Solving Large Numerical Optimization Problems in HPC with Python

Antonio Gómez-Iglesias
Texas Advanced Computing Center
University of Texas at Austin
Austin, TX
agomez@tacc.utexas.edu

ABSTRACT

Numerical optimization is a complex problem in which many different algorithms can be used. Distributed metaheuristics have received attention but they normally focus on small problems. Many large scientific problems can take advantage of these techniques to find optimal solutions for the problems. However, solving large scientific problems presents specific issues that traditional implementations of metaheuristics do not tackle. This research presents a large parallel optimization solver that uses Python to follow a generic model that can be easily extended with new algorithms. It also makes extensive use of NumPy for an efficient utilization of the computational resources and MPI4py for communication in HPC environments. The presented model has proven to be an excellent approach for solving very large problems in an efficient manner while using the computational resources in different HPC environments adequately.

1. INTRODUCTION

There are many different approaches for solving numerical optimization problems in large and distributed environments. Normally, metaheuristics can be easily implemented in parallel by using master-slave models, island models or any other type of collaboration among the processes involved in the computation. However, the interest of these algorithms is normally focused on solving relatively small problems. Even though the number of constraints is sometimes large, the computational requirements of the problems being solved is very low, so that many instances of the problem can be evaluated in a very short time. Oftentimes the implementation of a parallel algorithm is not even justified considering the overall execution time of the optimization.

Scientific problems can be identified as large in the area of optimization based on different characteristics. For example, a problem can be considered large because the number of elements to optimize is big. Or because solving a single instance of the problem has large computational requirements, specifically in terms of wall time. Another possibility is that the solution space is too large to be explored with brute force approaches like parametric scans. This research focuses on problems that fit into any of these categories, as long as the evaluation of solutions is sequential or, at most, threaded. However, targeting problems that already need many cores in different nodes is foreseen and part of the future work proposed in this contribution.

When dealing with very large scientific problems, in which the evaluation of a single instance of the problem can take several minutes or even hours, the parallelization becomes more critical. An efficient load balancing of the tasks involved in the optimization must be taken into account. Also, researchers normally use different scientific codes as black boxes when working with these scientific problems. Those codes are likely developed by a different research group. Researchers modify the inputs used by code and analyze the results of that input. Each input is an instance of the problem, and the numerical optimization consists of finding the instance that best satisfies a set of requirements.

This paper introduces a model for efficiently solving these problems in HPC environments. The model follows a consumer-producer or master-slave approach, where the master runs an optimization algorithm, generating the instances that need to be solved by the slave processes. The implementation of the model uses Python and MPI4py for an efficient communication mechanism among the processes involved in the optimization process. Python provides several advantages for this problem when compared to other languages: short development time, availability of efficient numerical libraries, easy to understand code, portability, and most importantly, interoperability with other scientific programming languages. This paper introduces different algorithms previously implemented in the model and the optimization of a nuclear fusion device as an example of the type of scientific problem that can be efficiently solved with this model. Due to the distributed nature of the model, it has been named DISOP: Distributed Solver for Optimization Problems.

This contribution shows how the proposed model represents an optimal alternative to solving large scientific optimization problems in large HPC resources. The design has been created taking into account the special characteristics of these resources and it attempts to have the smallest possible impact on the hardware. The easy adaptation of DISOP to different scientific problems as well as the implementation of different optimization algorithms is also shown.

The rest of this paper is as follows: Section 2 introduces other contributions related to this paper. Section 3 describes the model of the proposed optimization system. Section 4

PyHPC2015 November 15-20, 2015, Austin, TX, USA
© 2015 ACM. ISBN 978-1-4503-4010-6/15/11…$15.00
DOI: 10.1145/2835857.2835864

details an optimization algorithm implemented in this model while Section 5 explains a scientific problem solved using the previous algorithm. Finally, Section 6 summarizes the main contributions of this paper and proposes future work.

2. RELATED WORK

This is not the first model to use Python for metaheuristics in distributed environments. For example, DEAP [1] is a framework that already contains many different algorithms. It uses SCOOP [2] for the communication mechanism. While DEAP and this research share several commonalities, the research presented in this paper specifically focuses on large-scale problems. Even though the model used in DISOP can be used for large problems, it has not been designed with those problems in mind. DEAP also follows the model of allowing the users to re-implement whatever they need in the framework. This is a critical idea in DISOP since large problems need much more tuning when being solved because of the restrictions imposed by the computational requirements.

