p-MEMPSODE: Parallel and Irregular Memetic Global Optimization

C. Voglis [a,1], P.E Hadjidoukas [b], K.E. Parsopoulos [a], D.G. Papageorgiou [c], I.E. Lagaris [a] M.N. Vrahatis [d]

[a] *Department of Computer Science and Engineering, University of Ioannina, P.O. BOX 1186, GR-45110 Ioannina, Greece*

[b] *Chair of Computational Science, ETH Zurich, Zurich CH-8092, Switzerland*

[c] *Department of Materials Science and Engineering, University of Ioannina, P.O. BOX 1186, GR-45110 Ioannina, Greece*

[d] *Department of Mathematics, University of Patras, GR-26110 Patras, Greece*

Abstract

A parallel memetic global optimization algorithm suitable for shared memory multicore systems is proposed and analyzed. The considered algorithm combines two well-known and widely used population-based stochastic algorithms, namely Particle Swarm Optimization and Differential Evolution, with two efficient and parallelizable local search procedures. The sequential version of the algorithm was first introduced as MEMPSODE (MEMetic Particle Swarm Optimization and Differential Evolution) and published in the CPC program library. We exploit the inherent and highly irregular parallelism of the memetic global optimization algorithm by means of a dynamic and multilevel approach based on the OpenMP tasking model. In our case, tasks correspond to local optimization procedures or simple function evaluations. Parallelization occurs at each iteration step of the memetic algorithm without affecting its searching efficiency. The proposed implementation, for the same random seed, reaches the same solution irrespectively of being executed sequentially or in parallel. Extensive experimental evaluation has been performed in order to illustrate the speedup achieved on a shared-memory multicore server.

PACS: 02.60.Pn

Key words: Parallel Global Optimization, Multicores, OpenMP, Particle Swarm Optimization, Differential Evolution, Memetic Algorithms, Local Search.

[1] Corresponding author: Department of Computer Science and Engineering, University of Ioannina, P.O. BOX 1186, GR-45110 Ioannina, Greece, Tel: +30 2651008834, Email: voglis@cs.uoi.gr

PROGRAM SUMMARY

Manuscript Title: p-MEMPSODE: Parallel and Irregular Memetic Global Optimization
Authors: C. Voglis, P.E. Hadjidoukas, K.E. Parsopoulos, D.G. Papageorgiou, I.E. Lagaris, M.N. Vrahatis
Program Title: p-MEMPSODE
Journal Reference:
Catalogue identifier:
Licensing provisions:
Programming language: ANSI C
Operating system: Developed under the Linux operating system using the GNU compilers v.4.4.3 (or higher). Uses the OpenMP API and runtime system.
RAM: The code uses $\mathcal{O}(n \times N)$ internal storage, n being the dimension of the problem and N the maximum population size. The required memory is dynamically allocated.
Word size: 64
Number of processors used: All available.
Supplementary material:
Keywords: Global Optimization, Memetic Algorithms, Particle Swarm Optimization, Differential Evolution, Local Search, OpenMP.
PACS: 02.60.Pn
Classification: 4.9
External routines:

Nature of problem: Numerical global optimization of real valued functions is an indispensable methodology for solving a multitude of problems in science and engineering. Many problems exhibit a number of local and/or global minimizers, expensive function evaluations or require real-time response. In addition, discontinuities of the objective function, non-smooth and deceitful landscapes constitute challenging obstacles for most optimization algorithms.

Solution method: We implement a memetic global optimization algorithm that combines stochastic, population-based methods with deterministic local search procedures. More specifically, the Unified Particle Swarm Optimization and the Differential Evolution algorithms are harnessed with the derivative-free Torczon's Multi-Directional Search and the gradient-based BFGS methods. The produced hybrid algorithms posses inherent parallelism. The OpenMP tasking model is employed to take advantage of this inherent parallelism and produce an efficient software implementation. The proposed implementation reaches the same solution irrespectively of being executed sequential or in parallel, given the same random seed.

Restrictions: The current version of the software uses only double precision arithmetic. An OpenMP-enabled (version 3.0 or higher) compiler is required.

Unusual features: The software requires bound constraints on the optimization variables.

Running time: The running time depends on the complexity of the objective function (and its derivatives if used) as well as on the number of available cores. Extensive experimental results demonstrate that the speedup closely approximates ideal values.

LONG WRITE-UP

1 Introduction

Numerical global optimization is an indispensable tool that has been widely applied on many scientific problems [1]. Usually, the solution of a problem is translated to the detection of global minimizer(s) of a properly defined objective function. Also, suboptimal solutions of acceptable quality can be obtained by closely approximating the actual global minimizers of the problem. In order to achieve such solutions within reasonable time and resource constraints, and algorithms shall maintain proper trade-off between their *exploration* (diversification) and *exploitation* (intensification) properties. Unfortunately, these two requirements are habitually conflicting for most algorithms. For example, modern stochastic optimization approaches such as Evolutionary Algorithms (EAs) [2,3] and Swarm Intelligence (SI) methods [4] are characterized by high exploration capabilities but lack the solution refinement properties of local search approaches [5].

