Proceedings of GO 2005, pp. 1 – 6.

A Distributed Global Optimisation Environment for the European Space Agency Internal Network

Dario Izzo[1] and Mihály Csaba Markót [1]

[1] *Advanced Concepts Team, European Space Agency, Keplerlaan 1, 2201 AZ Noordwijk, The Netherlands.*
{Dario.Izzo,Mihaly.Csaba.Markot}@esa.int

Abstract Global optimisation problems arise daily in almost all operational and managerial phases of a space mission. Large computing power is often required to solve these kind of problems, together with the development of algorithms tuned to the particular problem treated. In this paper a generic distributed computing environment built for the internal European Space Agency network but adaptable to generic networks is introduced and used to distribute different global optimisation techniques. Differential Evolution, Particle Swarm Optimisation and Monte Carlo Method have been distributed so far and tested upon different problems to show the functionality of the environment. Support for both simple and multi-objective optimisation has been implemented, and the possibility of implementing other global optimisation techniques and integrating them into one single global optimiser has been left open. The final aim is that of obtaining a distributed global multi-objective optimiser that is able to 'learn' and apply the best combination of the available solving strategies when tackling a generic "black-box" problem.

Keywords: Distributed computing, idle-processing, global optimisation, differential evolution, particle swarm optimisation, Monte Carlo methods.

1. The Distributed Computing Environment

Our distributed computing architecture follows the scheme of a generic server-client model [10]: it consists of a central computer (server) and a number of user computers (clients). The architecture is divided into three layers on both the server and the client side (Figure 1). This promotes the modular development of the whole system: for example, the computation layer of the clients (involving the optimisation solver modules) can be developed and maintained independently of the other client layers.

The main tasks of the server are the following: pre-processing the whole computing (optimisation) problem by disassembling it into subproblems; distributing sets of subproblems among the clients; and generating the final result of the computation by assembling solutions received from the clients.

On the opposite side, the client computers ask for subproblems, solve them and send back the results to the server. As with many current distributed applications (employing common user desktop machines), our approach is also based on the utilisation of the idle time of the clients. More precisely, a client only asks for subproblems when there is no user activity detected either on the mouse or on the keyboard. In our environment we hide the client computations behind a screen saver, similarly to the most widely-known distributed computing project, the SETI@home project (http://setiathome.ssl.berkeley.edu).

As shown in Figure 1, the basic functionalities of the individual layers are the same for both the client and the server. The uppermost layers are visible to the client users and for the server administrator, respectively. These layers contain the screen saver (client) and a server

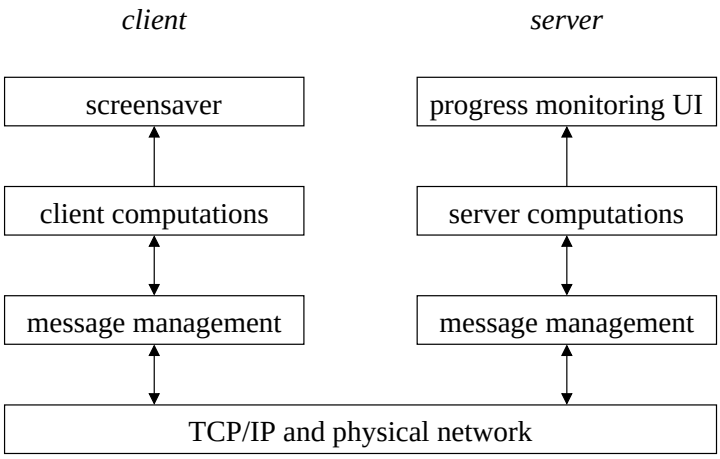

Figure 1. Architecture of the distributed environment.

progress-monitoring user interface (displaying the actual state of the computation process), on the server.

The top-level layers communicate with the real computation layers that are performing the numerical tasks. The computation layers are responsible for disassembling, distributing and assembling the whole computation task (server), and evaluating the subproblems (clients).