A similar approach to DEAP is inspyred [3]. It is also designed as a black box for small problems. A popular framework, ParadisEO [4], also presents the same issue. ParadisEO provides a large number of options in terms of algorithms, and it supports MPI, however it targets problems that are different from the ones of interest in this contribution.

While this research does not specifically target constrained models, different techniques could be implemented to solve those problems. Metaheuristics do not typically represent a good alternative when solving these models, although some cases have been solved with the help of these techniques [5]. However, mixing these approaches with other mathematical methods, in the so-called matheuristics [6], can be an optimal solution for these problems [7, 8]. Other frameworks already tried this [9] although with commercial solvers.

A previous version of this model was focused on using Grid infrastructures to carry out the optimization process [10]. That version only focused on one optimization algorithm and it was not designed to be extended with more algorithms or to different scientific problems. Also, it was especially designed to run on grid environments. It did not use MPI for communication; instead, a file-based communication model was implemented. This approach would lead to file system problems in HPC environments, although it proved to be an efficient choice for the grid.

3. DISTRIBUTED MODEL

The distributed model follows a consumer-producer schema. The master (producer) process creates new possible solutions for the problem and adds them to a queue. It communicates with the consumers to send them the solutions that have not been evaluated (candidate solutions), and to retrieve the results once the candidate solution has been processed.

This distributed model implements point-to-point communication. The different consumers do not need to all be synchronized, since each evaluation is independent from the others. When a consumer requires input, it remains locked until that input is received. It then processes that input and produces an evaluation. This evaluation is sent to the master process, the consumer remaining locked until the evaluation is received by the master and this one sends another solution to the consumer. The master process does not remain locked waiting for requests from any other process. It simply checks the status of all the processes and performs the action required depending of the type of the requests that it receives.

Two different lists are used to keep solutions that have not been evaluated and for solutions that have already been calculated. While for small problems reevaluating solutions might not be an issue, when the computational requirements of the problems are high, calculating the same solution several times has a large impact. Therefore, every time a new solution is being generated, all the previous solutions are considered so that no duplicates are created. This is a task that may have a high complexity specifically when a large number of possible solutions have already been calculated, so that creating new ones is more challenging. Since the main process does not perform any computation, this task does not represent a bottleneck in the overall execution of the optimization.

Fig. 1 shows the two lists used in this model. The list with pending solutions is a queue that stores instances that have not been calculated. These solutions are sent to the processes computing the instances. These processes return an instance evaluated, with its fitness value. The master process retrieves this value and stores the instance in a priority list. This priority list is then used to create new instances of the problem.

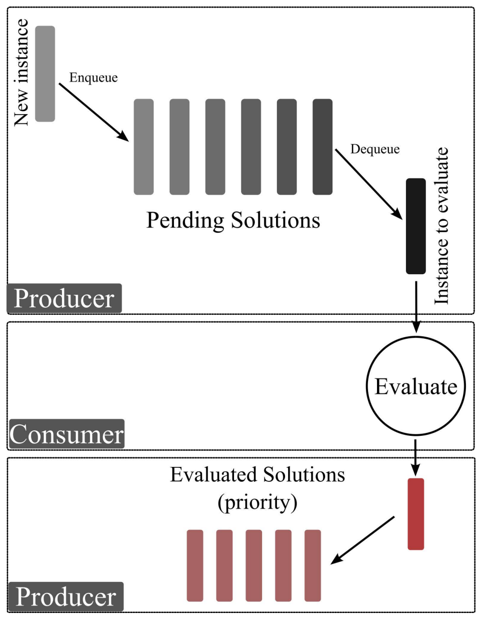

Figure 1: Producer-consumer model.

Initially the master or producer task will read an XML configuration file that describes the problem being optimized. Only this process needs this information, as it is only used to generate candidate solutions. Once this file has been read, the producer starts creating a set of candidate solutions based on the optimization algorithm being implemented and puts those solutions in the pending queue. In the meantime, the consumers are waiting for solutions to be sent to them. They create a request from the producer process for this input. This process will see the requests and send solutions from the pending queue to the other processes and periodically wait for those processes to send back the results. When a consumer has finished the evaluation of a solution, it sends the results to the producer by creating another request. The master task will detect this request, read the results, send another solution from the pending queue to the consumer process, and perform a set of actions based on the quality of the solution just evaluated. The actions will depend on the optimization algorithm being used. It will also add the evaluated solution to a sorted list of solutions. The optimization process will finish once a solution satisfying a given requirement or set of requirements has been found or after a predefined wall time.