Hybridization of optimization methods of different nature has proved valuable in addressing the aforementioned shortcomings [6]. In this context, an important family of global optimization algorithms has been developed, namely the *Memetic Algorithms* [7–9]. They stemmed from the necessity for powerful algorithms where the global exploration capability of EAs and SI approaches [2,4,10–15] is complemented with the efficiency and accuracy of standard *Local Search* (LS) techniques [5,16,17]. The combination and interaction between the two types of algorithms promotes the diffusion of their achievements by harnessing their strengths and generates algorithmic schemes that can tackle complex objective functions [18].

Another important issue in optimization problems is the running time of the algorithms. Demanding problems are usually accompanied by long execution times, attributed to their high computational demands and the complexity of the objective function. There are several applications where the time for a single function call is substantial, not to mention applications where real-time response is required. Such applications are frequently met in molecular biology, computational chemistry, engineering, aircraft design, and space trajectory planning [1,19–21].

Parallelization can drastically reduce the required processing time to find a solution. Parallelism in optimization methods can be found at various levels, including,

(i) function and gradient evaluations,

(ii) linear algebra computations,
(iii) operators of the optimization algorithms.

Global optimization algorithms that take advantage of parallel and distributed architectures are particularly suitable for solving problems with high computational requirements. The emerging multi-core architectures provide a cost-effective solution for high-performance optimization. Notwithstanding the large number of parallel global optimization algorithms in the literature, there are only a few practical implementations. Among the most representative are NOMAD [22,23], PaGMO/PyGMO [24,25], PGO [26], VTDIRECT95 [27], and pCMALib [28].

In this framework, the present paper introduces a parallel version of the recently proposed MEMPSODE software [29]. MEMPSODE implements a global optimization algorithm that falls into the category of Memetic Algorithms. It combines the exploration properties of two very popular population-based algorithms, namely Unified Particle Swarm Optimization (UPSO) and Differential Evolution (DE), with the strong convergence properties of the local search procedures implemented in the Merlin optimization environment [30]. MEMPSODE searching efficiency has been extensively examined in [31–34]. The proposed *Parallel MEMPSODE* (p-MEMPSODE) implementation adopts UPSO and DE as global search components, and integrates a quasi-Newton line search method that employs BFGS [5] updates as well as the Torczon's Multi-Directional Search (MDS) method [35]. Inheriting all convergence and exploration properties from its predecessor, p-MEMSODE aims to reduce the execution time on multicore servers.

The decision for using BFGS was based on its efficiency and popularity for smooth functions optimization, as well as its requirement for only first order derivatives that can be numerically approximated in parallel. On the other hand, MDS is an efficient, well studied, derivative-free method suitable for discontinuous functions, while it is inherently parallel. Both these methods satisfy our basic incentive to create a powerful hybrid optimization scheme with parallelizable components that can tackle the large runtime of various optimization problems. In order to achieve this on shared-memory multicore platforms, we used the OpenMP [36] tasking model, which allows for efficient exploitation of multiple levels of parallelism. Moreover, the choice of OpenMP offers portability and straightforward usage of our software.

Parallelization occurs at each iteration step of MDS and BFGS with numerical derivatives and does not modify their internal operations. Therefore, for a given starting point, the same minimum is retrieved executing the same number of iterations and function evaluations and irrespectively of the software being executed sequentially or in parallel. The same stands for the global components of the algorithm (UPSO and DE), where parallelization is also applied at their iteration level without affecting their outcome.

In the following sections, we describe in detail the parallelization properties and implementation of the proposed p-MEMPSODE implementation. Two levels of parallelism are exploited in order to achieve high performance and acceleration in execution time. Task spawning raises several issues due to its stochastic and irregular nature. Since we cannot foresee how many computational tasks may emerge at each iteration of the algorithm, task spawning and execution is self-adaptive to the specific instance of the algorithm.

2 Description of the Algorithms

The general algorithmic scheme implemented in p-MEMPSODE is based on a modified version of the serial algorithm implemented in [29]. It belongs to the category of Memetic Algorithms and combines the UPSO and DE algorithms with deterministic local searches that further exploit the best detected solutions. The hybridization strategy is essentially the same as presented in [29]. However, due to its sequential structure, the Merlin optimization environment is no longer employed in p-MEMPSODE. Instead, two local optimization algorithms that admit high parallelization are used. All algorithms as well as the hybrid scheme are described in the following sections.

2.1 Unified Particle Swarm Optimization

Particle Swarm Optimization (PSO) was originally introduced by Eberhart and Kennedy [37,38]. The algorithm employs a *swarm* of search agents, called *particles*, which iteratively probe the search space by changing their position according to an adaptable *velocity* (position shift). Also, the best position detected by each agent is stored in memory and it is exchanged among particles that belong in the same *neighborhood*.

