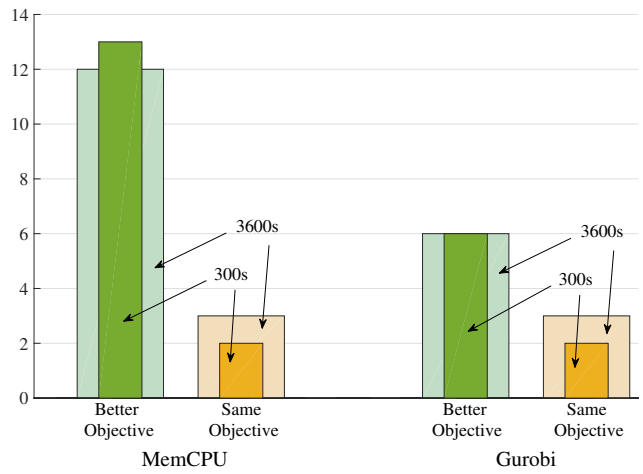

FIG. 3. Histogram of better objectives found by MemCPU or Gurobi 8.0 from Table II. The problems for which both solvers did not find any feasible solution have been excluded.

solver [7]. Gurobi is a renowned commercial solver for mixed-integer programming (MIP) used worldwide. In most cases it is employed as a reference because of its high-quality performance. Gurobi is a complex agglomerate of algorithms and heuristics to improve the time to, and quality of the solution of mixed-integer programming problems [43]. The main algorithm implemented in Gurobi is the branch-and-bound [44]. However, the latter is boosted by pre-processing, including variable pre-solving, cutting planes, and heuristics. In addition, Gurobi employs sophisticated algorithms and heuristics in order to further accelerate the branch-and-bound procedure [43]. The result is a collection of state-of-the-art algorithms, solution strategies and optimization toward the solution of MIP problems.

We stress here once more the major difference between our MemCPU solver and Gurobi. The former solves *differential equations* of a *physical system* that represents the original ILP problem. Gurobi, instead, is a sophisticated but still traditional (combinatorial) *algorithmic* approach. In other words, the memcomputing approach *first* transforms the original optimization problem into a Physics problem, and *then* simulates the dynamics of such a physical system, while maintaining the digital structure of inputs and outputs [28].

We have compared MemCPU versus Gurobi 8.0 in two different regimes: 300 seconds and 3380 seconds total running time. The Gurobi Solver was run on the Gurobi Cloud using an Amazon Web Services c4.4xlarge instance having 16 CPU cores and 30 GB RAM and the settings are described in Table I

The MemComputing, Inc. solver, MemCPU, has been implemented in interpreted MATLAB and run on an Intel Xeon 6148 with 192 GB RAM using only up to 10 cores at a time. However, the multi-core processing has only been used to run up to 10 identical versions of the solver (*replicas*) in parallel using `parpool`. The `parpool` function guarantees that each replica does not use multithreading. Therefore, each replica used exactly one core.

Each problem has been processed by MemCPU generating a SOAC representing the ILP problem without any pre-processing or variable pre-solving. Since MemCPU simulates an electronic circuit, there are physical parameters to be set that accelerate the self-organizations of the circuit depending on the problem at hand. Therefore, these parameters needed to be tuned [45]. The replicas have been used to run MemCPU with different initial conditions and different parameter sets for tuning purposes. The best outcome from the replicas has been selected as output.

In Table II the objectives after 300s and 3380s runs are reported for both MemCPU and Gurobi 8.0 using the aforementioned Gurobi parameter set. In Fig. 3 there is a direct comparison between MemCPU vs Gurobi by counting the number of problems for which each solver found solutions with better objective function values than the other. From the overall comparison we can see that the direct approach of MemCPU to solving ILP resulted in finding solutions with better objective function values on more than twice as many problems.

In the following subsections we discuss in detail some relevant results and characteristics of the MemCPU solver.