3.1 Generic Implementation

The proposed model is independent of the problem being optimized. It has been designed so that it can be easily extended or modified to tackle different scientific problems. This contribution presents a nuclear fusion problem as an example of the type of scenario that can be tackled with DISOP. However, by simply changing the method responsible for evaluating an instance of this particular model, the focus would be completely different. The specification of the problem is independent of the model, as DISOP proposes an XML-based approach where all the parameters of the problem are specified, together with their upper and lower bounds. The minidom [1] module is used to manage XML files within DISOP.

Moreover, the entire system is also designed to work with different algorithms as well as communication models. There are currently three different optimization algorithms already implemented:

1. Distributed Asynchronous Bees algorithm (DAB). DAB [11] is optimization algorithm based on the bees foraging behavior [12]. It can be naturally adapted to a distributed model.
2. Ant Colony Optimization (ACO). ACO [13] is a very popular algorithm that can be also adapted to distributed environments following different approaches: individual evaluations of candidate solutions by remote computing elements or independent executions of the entire algorithm in parallel with some exchange of information among processes at different times.
3. Simulated Annealing (SA). SA [14] a distribution of the evaluation of the possible solutions and a centralized master process has been implemented.

These three algorithms are popular options in the metaheuristics field or a variation of a popular algorithm. Many more algorithms can be easily added as long as they follow a consumer-producer model. This is something that could be easily achieved as all it requires is the execution of the algorithm in one process and the evaluation of candidate solutions in different processes.

Also, two different types of problems are solved and distributed with the software:

1. The first one is the optimization of a nuclear fusion device. This problem includes a complex application workflow and presents many different parameters that can be optimized. Also, the overall wall time for this problem is long due to the complexity of the computations involved in the evaluation of a given configuration of a fusion device.
2. The second problem is a generic non-separable function. The function can be easily implemented. This is a smaller problem, very typical in competitions and many scientific areas. This problem shows the feasibility of DISOP for these problems.

[1]https://docs.python.org/2/library/xml.dom.minidom.html

These two problems are used as examples for other typical cases. The first example can be used as a template for very complex optimization problems that require different tools to evaluate a given problem. The second problem is a more typical case, where the complexity of the implementation is often low.

3.2 MPI Implementation

As previously introduced, the implementation uses MPI4py for the communication between processes. Since the model follows a consumer-producer schema, only point-to-point communication is required. All the data being exchanged consists of NumPy arrays. Even integers are transformed into a NumPy array of length 1. No complex data is used in the communication process. During the initial development of DISOP, Python objects as well as NumPy arrays were being used for communication. However, it quickly became clear that it was difficult to use both types and that it was slowing down the development, so it was decided to only use NumPy arrays.

Due to the limited availability of asynchronous methods in MPI4py at the time of developing DISOP, the resulting implementation is a bit intricate when compared to a possible C/C++ implementation using MPI. However, an almost fully asynchronous version of the framework has been created. In the consumer-producer model, consumers need to be synchronized, since they can not carry out any calculation without having anything to compute. However, the producer needs to be able to perform different operations independently from the consumers. Therefore, the main process is completely asynchronous and will only communicate with the other processes when they request it.

DISOP has been used in three different clusters with different MPI implementations [15–17] without issues. These three clusters offer different singularities and challenges, and this proves the interoperability and possibilities of DISOP as a framework that can be widely used.

3.3 Checkpointing

DISOP has been designed with an important checkpointing component on it. There are two main reasons for this:

1. To be able to deal for hardware issues. In HPC environments it is not uncommon to find hardware failures that lead to failures in the jobs.
2. To solve very long optimization problems. Many HPC centers have very strict limitations on the maximum wall time for a job. Once that time is reached, the job is killed. However, for the types of problems that this research is targeting, it is very common to have very long execution times.