The lowermost, so-called message management layers maintain connection with the computation layers, and send and receive the problem and solution 'packages' between the client and the server. This service was implemented by network sockets using the Windows Sockets version 2 Application Programming Interface [12].

The environment was programmed in Visual C++, strongly utilising the advantages of the object-oriented language. The introduction of the detailed architecture is a subject of separate, forthcoming publications (due to its extent); here we restrict ourselves to giving a short overview on the key concepts and building blocks.

The environment provides several data storage classes to program the various solvers in an easy way. The basic data type is called SOL_PAIR; it is the representation of an $(x, f(x)) \subset \mathbb{R}^n \times \mathbb{R}^m$ pair. Both components are implemented as a variable size vector, which allows us to deal with single and multi-objective optimisation problems, constraint satisfaction problems (with no objective function), and virtually every kind of distributable (even not necessarily optimisation!) problems. For population-based solvers, it was particularly useful to have a POPULATION storage class, which is simply a set (list) of SOL_PAIRs. The basic storage types used during the client-server communication are called 'packages': the most important one is a base class called PS_PACKAGE, which is used to derive the specific problem and solution packages for the particular solvers. A package typically involves a data storage object (such as a POPULATION) together with instruction (server) and solution (client) information. All the above classes have member functions which transform the data to and from a stream of characters. The latter data type is used by the message management routines to transfer the data on the network.

The server computation layer is based on a C++ abstract class called SERVER. The particular servers (implementing various strategies to solve the whole problem) are derived from this class. On the other hand, each client computation layer contains the set of available solvers. The solvers are derived from an abstract SOLVER class. This means that the environment can be arbitrarily extended by adding server and solver classes. Moreover, since each problem package (sent out to a client) is always solved by one specified solver, various server strategies can be tested without changing the client application. It is very important that the solution packages should always be kept in a consistent state by the solver: if the computation is

interrupted by a user action, a fraction (but still useful part) of the whole solution is sent back to the server.

The optimisation problems are implemented as instances of an OPT_PROB class: in practice, this means that for every particular optimisation problem the user have to provide the following routines: function evaluation at a given point, random generation of a feasible point, feasibility checking of a given point (and its substitution with a feasible point in case of infeasibility), and a routine implementing a preference relation between every two feasible solutions.

2. The global optimisation algorithms

In the present version of the software, there are three different optimisation solvers available:

1. Monte–Carlo search (MC, [7]). In this method, the server requests the clients to create a certain number of independent random samples from the feasible search space (using uniform distribution in each variable) and sends the most promising solution back to the server. The server maintains a set of the best solutions received (Pareto optimality criteria are used in the case of multi-objective optimisation problems).

2. Differential Evolution (DE). This novel optimisation algorithm is based on updating each element of a set (population) of feasible solutions by using the difference of two other randomly selected population elements. The method is described in detail in [9]. In our environment the DE server updates a fixed-size main population, and each problem package consists of a request to evolve a randomly-selected subpopulation for a specified number of iterations. The main population is then updated by the returned population with respect to the preference relation of the particular problem.

3. Particle Swarm optimisation (PSO). This is another population-based algorithm inspired by the social behaviour of bird or fish flockings [5]. In a PSO method, each element (particle) evolves by taking the combination of the current global best and individual best solutions into account. In the proposed distributed version of the PSO method, the server updates a main population and sends request to the clients to evolve a random subpopulation (in the same way as for the DE algorithm). This distributed variant shows similarities with the Multi-Swarm optimisation techniques [1] developed as a possible improvement of the PSO algorithm.