Putting it formally, consider the n-dimensional continuous optimization problem,

$$\min_{x \in X \subset \mathbb{R}^n} f(x), \tag{1}$$

where the search space X is an orthogonal hyperbox in $\mathbb{R}^n$, i.e., $X \equiv [l_1, r_1] \times [l_2, r_2] \times \cdots \times [l_n, r_n]$. A swarm is a group of N particles (search agents), $S = \{x_1, x_2, \ldots, x_N\}$. Each particle assumes a current position,

$$x_i = (x_{i1}, x_{i2}, \ldots, x_{in})^\top \in X, \qquad i \in I,$$

an adaptable velocity (position shift),

$$v_i = (v_{i1}, v_{i2}, \dots, v_{in})^\top, \qquad i \in I,$$

and a memory of the best position it has visited so far,

$$p_i = (p_{i1}, p_{i2}, \dots, p_{in})^\top \in X, \qquad i \in I,$$

where $I = \{1, 2, \dots, N\}$. The swarm and velocities are usually initialized randomly and uniformly within the search space.

Also, each particle assumes a *neighborhood*, which is defined as a set indices of other particles with which it exchanges information,

$$\mathcal{N}_i = \{i - m, \dots, i - 1, i, i + 1, \dots, i + m\}.$$

This is the well known *ring* neighborhood topology [39], which assumes that the two ends of the ring coincide, i.e., the indices recycle at the ends. If all neighborhoods include the whole swarm, the *gbest PSO model* is defined. Otherwise, we have the case of the *lbest PSO model.*

Let g_i denote the best particle in $\mathcal{N}_i$, i.e.,

$$g_i = \arg\min_{j \in \mathcal{N}_i} f(p_j),$$

and t denote the algorithm's iteration counter. Then, the particle positions and velocities are updated at each iteration as follows [40]:

$$v_{ij}^{(t+1)} = \chi \left[v_{ij}^{(t)} + c_1 r_1 \left(p_{ij}^{(t)} - x_{ij}^{(t)} \right) + c_2 r_2 \left(p_{g_i j}^{(t)} - x_{ij}^{(t)} \right) \right], \tag{2}$$

$$x_{ij}^{(t+1)} = x_{ij}^{(t)} + v_{ij}^{(t+1)}, \tag{3}$$

where $i \in I$, $j = 1, 2, \dots, n$; χ is the *constriction coefficient*; c_1 and c_2 are positive constants called *cognitive* and *social* parameter, respectively; and r_1, r_2, are random numbers drawn from a uniform distribution in the range $[0, 1]$. The best position of each particle is also updated at each iteration as follows,

$$p_{ij}^{(t+1)} = \begin{cases} x_{ij}^{(t+1)}, & \text{if } f\left(x_{ij}^{(t+1)}\right) < f\left(p_{ij}^{(t)}\right), \\ p_{ij}^{(t)}, & \text{otherwise.} \end{cases} \tag{4}$$

Clerc and Kennedy [40] provided a stability analysis of the PSO model described above, resulting in a relation among its parameters,

$$\chi = \frac{2}{|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}|}, \tag{5}$$

for $\varphi = c_1 + c_2 > 4$, from which the default parameter set, $\chi = 0.729, \quad c_1 = c_2 = 2.05$, was derived.

Unified PSO (UPSO) generalizes the basic PSO scheme by combining the lbest and gbest models, in order to harness their exploration/exploitation properties. Let $G_i^{(t+1)}$ and $L_i^{(t+1)}$ denote the velocity of x_i in the gbest and lbest PSO model, respectively [41,42], i.e.,

$$G_{ij}^{(t+1)} = \chi \left[v_{ij}^{(t)} + c_1 r_1 \left(p_{ij}^{(t)} - x_{ij}^{(t)} \right) + c_2 r_2 \left(p_{gj}^{(t)} - x_{ij}^{(t)} \right) \right], \tag{6}$$
$$L_{ij}^{(t+1)} = \chi \left[v_{ij}^{(t)} + c_1 r_1 \left(p_{ij}^{(t)} - x_{ij}^{(t)} \right) + c_2 r_2 \left(p_{g_i j}^{(t)} - x_{ij}^{(t)} \right) \right], \tag{7}$$

where g is the index of the overall best particle, i.e.,

$$g = \arg \min_{j=1,\ldots,N} f(p_j).$$

Then, UPSO updates the particles and velocities as follows [41,42],

$$U_{ij}^{(t+1)} = u\, G_{ij}^{(t+1)} + (1-u)\, L_{ij}^{(t+1)}, \tag{8}$$
$$x_{ij}^{(t+1)} = x_{ij}^{(t)} + U_{ij}^{(t+1)}, \tag{9}$$

where $i \in I$, $j = 1, 2, \ldots, n$, and the parameter $u \in [0, 1]$ is called the *unification factor* and it balances the trade-off between the two models. Obviously, the standard lbest PSO model corresponds to $u = 0$, and the gbest PSO model is obtained for $u = 1$. All intermediate values $u \in (0, 1)$ produce combinations with diverse convergence properties.