The checkpointing mechanism allows the computation to be resumed at any time during the optimization process. The only calculations that will have to be performed again if the computation is restarted correspond to those candidate solutions being evaluated when the job was terminated. Solutions already evaluated are kept and used in the new optimization process. This model has made it possible to solve problems with an overall execution wall time of more than 1 month.

The only issue that the checkpointing mechanism presents is the overhead that it introduces. However the overhead is very low in the current consumer-producer model and for the type of problems that are being solved. The overall execution time for a single candidate solution can take long. In the meantime, while all the consumers are evaluating solutions, the producer model carries out the checkpointing mechanism. It also creates new candidate solutions and performs different computations depending on the optimization algorithm that is being used.

4. AN EXAMPLE

A first optimization algorithm implements a distributed version the ABC algorithm [12] to solve scientific problems on clusters. The pseudo-code of the proposed model, named Distributed Asynchronous Bees Algorithm (DAB) [11] can be seen in Alg. 1. This algorithm has been previously used in grid environments with optimal results.

Line 2 creates an initial population of candidate solutions. The scout bee generates these solutions. The same method is also implemented on line 28. The scout bee generates a random solution by using Eq. 1, where v_i is the value for parameter i, L_i is the lower bound of the parameter, and U_i is the upper bound. $rand$ is a function that returns a random number within the range $[L_i, U_i]$ using a normal distribution.

$$v_i = rand(L_i, U_i) \tag{1}$$

Line 4 retrieves a list S of all the slaves that are waiting for input. This list is then traversed and a candidate solution p in the list of solutions is sent to each element s in the list S (line 6).

If there are not candidate solutions in the list P, new solutions p need to be created before they can be sent to the remote processing elements. Lines 11 and 14 call the corresponding method for each type of bee to create new candidate solutions based on the characteristics of the bee (see Sec. 4.1).

Lines 12 and 15 insert the newly created solution p into the queue of candidate solutions P.

Line 16 checks if any of the consumers has finished the evaluation of a candidate solution. If that is the case, the evaluation of that solution is retrieved and analyzed. If the solution was good, the global best might be updated. Also, new solutions can be generated based on it depending on the evaluation or quality of the solution. An in-depth search can be created by introducing small modifications on the parameters of that solution. It could also be the last evaluation of a solution based on a previous good solution. In this case, if no improvement has been found, the solution is abandoned (lines 28 and 29).

4.1 Parameters of the Model

Algorithm 1: The DAB Algorithm Pseudocode

```
1  RC = 0;
2  $P \leftarrow$ InitilizePopulation;
3  while Stop criterion not reached do
       // send solutions
4      $S \leftarrow$ GetConsumersWaiting;
5      foreach $s \in S$ do
6          SendSolutionToSlave($p$,$s$);
7          if $P! = \emptyset$ then
8              continue;
9          while Sizeof(P)<MaxPopulationSize do
10             foreach $e \in Employed$ do
11                 $p \leftarrow$CreateNewSolution($e$);
12                 $P \leftarrow$InsertSolution($p$);
13             foreach $e \in Onlookers$ do
14                 $p \leftarrow$CreateNewSolution($e$);
15                 $P \leftarrow$InsertSolution($p$);
       // retrieve solutions
16     $S \leftarrow$ GetConsumersFinished;
17     foreach $s \in S$ do
18         $e, bee \leftarrow$GetEvaluation($s$);
19         PutSolutionOnFinished ($e$);
20         if isNewGlobalBest(e) then
21             UpdateGlobalBest($e$);
22         if BeeIsEmployed(b) then
23             if isSolutionBetterThanLocal(b,e) then
24                 UpdateLocalBest($b$,$e$);
25             else
26                 IncreaseIterationsBee($b$);
27             if IterationsNoImproveExceeded(b) then
                   // abandon solution
28                 $p \leftarrow$CreateScoutSolution;
29                 PutSolutionOnBee($b$,$p$);
```

All of the parameters of the model are passed using an INI file. While other algorithms implemented in DISOP might use different formats to configure the different options of the algorithm, it is recommended to use this format for future implementations. The ConfigParser [2] module is used to read these files.

1. Number of onlookers.
2. Number of employed.
3. Onlookers factor (σ): the modification factor used by an onlooker bee to calculate the new value of a parameter. New values are calculated by using Eq. 2, where v_i is the new value of parameter i, while x_i is the current value of the parameter.
$$v_i = x_i \pm \sigma \times x_i \tag{2}$$
4. Onlookers change probability: the probability that an onlooker bee of will change each of the parameters involved in the optimization.