1. *The* `f2000` *Problem*

One very interesting result that deserves to be discussed separately is related to the outcome of MemCPU for the `f2000` problem of the MIPLIB 2010 library. The `f2000` problem belongs to the class of hard random problems [44] and was selected from the pseudo-boolean competition 2010 [46], that was a special event of the satisfiability (SAT) 2010 conference. Since then, `f2000` has been part of the next editions of the competition and also part of the open problems of the MIPLIB 2010 library [25].

Despite many groups from both SAT and MIP communities having tried, using both complete and incomplete solvers, to find feasible solutions for this problem during the past eight years, to the best of our knowledge, no one has been able to find a feasible solution yet. Currently the feasibility of `f2000` is classified as “unknown” by MIPLIB [25].

Running this problem using the MemCPU solver, we already found, within 60s, the first feasible solution to the problem, and for longer run times more solutions with objectives of increasing quality (see also Sec. IV 4). This result is then representative of both the uniqueness and power of the memcomputing approach.

	MemCPU		Gurobi 8.0	
File Name	300s	3380s	300s	3380s
bab1	-197710.06	-202800.20	-218764.89	-218764.89
bab3	TO	TO	-654569.46	-656193.04
circ10-3	362.00	312.00	TO	386.00
datt256	TO	TO	TO	TO
ds-big	29780.20	6902.00	762.93	762.93
ex1010-pi	237.00	237.00	240.00	238.00
f2000	1950.00	1846.00	TO	TO
ivu06-big	TO	349.02	9416.00	159.96
methanosarcina	2734.00	2731.00	2756.00	2737.00
neos-952987	TO	TO	TO	TO
ns1853823	144000.00	84000.00	284000.00	124000.00
ns894236	17.00	17.00	17.00	17.00
ns894786	13.00	13.00	14.00	13.00
ns903616	20.00	19.00	20.00	19.00
pb-simp-nonunif	87.00	71.00	75.00	42.00
ramos3	186.00	186.00	252.00	244.00
rmine14	-4208.27	-4223.09	-194.06	-4283.04
rmine21	-9340.90	-10392.05	TO	-214.18
rmine25	-13295.47	-15037.29	TO	-185.33
sts405	340.00	340.00	342.00	342.00
sts729	617.00	617.00	650.00	648.00
t1717	206003.00	192840.00	201342.00	201342.00
t1722	126537.00	119071.00	129822.00	119764.00
zib01	TO	TO	TO	TO

TABLE II. Objectives found after running MemCPU and Gurobi 8.0 for 300 seconds and 3380 seconds. Objectives are in arbitrary units and TO = Time Out. The problems are from the class "open" of MIPLIB 2010 library [25] restricted to the 0-1 programming only.

2. *Deep Diving Objectives*

Looking closer at Table II, we can see that for many problems, irrespective of their size and structure, MemCPU found very quickly much better objectives. This shows that, for these problems, solutions with objectives much closer to the global minimum are strong attractors for the SOAC. For some of them, the convergence was so quick that Gurobi did not find in one hour what MemCPU found in five minutes, possibly demonstrating the capacity for orders of magnitude speed-up of the memcomputing approach versus the traditional algorithmic one.

For some of the problems (e.g., `ramos3`, `ns1853823`, `ex1010-pi`), it is also possible that MemCPU may have found the global optimum since no refinement in the objective was found after minutes of run time. However, since MemCPU does not provide proof of optimality, we could not prove mathematically the validity of this statement.

Finally, it is worth mentioning that for the problems for which Gurobi performed better than MemCPU, from its log file (and also comparing the results for 300s and 3380s runs) it is clear that Gurobi's pre-processing was particularly effective at simplifying the problem which may have contributed to it finding a very good initial feasible solution. Instead, as we have already discussed, MemCPU does not include any pre-processing and it starts from random initial conditions. In addition, for these problems, both the MemCPU and Gurobi parameter tuning may not necessarily be optimal. At the moment, MemComputing, Inc. is working on an automatic routine to predict and fine-tune the MemCPU parameters [45]. Furthermore, since the design of SOAGs is not unique, further improvements that may accelerate the simulations and provide better solutions may be possible. Work along these lines is underway.