UPSO can also assume a mutation operator that resembles mutation in EAs, inducing further diversity in the swarm. The reader is referred to the main MEMPSODE software [29] for further details.

2.2 Differential Evolution

Differential Evolution (DE) was introduced by Storn and Price [43,44]. It works similarly to PSO, assuming a population $P = \{x_1, x_2, \ldots, x_N\}$ of search points, called *individuals*, that probe the search space $X \subset \mathbb{R}^n$. Again, the population is randomly initialized following a uniform distribution within the search space.

Each individual is an n-dimensional vector (candidate solution),

$$x_i = (x_{i1}, x_{i2}, \ldots, x_{in})^\top \in X, \qquad i \in I,$$

where $I = \{1, 2, \ldots, N\}$. The population P is iteratively evolved by applying the *mutation* and *recombination* operators on each individual. Mutation produces a new vector v_i for each individual x_i, by combining some of the rest individuals of the population. There are various operators for this task. Some of the most popular are the following:

$$\begin{aligned}
\text{OP1:}\quad & v_i^{(t+1)} = x_g^{(t)} + F\left(x_{r_1}^{(t)} - x_{r_2}^{(t)}\right), && (10)\\
\text{OP2:}\quad & v_i^{(t+1)} = x_{r_1}^{(t)} + F\left(x_{r_2}^{(t)} - x_{r_3}^{(t)}\right), && (11)\\
\text{OP3:}\quad & v_i^{(t+1)} = x_i^{(t)} + F\left(x_g^{(t)} - x_i^{(t)} + x_{r_1}^{(t)} - x_{r_2}^{(t)}\right), && (12)\\
\text{OP4:}\quad & v_i^{(t+1)} = x_g^{(t)} + F\left(x_{r_1}^{(t)} - x_{r_2}^{(t)} + x_{r_3}^{(t)} - x_{r_4}^{(t)}\right), && (13)\\
\text{OP5:}\quad & v_i^{(t+1)} = x_{r_1}^{(t)} + F\left(x_{r_2}^{(t)} - x_{r_3}^{(t)} + x_{r_4}^{(t)} - x_{r_5}^{(t)}\right), && (14)
\end{aligned}$$

where t denotes the iteration counter; $F \in (0, 1]$ is a fixed user-defined parameter; g denotes the index of the best individual in the population; and $r_j \in \{1, 2, \ldots, N\}$, $j = 1, 2, \ldots, 5$, are mutually different randomly selected indices that differ also from the index i. All vector operations in Eqs. (10)-(14) are performed componentwise.

After mutation, recombination is applied to produce a *trial vector*,

$$u_i = (u_{i1}, u_{i2}, \ldots, u_{in}), \qquad i = 1, 2, \ldots, N,$$

for each individual. This vector is defined as follows:

$$u_{ij}^{(t+1)} = \begin{cases} v_{ij}^{(t+1)}, & \text{if } R_j \leqslant CR \text{ or } j = \mathrm{RI}(i), \\ x_{ij}^{(t)}, & \text{if } R_j > CR \text{ and } j \neq \mathrm{RI}(i), \end{cases} \tag{15}$$

where $j = 1, 2, \ldots, n$; R_j is a random variable uniformly distributed in the range $[0, 1]$; $CR \in [0, 1]$ is a user-defined *crossover constant*; and $\mathrm{RI}(i) \in \{1, 2, \ldots, n\}$, is a randomly selected index.

Finally, each trial vector u_i is compared against its original individual x_i and the best between them comprise the new individual in the next generation, i.e.,

$$x_i^{(t+1)} = \begin{cases} u_i^{(t+1)}, & \text{if } f\left(u_i^{(t+1)}\right) < f\left(x_i^{(t)}\right), \\ x_i^{(t)}, & \text{otherwise.} \end{cases} \tag{16}$$

DE is a relatively greedy algorithm that can be very efficient under proper parameter setting. The reader is referred to [29] for further implementation details.

2.3 Local Search

As already mentioned, the Merlin optimization environment, which was the main local search provider for the MEMPSODE software, was abandoned in the present work due to its serial nature. Instead, two LS approaches were selected such that nested parallelization could be achieved. The first level corresponds to the individual particles of the swarm and the second level to the execution of the LS procedures. We acknowledge the need of LS algorithms that work on discontinuous and noisy functions, as well as the necessity of a robust algorithm for continuous well-behaved functions. For these reasons, the selected algorithms are Torczon's Multi-Directional Search (MDS) [35] and the BFGS [5] quasi-Newton method with either parallel numerical or analytical derivatives. Although BFGS with analytical derivatives is not parallel *per se*, it achieves great performance on many continuously differentiable cases. In the following subsections we give a brief introduction to the MDS and BFGS algorithms, which are not provided in the corresponding MEMPSODE references [29].