[2]https://docs.python.org/2/library/configparser.html

5. Employed change probability: the probability that an employed bee will change each of the parameters involved in the optimization.

6. Abandoned iterations: the number of evaluations of a candidate solution without being improved before the solution is replaced by a new one.

7. Elite queue size: the number of elements in the list containing the best solutions found so far.

5. APPLICATION

The problem used in this paper is the optimization of the pressure profile of nuclear fusion reactors so that it increases the beta β of the plasma. This β is the ratio of the plasma pressure to the magnetic pressure (see Eq. 3, where n is the number density, k_B the Boltzmann constant, T the temperature, B the magnetic field and μ_0 the permeability). Achieving a high β is critical because the temperature increases with the pressure.

$$\beta = \frac{nk_BT}{(B^2/2\mu_0)} \tag{3}$$

However, in order to increase the plasma pressure, more expensive magnetic coils are needed. The cost of these coils is not considered in this particular problem.

In the past, different experiments have presented the optimization of the magnetic fields by improving the Fourier modes describing the plasma surface [11]. In any nuclear fusion device, it is important to attain the best magnetic configuration possible for confined plasma to avoid the transport of particles and heat. This configuration is produced by the force balance between the magnetic field produced by the fusion device and the pressure of the plasma. The optimization uses the previously described DAB algorithm.

5.1 Workflow Application

The tools required for running the optimization described above are shown in Fig. 2 and described below. These are well-known tools in the nuclear fusion community, and they are serial applications developed in Fortran. These tools are included in the optimization model to evaluate the solutions previously generated. These solutions use different values for the parameters specifying the pressure profile.

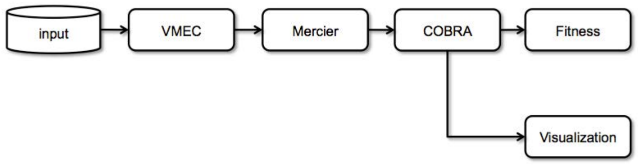

Figure 2: Workflow to measure the quality of the plasma confinement in a stellarator

Change the image to include the visualization

Additionally, there is a visualization component mostly developed in Python that allows the plasma confined inside the fusion device to be displayed, as shown in Fig. 3.

VMEC.

The Variational Moment Equilibrium Code [18] calculates the configuration of the magnetic surfaces in a stellarator.

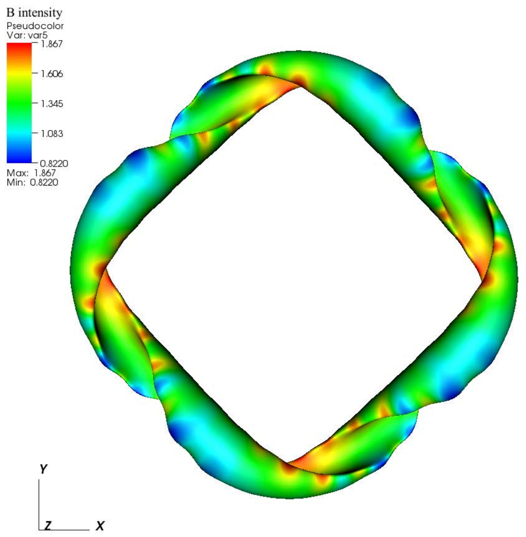

Figure 3: Overview of the plasma confined with the color representing the intensity of the magnetic field.

The solution is given in terms of its Fourier harmonic coefficients, as in Eq. 4, where ρ is the effective normalized radius of a particular point on a magnetic surface and θ and ϕ, are the poloidal and toroidal angles, which take values in the range $\{0...360\}$.

$$B(\rho,\theta,\phi) = \sum_{mn} B_{mn}(\rho)\cos(m\theta - n\phi) \tag{4}$$

This is the main component of the work flow. The time required for a candidate solution (configuration defining a stellarator) to converge into the final solution varies from few minutes to hours.

Mercier.

The Mercier stability criterion is implemented in the VMEC code that calculates this value for the effective radius 0 to 1 [19]. Since the area of interest is in the range $[0.8-1]$, only those values are taken into account. Stable Mercier configurations are those satisfying the criterion in that range. A configuration is considered non-valid if it is unstable Mercier within that area.