3. *Scaling with Problem Size*

The `rmine` benchmark is a series of problems that model the open pit mining problem [25]. This problem is industrially relevant and has been heavily studied by the MIPLIB organizers themselves in the past years [47]. Moreover, the MIPLIB 2010 library contains 5 instances with increasing numbers of variables depending on the refinements used to represent the problem. Therefore, this particular problem provides a benchmark for studying the scaling properties of both solvers.

We have then considered runs of 3380 seconds to assess the scaling properties for this benchmark. However, we include in the analysis only 4 out of the 5 instances because the smallest one (`rmine6`) belongs to the "easy"

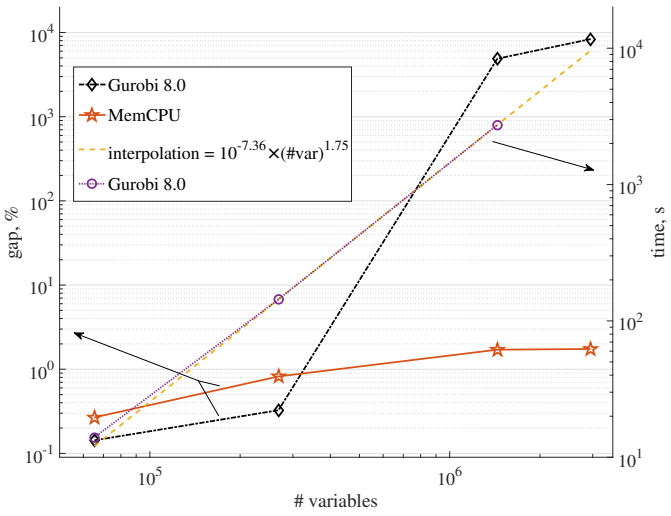

FIG. 4. Left y-axis: The gap defined in Eq. (2) for the `rmine` benchmark for both MemCpu and Gurobi 8.0. Right y-axis: Gurobi 8.0 pre-processing time.

category. Therefore, it is exactly solved in less than 3380s.

In Fig. 4 we report the gap defined in Eq. (2) for both Gurobi 8.0 and MemCPU after 3380s run. Gurobi converges very fast for the small instances to objectives smaller than 1%. However, by increasing the number of variables (namely, by increasing problem refinement), Gurobi was unable to complete the root simplex/barrier within the time limit and, consequently, showed much worse performance, providing solutions with objectives on the order of tens of thousands of %. On the other hand, MemCPU, even though, for smaller instances, had a slightly slower convergence to find solutions, it maintained the scaling also at high numbers of variables, providing a solution at about 1% gap for very large instances as well.

In order to understand better the reason for this difference between MemCPU and Gurobi, from the log file of the latter we found that the pre-processing time for those instances grew almost quadratically as shown in Fig. 4. We acknowledge that this could be largely a consequence of having set the *Presolve* parameter to its maximum value. However, the pre-processing is only the starting point of the solution process used by Gurobi that for these problems can simply grow unbounded. This is easily seen, for example in the `rmine21` problem, for which Gurobi was able to finish the pre-processing in a reasonable amount of time, but then spent 39 minutes on the root simplex/barrier, and then no refinement of the gap was made in the remaining time by the branch-and-bound process.

4. Data Availability

In order for the interested reader to check all the results discussed in the previous sections, and confirm the validity of what we have reported in this paper, we have made available all solution files on MemComputing, Inc. webpage: `http://memcpu.com/downloads`. The files are in the .sol format, and each file name corresponds to the name of the problem that can be downloaded from the MIPLIB 2010 library [25].

By using these data files the reader can use any solver for ILP (for instance Gurobi), and verify that all the objectives found by MemCPU and reported in Table II are indeed feasible solutions.

V. CONCLUSIONS

In summary, we have shown how to employ *digital* (hence scalable) memcomputing machines to tackle the important problem class of integer linear programming problems. We have proposed a new set of self-organizing gates that self-organize to satisfy algebraic relations. When assembled together, these gates form a self-organizing circuit specifically designed to solve a given ILP problem.