2.3.1 Multi-Directional Search

In the early 90s, the *Multi-Directional Search* (MDS) algorithm was introduced by Torczon [35]. MDS operates on the $n+1$ vertices of an n-dimensional simplex defined in the search space. It applies a sequence of *reflection*, *expansion*, and *contraction* operators on the edges of the simplex and, under mild assumptions, it provides guaranteed convergence to a local minimum [45]. MDS was devised to operate without using derivative information of the objective function but only concurrent function evaluations.

If t denotes the iteration counter, the corresponding simplex consists of $n+1$ points,

$$S^{(t)} = \left\{x_0^{(t)}, x_1^{(t)}, \ldots, x_n^{(t)}\right\}, \qquad x_i^{(t)} \in X \subset \mathbb{R}^n.$$

The barycenter of the simplex,

$$x_c^{(t)} = \frac{1}{n+1} \sum_{i=0}^{n} x_i^{(t)},$$

is considered as the approximation to the minimizer. The objective function is evaluated at all vertices of the simplex and the indices are rearranged such that,

$$f\left(x_0^{(t)}\right) = \min_{i=0,\ldots,n} f\left(x_i^{(t)}\right),$$

i.e., the *best* vertex (the one with the smallest function value) is always $x_0^{(t)}$.

The transition to the next iteration is performed by pivoting the simplex

Procedure MDS(f, $\mathbf{x}^{(0)}$, μ, θ, $\mathbf{x}^*$)

```
   Input: Objective function, $f : X \subset \mathbb{R}^n \to \mathbb{R}$; initial point $\boldsymbol{x}^{(0)}$; parameters $\mu \in (1, +\infty)$, $\theta \in (0, 1)$
   Output: Approximation to local minimizer: $x^*$
1  Create $S^{(0)} = \{\boldsymbol{x}_0^{(0)}, \boldsymbol{x}_1^{(0)}, \dots, \boldsymbol{x}_n^{(0)}\}$ that contain $\boldsymbol{x}^{(0)}$
2  $min \leftarrow \arg\min_i \{f(\boldsymbol{x}_i^{(0)})\}$ and swap $\boldsymbol{x}_{min}^{(0)}$ and $\boldsymbol{x}_0^{(0)}$
3  for $t = 0, 1, \dots$ do
4      Check the stopping criterion
       // Reflection step
5      for $i = 1, 2, \dots, n$ do
6          $\boldsymbol{r}_i^t \leftarrow 2\boldsymbol{x}_0^{(t)} - \boldsymbol{x}_i^{(t)}$
7          Evaluate $f(\boldsymbol{r}_i^t)$
8      end
9      if $\min_i \left\{f(\boldsymbol{r}_i^t), i = 1, 2, \dots, n\right\} < f(\boldsymbol{x}_0^{(t)})$ then
           // Expansion step
10         for $i = 1, 2, \dots, n$ do
11             $\boldsymbol{e}_i^t \leftarrow (1 - \mu)\boldsymbol{x}_0^{(t)} + \mu \boldsymbol{r}_i^{(t)}$
12             Evaluate $f(\boldsymbol{e}_i^t)$
13         end
14         if $\min_i \left\{f(\boldsymbol{e}_i^t), i = 1, 2, \dots, n\right\} < \min_i \left\{f(\boldsymbol{r}_i^t), i = 1, 2, \dots, n\right\}$ then
               // Expansion step accepted
15             $\boldsymbol{x}_i^{(t+1)} \leftarrow \boldsymbol{e}_i^t,\ i = 1, 2, \dots, n$
16         else
               // Reflection step accepted
17             $\boldsymbol{x}_i^{(t+1)} \leftarrow \boldsymbol{r}_i^t,\ i = 1, 2, \dots, n$
18         end
19     else
           // Contraction step
20         for $i = 1, \dots, n$ do
21             $\boldsymbol{c}_i^t \leftarrow (1 + \theta)\boldsymbol{x}_0^{(t)} - \theta \boldsymbol{r}_i^{(t)}$
22             Evaluate $f(\boldsymbol{c}_i^t)$
23         end
           // Always accept contraction
24         $\boldsymbol{x}_i^{(t+1)} \leftarrow \boldsymbol{c}_i^t, i = 1, 2, \dots, n$
25     end
26     $min \leftarrow \arg\min_i \{f(\boldsymbol{x}_i^{(t+1)})\}$ and swap $\boldsymbol{x}_{min}^{(t+1)}$ and $\boldsymbol{x}_0^{(t+1)}$
27 end
28 $\boldsymbol{x}^* \leftarrow \frac{1}{n+1} \sum_{i=0}^{n} \boldsymbol{x}_i^{(t)}$
```

around $\boldsymbol{x}_0^{(t)}$ and attempting a reflection step. The objective function is then evaluated at the n reflected vertices of the simplex. If a better point is obtained, then an expansion step is attempted to produce an even larger reflected simplex. On the other hand, if the reflection step fails to improve the best point, a contraction step is attempted to reduce the size of the considered simplex.