COBRA.

The Ballooning stability criterion is the next optimization function that will be considered in this problem. Ballooning stability criterion requires the use of the COBRA code after VMEC [20].

Beta Optimization.

VMEC calculates the value of β. This value needs to be extracted from the results files only when the candidate solution has converged and is also Mercier and Ballooning stable.

5.2 Results

Two different existing stellarators (TJ-II [21] and W7-X [22]) have been considered for this experiment. The goal in this problem is to find the best possible pressure profile for each stellarator. A profile defined by a polynomial of degree nine is used in this problem. Each evaluation of the stellarator can take from a few minutes to more than three hours. DISOP is started with a limited wall time of 48 hours, and after the 48 hours, another instance of DISOP is started using the last status of the previous execution. Overall, two weeks of total wall time were required for each of the solutions here presented using 256 cores.

Due to the nature of the framework, the extremely low communication required in the optimization process, and the type of problem being solved, the speedup attained is almost linear. Only at the beginning of the execution some delays are introduced since all the consumers are waiting for the main process to send them solutions to evaluate. After this, since each evaluation will take a different amount of time to finish, the communication between consumer and producer processes takes place at different times and there are not bottlenecks due to this. Given the master process does not perform any calculation, it diminishes the performance that can be achieved, especially when few cores are used in the computation.

The best configurations found are shown in Fig. 4 for TJ-II and Fig. 5 for W7-X. It is clear how the pressure profile is completely opposite for both machines. This shows the many different possibilities that exist for designing new stellarators and how the proposed method framework is able to solve different problems.

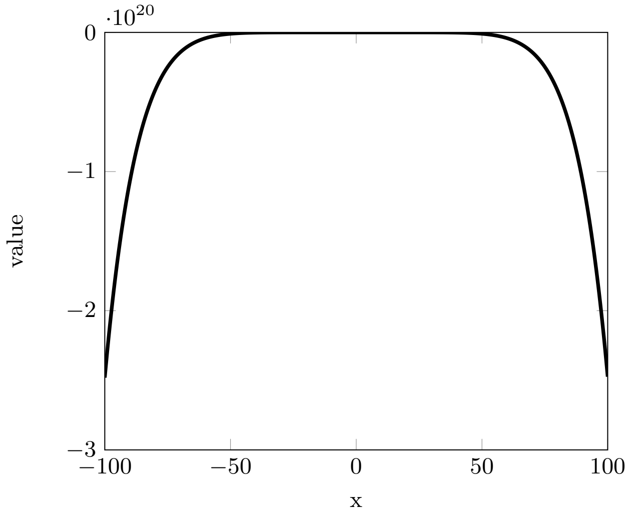

Figure 4: TJ-II Pressure profile.

6. CONCLUSIONS AND FUTURE WORK

This paper has presented a framework for solving large scientific optimization problems in HPC environments. The framework is fully implemented in Python and has been designed with the aim of being easily extensible and adaptable to many different problems and optimization algorithms.

The paper has introduced a large problem being solved with this framework using the ABC algorithm. The problem is complex in terms of solution space, required execution time for a single solution, and overall execution time. While many frameworks are not designed to efficiently solve these problems, DISOP allows users to tackle complex problems. The checkpointing mechanism provides an easy way to carry out very long optimization processes.

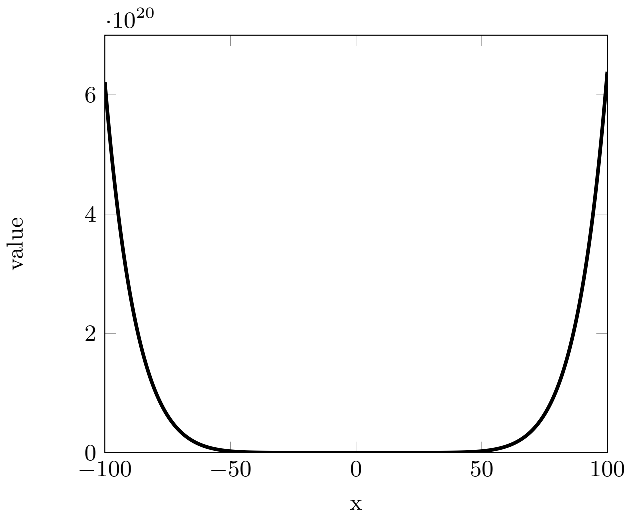

Figure 5: W7-X Pressure profile

The main component being considered as future work is the extension of the communication models in DISOP. Currently, only a consumer-producer model as the one depicted in Fig. 6 is being used. The extension will consist of allowing a fully decentralized model without a central component on it. This can be used for fully distributed algorithms or it will also allow to have different instances of the same or different algorithms working on different machines on the same problem.