We have then simulated the corresponding equations of motion of these circuits to find solutions to a variety of benchmark ILP problems as reported in the MIPLIB 2010 library. We have compared our results with a well-known commercial solver (Gurobi). Our solver is extremely efficient in finding very good objectives for these problems.

In particular, we have found within minutes feasible solutions for the f2000 ILP problem (of MIPLIB 2010) whose feasibility, to the best of our knowledge, has remained unknown for the past eight years. We have also shown that our approach maintains a high quality of solutions with increasing size of the problem.

Since our solver has been implemented using interpreted MATLAB, there is plenty of room to speed up the reported calculations. In addition, since memcomputing machines employ non-quantum systems, they can be easily implemented in hardware using standard electronic components, thus offering a realistic path to real-time computing for these, and other, important combinatorial optimization problems.

[1] Schrijver. *Theory of Linear Integer Programming* (John Wiley & Sons, 1998).
[2] Gnu linear programming kit, version 4.32. URL `http://www.gnu.org/software/glpk/glpk.html`.
[3] Forrest, J. *et al.* Coin-or/cbc: Version 2.9.9 (2018).
[4] Gleixner, A. *et al.* The SCIP Optimization Suite 6.0. ZIB-Report 18-26, Zuse Institute Berlin (2018). URL `http://nbn-resolving.de/urn:nbn:de:`