The procedure is repeated until a termination criterion is satisfied. This criterion can be a maximum number of iterations or function evaluations or a number of consequent non-improving iterations. The MDS algorithm is presented in the pseudocode of Procedure MDS. Apparently, the function evaluations in lines 7, 12, and 22, are independent, hence, they can be performed in parallel.

Procedure BFGS(f, $\mathbf{x}^{(0)}$, $\mathbf{x}^{*}$)

```
Input  : Objective function, $f : S \subset \mathbb{R}^n \to \mathbb{R}$; starting point: $x^{(0)}$
Output: Approximation of the minimizer: $x^*$
// Initialization
1  $B^{(0)} = I$
2  $g^{(0)} = \nabla f(x^{(0)})$
3  for $k \leftarrow 1, 2, \ldots$ do
       // Linear system solution
4      Solve $B^{(k)} d^{(k)} = -g^{(k)}$
       // Line search
5      Set $x^{(k+1)} = x^{(k)} + \lambda^{(k)} d^{(k)}$, where $\lambda^{(k)}$ satisfies the Wolfe conditions (1) and (2)
       // Derivative evaluation
6      Calculate $g^{(k+1)} = \nabla f(x^{(k+1)})$
       // BFGS update
7      $y^{(k)} = g^{(k+1)} - g^{(k)}$
8      $s^{(k)} = x^{(k+1)} - x^{(k)}$
9      $B^{(k+1)} = B^{(k)} - \frac{B^{(k)} s^{(k)} s^{(k)\top} B^{(k)}}{s^{(k)\top} B^{(k)} s^{(k)}} + \frac{y^{(k)} y^{(k)\top}}{y^{(k)\top} s^{(k)}}$
10     Stop if convergence criterion is met
11 end
12 Set $x^* = x^{(k+1)}$
```

2.3.2 BFGS Algorithm

The BFGS algorithm falls into the category of quasi-Newton methods [5,17]. Quasi-Newton algorithms assume that, at each iteration k, a point $x^{(k)}$, the gradient $g^{(k)}$, and an approximation $B^{(k)}$ to the Hessian matrix, are available. Then, a descent direction $d^{(k)}$ is computed by solving the linear system,

$$B^{(k)}\, d^{(k)} = -g^{(k)},$$

and a *line search* is initiated from $x^{(k)}$ along the search direction $d^{(k)}$. The outcome of the line search is a scalar step $\lambda^{(k)}$ that leads to the next approximation,

$$x^{(k+1)} = x^{(k)} + \lambda^{(k)}\, d^{(k)}.$$

The step $\lambda^{(k)}$ is properly selected to satisfy the *Wolfe conditions*,

$$f\left(x^{(k)} + \lambda^{(k)} d^{(k)}\right) \leqslant f\left(x^{(k)}\right) + \rho\, \lambda^{(k)}\, g^{(k)\top} d^{(k)}, \tag{17}$$

$$\nabla f\left(x^{(k)} + \lambda^{(k)} d^{(k)}\right)^{\top} d^{(k)} \geqslant \sigma\, \nabla f\left(x^{(k)}\right)^{\top} d^{(k)}, \tag{18}$$

where $\rho \in (0,1)$ and $\sigma \in (\rho, 1)$.

The iteration is completed by updating the approximate Hessian matrix $B^{(k)}$ using only first-order derivative information from $x^{(k)}$ and $x^{(k+1)}$. The algorithm is given in Procedure BFGS.

2.4 Memetic Strategies

The MEMPSODE software implements the *memetic strategies* proposed in [46] for the application of local search:

Strategy 1: LS is applied only on the overall best position p_g of the swarm.
Strategy 2: LS is applied on each locally best position, p_i, $i \in I$, with a prescribed fixed probability, $\rho \in (0, 1]$.
Strategy 3: LS is applied both on the best position p_g as well as on some randomly selected locally best positions p_i, $i \in I$.

The application of a LS can either take place at each iteration or after a specific number of iterations. Also, for practical reason, only a small number of particles are considered as start points for LS, following the suggestions in [47]. The corresponding DE strategies assume only the corresponding individuals in the population. The reader is referred to [29] for further details. In the current p-MEMPSODE implementation, the aforementioned strategies were retained.

3 Parallelization Issues

3.1 Parallelizing PSO and DE

The structure of both PSO and DE is intrinsically parallel and perfectly suits the well established master-worker execution model. The master retains the current and best positions in the PSO case and the population for the DE case. It creates N function evaluation tasks for the workers. A synchronization point exists at the end of each iteration of the algorithm so that the master retrieves all new function values before updating swarm velocities and positions or DE individuals.