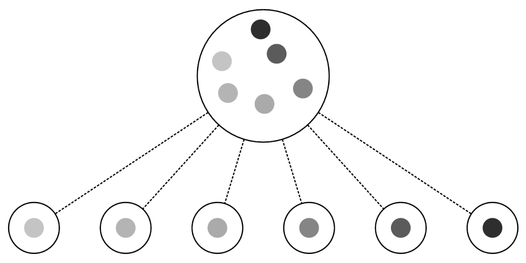

Figure 6: Consumer-producer model.

Also, as future work, the comparison of the results achieved with DISOP for other problems that have already being implemented [23] is foreseen.

Finally, as previously indicated, solving problems that already present a parallel implementation is expected in the future. Some changes will be required in DISOP to be able to tackle those problems.

7. ACKNOWLEDGMENTS

The author would like to thank Miguel Cárdenas-Montes and Francisco Castejón from CIEMAT, and Miguel A. Vega-Rodríguez, from the University of Extremadura, Spain, for their contribution in this research. Special thanks to Ashley

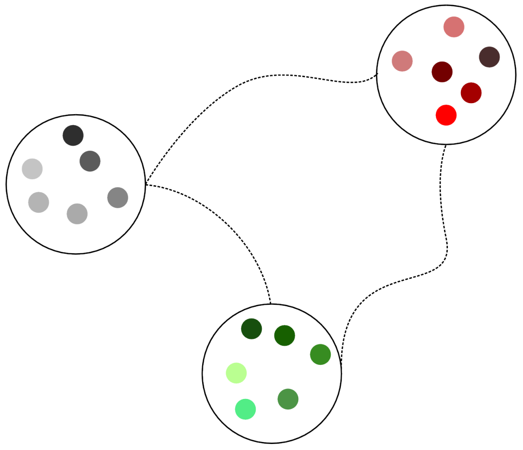

Figure 7: Fully distributed model.

Bucholz, from the Texas Advanced Computing Center, for her interest in this work.

8. REFERENCES

[1] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, jul 2012.

[2] Y. Hold-Geoffroy, O. Gagnon, and M. Parizeau, "Once you SCOOP, no need to fork," in *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, ser. XSEDE '14. New York, NY, USA: ACM, 2014, pp. 60:1–60:8. [Online]. Available: http://dx.doi.org/10.1145/2616498.2616565

[3] "inspyred: Bio-inspired algorithms in Python," http://aarongarrett.github.io/inspyred/, 2015, [Online; accessed 22-Sep-2015].

[4] S. Cahon, N. Melab, and E.-G. Talbi, "ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics," *Journal of Heuristics*, vol. 10, no. 3, pp. 357–380, May 2004. [Online]. Available: http://dx.doi.org/10.1023/B:HEUR.0000026900.92269.ec

[5] B. Gasbaoui and B. Allaoua, "Ant colony optimization applied on combinatorial problem for optimal power flow solution," 2009.

[6] V. Maniezzo, T. Sttzle, and S. Vo, *Matheuristics: Hybridizing Metaheuristics and Mathematical Programming*, 1st ed. Springer Publishing Company, Incorporated, 2009. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-1306-7

[7] A. Gómez-Iglesias, A. T. Ernst, and G. Singh, "Scalable multi swarm-based algorithms with lagrangian relaxation for constrained problems," in *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2013 / 11th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA-13 / 12th IEEE International Conference on Ubiquitous Computing and Communications, IUCC-2013, Melbourne, Australia, July 16-18, 2013.* IEEE Computer Society, 2013, pp. 1073–1080. [Online]. Available: http://dx.doi.org/10.1109/TrustCom.2013.241

[8] O. Brent, D. R. Thiruvady, A. Gomez-Iglesias, and R. Garcia-Flores, "A parallel lagrangian-aco heuristic for project scheduling," in *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2014, Beijing, China, July 6-11, 2014.* IEEE, 2014, pp. 2985–2991. [Online]. Available: http://dx.doi.org/10.1109/CEC.2014.6900504

[9] E. D. Dolan, J. J. Moré, and T. S. Munson, "Benchmarking optimization software with cops 3.0," in *MATHEMATICS AND COMPUTER SCIENCE DIVISION, ARGONNE NATIONAL LABORATORY*, 2004.