0297-zib-69361.
[5] Ralphs, T. *et al.* Coin-or/symphony: Version 5.6.16 (2017).
[6] Berkelaar, M., Eikland, K. & Notebaert, P. lp_solve 5.5, open source (mixed-integer) linear programming system. Software (2004). URL http://lpsolve.sourceforge.net/5.5/.
[7] URL http://www.gurobi.com.
[8] URL https://www.ibm.com/analytics/cplex-optimizer.
[9] URL http://www.fico.com/en/products/fico-xpress-optimization.
[10] URL http://mathworks.com.
[11] Applegate, D., Bixby, R., Chvatal, V. & Cook, W. Concorde tsp solver (2006).
[12] Floudas, C. A. & Lin, X. Mixed integer linear programming in process scheduling: Modeling, algorithms, and applications. *Annals of Operations Research* **139**, 131–162 (2005).
[13] Boston, K. & Bettinger, P. An analysis of monte carlo integer programming, simulated annealing, and tabu search heuristics for solving spatial harvest scheduling problems. *Forest Science* **45**, 292–301 (1999).
[14] Sørensen, M. & Stidsen, T. R. Hybridizing integer programming and metaheuristics for solving high school timetabling. In *Proceedings of the 10th international conference of the practice and theory of automated timetabling*, 557–560 (2014).
[15] Danna, E., Rothberg, E. & Pape, C. L. Exploring relaxation induced neighborhoods to improve MIP solutions. *Mathematical Programming* **102**, 71–90 (2004).
[16] Glover, F. HEURISTICS FOR INTEGER PROGRAMMING USING SURROGATE CONSTRAINTS. *Decision Sciences* **8**, 156–166 (1977).
[17] Barnhart, C., Johnson, E. L., Nemhauser, G. L., Savelsbergh, M. W. & Vance, P. H. Branch-and-price: Column generation for solving huge integer programs. *Operations research* **46**, 316–329 (1998).
[18] Benders, J. F. Partitioning procedures for solving mixed-variables programming problems. *Numerische mathematik* **4**, 238–252 (1962).
[19] Hooker, J. N. & Ottosson, G. Logic-based benders decomposition. *Mathematical Programming* **96**, 33–60 (2003).
[20] Abara, J. Applying integer linear programming to the fleet assignment problem. *Interfaces* **19**, 20–28 (1989).
[21] Kroon, L. *et al.* The new dutch timetable: The or revolution. *Interfaces* **39**, 6–17 (2009).
[22] Stahlbock, R. & Voß, S. Operations research at container terminals: a literature update. *OR spectrum* **30**, 1–52 (2008).
[23] Melo, M. T., Nickel, S. & Saldanha-Da-Gama, F. Facility location and supply chain management–a review. *European journal of operational research* **196**, 401–412 (2009).
[24] Bard, J. F., Binici, C. *et al.* Staff scheduling at the united states postal service. *Computers & Operations Research* **30**, 745–771 (2003).
[25] URL http://miplib.zib.de.
[26] Traversa, F. L. & Di Ventra, M. Universal memcomputing machines. *IEEE Trans. Neural Netw. Learn. Syst.* **26**, 2702 (2015).
[27] Di Ventra, M. & Pershin, Y. V. The parallel approach. *Nature Physics* **9**, 200–202 (2013).
[28] Di Ventra, M. & Traversa, F. L. Perspective: Memcomputing: Leveraging memory and physics to compute efficiently. *Journal of Applied Physics* **123**, 180901 (2018).
[29] Traversa, F. L. & Di Ventra, M. Polynomial-time solution of prime factorization and np-complete problems with digital memcomputing machines. *Chaos: An Interdisciplinary Journal of Nonlinear Science* **27**, 023107 (2017).
[30] Di Ventra, M. & Traversa, F. L. Absence of chaos in digital memcomputing machines with solutions. *Phys. Lett. A* **381**, 3255 (2017).
[31] Di Ventra, M. & Traversa, F. L. Absence of periodic orbits in digital memcomputing machines with solutions. *Chaos: An Interdisciplinary Journal of Nonlinear Science* **27**, 101101 (2017).
[32] Garey, M. R. & Johnson, D. S. *Computers and Intractability; A Guide to the Theory of NP-Completeness* (W. H. Freeman & Co., New York, NY, USA, 1990).
[33] URL www.memcpu.com.
[34] Di Ventra, M. & Traversa, F. L. Self-organizing logic gates and circuits and complex problem solving with self-organizing logic circuits, US patent application No. 15/557,641, US patent No. 9,911,080 (2018).
[35] Traversa, F. L., Ramella, C., Bonani, F. & Di Ventra, M. Memcomputing NP-complete problems in polynomial time using polynomial resources and collective states. *Science Advances* **1**, e1500031 (2015).
[36] Di Ventra, M., Traversa, F. L. & Ovchinnikov, I. V. Topological field theory and computing with instantons. *Ann. Phys. (Berlin)* 1700123 (2017).
[37] Bearden, S. R., Manukian, H., Traversa, F. L. & Di Ventra, M. Instantons in self-organizing logic gates. *Physical Review Applied* **9** (2018).
[38] Traversa, F. L., Cicotti, P., Sheldon, F. & Di Ventra, M. Evidence of exponential speed-up in the solution of hard optimization problems. *Complexity* **2018**, 1–13 (2018).
[39] Sheldon, F., Cicotti, P., Traversa, F. L. & Di Ventra, M. Stress-testing memcomputing on hard combinatorial optimization problems. *Preprint arXiv:1807.00107* (2018). http://arxiv.org/abs/1807.00107v1.
[40] Sheldon, F., Traversa, F. L. & Di Ventra, M. Taming a non-convex landscape with long-range order. *In preparation* .
[41] Manukian, H., Traversa, F. L. & Di Ventra, M. Accelerating deep learning with memcomputing. *arXiv:1801.00512* (2018).
[42] Traversa, F. L. Collective Computing. *In preparation* (2018).
[43] URL http://www.gurobi.com/resources/getting-started/mip-basics.
[44] Arora, S. & Barak, B. *Computational Complexity: A Modern Approach* (Cambridge University Press, 2009).
[45] Foertsch, J., Qian, Z. & Traversa, F. L. Tuning and prediction of optimal parameters for algorithm configuration. *In preparation* (2018).
[46] URL http://www.cril.univ-artois.fr/PB10.
[47] Shinano, Y. *et al.* Solving open MIP instances with ParaSCIP on supercomputers using up to 80,000 cores. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (IEEE, 2016).