Recently, asynchronous implementations have been proposed for PSO [48]. In such cases, there is no synchronization point and the updating step is performed by the master using the currently available information. Preliminary experiments have shown that although the synchronized version may achieve faster convergence, asynchronous task handling can increase parallel efficiency.

Although the synchronous variant has been followed in p-MEMPSODE, no loss of parallel efficiency is guaranteed by the employment of a dynamic nested parallelization scheme (see following sections). The synchronization point at the end of each iteration ensures that the approximation of the minimizer will

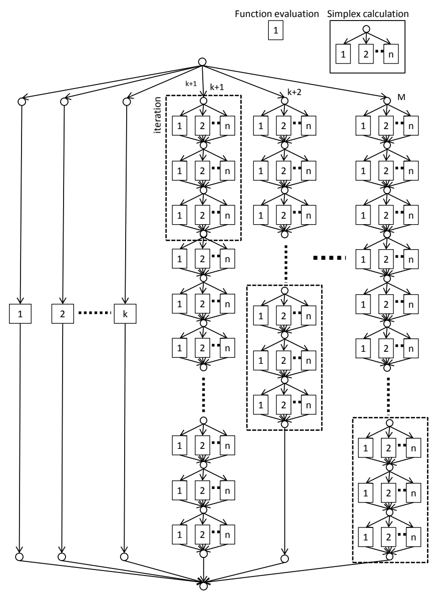


Fig. 1. Execution task graph for the memetic scheme with MDS local search.

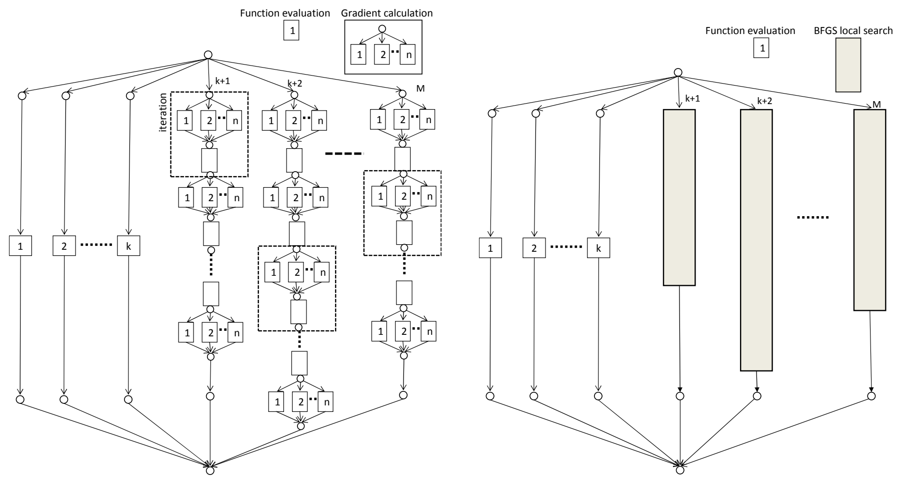
not be affected by parallel or sequential execution of the algorithm.

3.2 Parallelizing MDS and BFGS

MDS is an inherently parallel method. At each iteration, at least n function evaluations can be performed concurrently. In the simplest implementation, three synchronization points for n function evaluation tasks can be included in lines 8, 13, and 23 of Procedure MDS. Alternatively, one may concurrently execute the reflection, expansion, and contraction steps by launching $3n$ function evaluation tasks. In this case, the best simplex is chosen for the next iteration after gathering all results. Increasing the number of concurrent tasks may lead to better parallel efficiency on larger computational systems.

Regarding the BFGS algorithm, since we consider function evaluations as the major computational tasks, the essential source of parallelization lies in the parallel estimation of derivatives. Numerical differentiation via finite differences can also be efficiently implemented using the master-worker model [49].

It is obvious that parallelizing each iteration of MDS or BFGS using synchronization points does not affect the minimum approximation and the required number of iterations and function evaluations. That means that executing the parallel algorithms, from the same starting point, either in parallel or sequentially, they will lead to the same minimizer having consumed the same number of function evaluations.



(a) Numerical derivatives (b) Analytical derivatives

Fig. 2. Execution task graph for the memetic scheme with BFGS local search.

3.3 Parallelizing the Memetic Algorithm

The stochastic nature of its components, renders the Memetic Algorithm a difficult case for effective parallelization. The basic loop of the algorithm spawns N independent tasks, each one associated with a specific particle/individual. Some of these tasks are simple function evaluations (FE) while the rest are local searches (LS). The probability ρ controls which particles will execute FE (approximately $(1-\rho)\times N$ particles) and which will initiate a local search (approximately $\rho \times N$ particles).

An LS algorithm may require several hundreds of function evaluations, thus its computational cost is substantially large. On the other hand, a worker, i.e. processing unit, that is assigned an FE task is expected to finish much sooner than a worker executing an LS. One way to achieve higher efficiency is to introduce an additional level of parallelism inside the LS task, by assigning new tasks to processors that have already finished their FEs. This requires the use of parallel local searches like MDS or BFGS with parallel numerical derivatives.