[10] A. Gómez-Iglesias, F. Castejón, and M. A. Vega-Rodríguez, "Distributed bees foraging-based algorithm for large-scale problems," in *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May 2011 - Workshop Proceedings.* IEEE, 2011, pp. 1950–1960. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2011.355

[11] A. Gómez-Iglesias, M. A. Vega-Rodríguez, and F. Castejón, "Distributed and asynchronous solver for large CPU intensive problems," *Appl. Soft Comput.*, vol. 13, no. 5, pp. 2547–2556, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.asoc.2012.11.031

[12] D. Karaboga and B. Basturk, "A powerful and efficient algorithm for numerical function optimization: Artificial bee colony (abc) algorithm," *J. of Global Optimization*, vol. 39, no. 3, pp. 459–471, Nov. 2007. [Online]. Available: http://dx.doi.org/10.1007/s10898-007-9149-x

[13] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 26, no. 1, pp. 29–41, Feb 1996. [Online]. Available: http://dx.doi.org/10.1109/3477.484436

[14] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *SCIENCE*, vol. 220, no. 4598, pp. 671–680, 1983. [Online]. Available: http://dx.doi.org/10.1126/science.220.4598.671

[15] "Stampede cluster," https://www.tacc.utexas.edu/stampede/, 2015, [Online; accessed 22-Sep-2015].

[16] "Bragg cluster," https://wiki.csiro.au/display/ASC/CSIRO+Accelerator+Cluster+-+Bragg, 2015, [Online; accessed 22-Sep-2015].

[17] "Euler cluster," http://rdgroups.ciemat.es/en_US/web/sci-track/euler, 2015, [Online; accessed 22-Sep-2015].

[18] S. P. Hirshman and G. H. Neilson, "External inductance of an axisymmetric plasma," *Physics of Fluids*, vol. 29, no. 3, pp. 790–793, 1986. [Online]. Available: http://dx.doi.org/10.1063/1.865934

[19] C. C. Hegna and N. Nakajima, "On the stability of mercier and ballooning modes in stellarator configurations," *Physics of Plasmas*, vol. 5, no. 1336, pp. 1336–1344, 1998. [Online]. Available: http://dx.doi.org/10.1063/1.872793

[20] R. Sanchez, S. P. Hirshman, J. C. Whitson, and A. S. Ware, “COBRA: an optimized code for fast analysis of ideal ballooning stability of three-dimensional magnetic equilibria,” *Journal of Computational Physics*, vol. 161, no. 2, pp. 576–588, 2000. [Online]. Available: http://dx.doi.org/10.1006/jcph.2000.6514

[21] C. Alejaldre et al., “First plasmas in the TJ-II flexible heliac,” *Plasma Physics and Controlled Fusion*, vol. 41, no. 3A, p. A539, 1999. [Online]. Available: http://dx.doi.org/10.1088/0741-3335/41/3A/047

[22] V. Erckmann, H. J. Hartfuss, M. Kick, H. Renner, J. Sapper, F. Schauer, E. Speth, F. Wesner, F. Wagner, M. Wanner, A. Weller, and H. Wobig, “The W7-X project: scientific basis and technical realization,” in *Fusion Engineering, 1997. 17th IEEE/NPSS Symposium*, vol. 1. San Diego, California: IEEE, Oct 1997, pp. 40–48. [Online]. Available: http://dx.doi.org/10.1109/FUSION.1997.685662

[23] M. Cárdenas-Montes, M. A. Vega-Rodríguez, J. J. Rodríguez-Vázquez, and A. Gómez-Iglesias, “A comparison exercise on parallel evaluation of rosenbrock function,” in *Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, Companion Material Proceedings*, J. L. J. Laredo, S. Silva, and A. I. Esparcia-Alcázar, Eds. ACM, 2015, pp. 1361–1362. [Online]. Available: http://doi.acm.org/10.1145/2739482.2764641