3. The optimisation problems

Originally, our method was designed to deal with bound-constrained optimisation problems:

$$\min f(x) \tag{1}$$

$$\text{subject to } x \in D, \tag{2}$$

where $D = [L_i, U_i], L_i, U_i \in \mathbb{R}$, and the objective function $f : \mathbb{R}^n \to \mathbb{R}^m$ is continuous in D. Nevertheless, the currently implemented solvers allow us to handle a certain group of inequality-constrained problems as well: namely, optimisation problems which, in addition to (2), have further inequality constraints in the form of

$$g_{i,1}(x_1, \ldots, x_{i-1}) \le x_i \le g_{i,2}(x_1, \ldots, x_{i-1}),\ i = 2, \ldots, n, \tag{3}$$

where the exact upper bound of $g_{i,1}$ and the exact lower bound of $g_{i,2}$ can be determined in a machine computable form (e.g. as an expression or a subroutine) for all $x_j \in [L_j, U_j], j = 1, \ldots, i-1$. This property allows us to replace a infeasible solution with a 'close' feasible solution, e.g. one located on the boundary of the feasible set. (Note, that the radio occultation

problem below can be formalized as a constrained problem in the above form: when generating feasible satellite orbits, the orbital elements *eccentricity* and *orbital period* are bounded by a function of the *semi-major axis*.)

We performed the numerical tests on the following hard test problems:

1. SAT: The radio-occultation problem described in [4] in detail. This is an optimisation problem of satellite constellations with a complex objective function structure. The optimisation has a dual objective: maximising the number of satellite occultations while distributing the occultations as uniformly as possible on the latitudes. This last objective is described by the standard deviation of the number of occultations occurring at different latitude stripes.

2. RB: The generalisation of the Rosenbrock global optimisation test function [8] given by minimising
$$f(x) = \sum_{i=1}^{n-1} (100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2). \tag{4}$$
We have used $n = 50$ and $x_i \in [-5.12, 5.12],\ i = 1, \ldots, n$.

3. LJ: A potential energy minimisation problem for the Lennard-Jones atom cluster for $d = 38$ atoms [11]. The potential is given by
$$f(p) = \sum_{1 \le i < j \le d} 4((1/r_{ij})^{12} - (1/r_{ij})^6), \tag{5}$$
where $p_i = (x_i, y_i, z_i),\ i = 1, \ldots, d$ is the location of the ith atom, and r_{ij} is the Euclidean distance between atoms i and j. We have used $x_1 = y_1 = z_1 = y_2 = z_2 = z_3 = 0$ and $x_i \in [0, 6],\ y_i, z_i \in [-3, 3]$ for all other variables. (Thus, (5) with $d = 38$ corresponds to an $n = 108$–dimensional problem.) This problem has important practical generalisations and it serves as a good test case for parallel and distributed solvers. The chosen problem instance is perhaps the most challenging one in the range of $1 \le n \le 50$. (Note that we did not intend to improve the best existing solution – this would definitely require far more sophisticated algorithms and problem formulation.)

4. Preliminary test results

We solved the above problems with each solver 10 times. The solver parameters related to the distributed implementations were the following:

- Population-related settings of the DE and PSO solvers:
  - The size of the main population was set to $MP = 5n$ for all problems.
  - The size of the subpopulations was set to $SP = MP/5$ for problems SAT and RB, and to $SP = MP/10$ for problem LJ. The reason for the latter setting was that we had to limit the size of problem and solution packages to about 30Kbytes in order to keep the network communication traffic within certain bounds.
  - The allowed number of iterations for evolving the subpopulations was chosen to be $IT = 2000$ for problems RB and LJ, and to $IT = 200$ for problem SAT.
- In the case of the MC solver, the allowed number of random sample generation per package was $IT \cdot SP$.
- For all solvers, the number of function evaluations was limited to $240\,000$, $5\,000\,000$, and $5\,400\,000$ for problems SAT, RB, and LJ, respectively.

The further control parameters of the DE and PSO algorithms were the default values taken from the implementations [2] and [3], respectively. For DE, we employed the algorithm variant cited as 'DE1' in the above reference.

The computations were performed during normal working days at the European Space Research and Technology Centre (ESTEC). The client application (hidden behind the screen saver) was installed on nine Windows–XP desktops. The sum of the CPU frequencies of the computers was approximately 15.1 GHz, which corresponds to a double-precision theoretical peak performance of about 15.1 Gflops. The results are summarized in Table 1 with respect to the **best** achieved objective function values. To measure the efficiency for the multi-objective SAT problem, we used a further criterion ([4]) in order to compare two solutions. (The variance of the individual test results was small for all strategies, thus, the displayed values can serve as a valid base of a comparison.) The last line of the table shows the previously known best solutions. These values come from [4] for SAT, and from [11] for LJ, respectively. The value given for RB is the known global minimum.

Table 1. The best solutions found during the test runs

solver	*SAT*	*RB*	*LJ*
MC	(1 134, 6.20)	1.574e+5	-10.16
DE	(2 722, 4.15)	24.98	-25.67
PSO	(2 174, 4.78)	27.20	-27.19
best known	(1 535, 7.78)	0	-173.93

Summarising the results, we can state that on the SAT and RB problems the DE algorithm outperformed the other two methods, while on the LJ problem the PSO method worked best. As we expected, these more sophisticated methods behaved far more efficiently than the MC search. In particular, on the SAT problem the Differential Evolution resulted in a big improvement on the previously known best solution (obtained by Monte–Carlo search,[4]). Our main future task is to find suitable distributed generalisations of these (and other) solvers in order to reach further performance improvements.

5. Conclusion and future research

Besides our future plans to extend the system with further global optimisation methods, the most promising way of improving the performance of the distributed environment is to employ the available solvers in an intelligent, co-operative way. This idea requires the comparison of the behaviour of the different solvers. We plan to develop and investigate a set of heuristics to direct the allocation of the packages and the selection of solvers. One such heuristic indicator can be the average (or expected) effort needed to improve the best existing solution by a unit amount, while using a given number of function evaluations and solver strategy. This indicator can be continuously updated for each solver in running time, and can be used as the basis measurement for further decisions.

References

[1] T. Blackwell and J. Branke, "Multi-swarm optimisation in Dynamic Environments", Proceedings of the Applications of Evolutionary Computing, EvoWorkshops, 2004, pp. 489–500.

[2] http://www.icsi.berkeley.edu/~storn/code.html

[3] http://www.engr.iupui.edu/~shi/pso.html

[4] D. Izzo, M.Cs. Markót, and I. Nann, “A distributed global optimiser applied to the design of a constellation performing radio-occultation measurements". Proceedings of the 15th AAS/AIAA Space Flight Mechanics Conference, Copper Mountain, Colorado, 2005.

[5] J. Kennedy and R.C. Eberhart, “Particle swarm optimisation", Proceedings of the IEEE international conference on neural networks, 1995, Vol. IV, pp. 1942–1948. IEEE service center, Piscataway, NJ.

[6] J. Lampinen, “Differential Evolution — New Naturally Parallel Approach for Engineering Design optimisation". In: Barry H.V. Topping (ed.): Developments in Computational Mechanics with High Performance Computing. Civil-Comp Press, Edinburgh, 1999, pp. 217–228.

[7] N. Metropolis and S. Ulam, “The Monte Carlo Method", J. Amer. Stat. Assoc. (44), 1949, pp. 335–341.

[8] H.H. Rosenbrock, “An automatic method for finding the greatest or least value of a function. Computer Journal (3), 1960, pp. 175–184.

[9] R. Storn and K. Price, “Differential Evolution - a Simple and Efficient Heuristic for Global optimisation over Continuous Spaces", J. Global optimisation (11), 1997, pp. 341-359.

[10] A.S. Tanenbaum, *Computer Networks*, Prentice Hall, 2003.

[11] D.J. Wales and J.P.K. Doye, “Global optimisation by basin-hopping and the lowest energy structures of Lennard-Jones clusters containing up to 110 Atoms", J. Phys. Chem. A. (101), 1997, pp. 5111–5116.

[12] Windows Sockets 2 API documentation,
ftp://ftp.microsoft.com/bussys/winsock/winsock2/WSAPI22.DOC