In the case that we employ MDS as the LS component, we introduce a second level of parallelism. At this inner level, n independent calls to the objective function are made for the computation of the reflection simplex and, subsequently, n calls for expansion and n calls for contraction. The distribution of tasks in this case is shown in Fig. 1, where the computational tasks are depicted in square boxes and represent function evaluations. The iterations of the MDS procedure are not known beforehand, unless we choose a single termination criterion that is based on the maximum number of iterations or function evaluations. However, in practice, termination criteria based on the proximity to the solution are habitually used. In this way, wasting of computational resources is avoided.

When the user selects the BFGS algorithm, there are two alternative task distribution schemes based on whether parallel numerical or analytical derivatives are used. The task graph of the first case is shown in Fig. 2(a). Here, the iterative local optimization spawns an inner-level group of at least n tasks that correspond to function evaluations required for the gradient calculation. On the other hand, when analytical derivatives are employed the task graph of the memetic algorithm is the one illustrated in Fig. 2(b). In this case, lengthy first level tasks are created. Thus, in the best case, the total runtime will be equal to the lengthiest local search task.

3.4 OpenMP directives

According to the above presentation, the parallelism of the Memetic Algorithm is highly irregular and depends on several factors:

(a) The swarm-size and the probability of performing local searches.
(b) The time steps required by the local optimization for finding a minimum.
(c) The dimensionality and execution time of the objective function.

An efficient parallel implementation requires flexible management of this dynamic and highly irregular nested parallelism. To achieve this on shared-memory platforms, we employed the OpenMP [36] tasking model.

Tasks were first introduced in OpenMP v.3.0, aiming to extend the expressiveness of the programming model beyond loop-level and coarse-grain parallelism. Both function evaluations and MDS calls are expressed with OpenMP tasks, with the latter dynamically spawning further function evaluation tasks.

Specifically, only a single team of threads for all levels of parallelism is created. The master thread runs an implicit task, which is the *primary task* of the application and it executes the main loop of the algorithm, while it iteratively spawns first-level tasks using the `task` construct. The rest of the threads reach the end of the parallel region and begin the execution of these tasks. The primary task then encounters a `taskwait` construct and waits for its children tasks to complete, while the master thread participates in their execution too.

Any OpenMP thread that executes an MDS task, dynamically spawns additional tasks at the innermost level of parallelism, following the same fork-join (master-worker) approach. The underlying OpenMP runtime library is responsible for the scheduling of all these tasks across the available cores.

The OpenMP tasking model allows to instantiate the task graph of the hybrid algorithm in a straightforward way and effectively exploit the multiple levels of parallelism. In addition, due to the single team of threads, our parallel application avoids the overheads and performance implications of OpenMP nested parallel regions.

Listing 1. OpenMP directives in MDS

```
while (termination_condition()) {
   k = minimum_simplex(fu, n);     // Find minimum simplex point k
   swap_simplex(u, fu, n, k, 0);  // Swap minimum and first point

   // rotation step
   fr[0] = fu[0];
   for (i = 1; i < n + 1; i++) {
      r[i] = u[0] - (u[i] - u[0]);
#pragma omp task shared(fr, r, n) firstprivate(i)
      {
         Objective_F(&r[i], n, &fr[i]);
      }
   }
#pragma omp taskwait
   k = minimum_simplex(fr, n);
   if (fr[k] < fu[0]){
      expand = 1;
   }
   if (expand){ // expand
      fec[0] = fu[0];
      for (i = 1; i < n + 1; i++) {
         ec[i] = u[0] - mu * ((u[i] - u[0]));
#pragma omp task shared(fec, ec, n) firstprivate(i)
         {
            Objective_F(&ec[i], n, &fec[i]);
         }
      }
#pragma omp taskwait
      kec = minimum_simplex(fec, n);
      if (fec[kec] < fr[k]) {
         assign_simplex(u, fu, ec, fec, n);
      } else {
         assign_simplex(u, fu, r, fr, n);
      }
   }
   else { // contract
      fec[0] = fu[0];
      for (i = 1; i < n + 1; i++) {
         ec[i] = u[0] + theta * ((u[i] - u[0]));
#pragma omp task shared(fec, ec, n) firstprivate(i)
         {
            Objective_F(&ec[i], n, &fec[i]);
         }
      }
#pragma omp taskwait
      assign_simplex(u, fu, ec, fec, n);
   }
} /* while termination */
```

Listing 2. OpenMP directives in the core of the memetic scheme

```
// Define which particles will perform local searches
for (i = 0; i < M; i++) {
   if (drand48() < P) {
      eval[i] = 0;
      local[i] = 1;
   } else {
      eval[i] = 1;
      local[i] = 0;
   }
}

for (i = 0; i < M; i++) {
   if (eval[i] == 1) {
#pragma omp task untied firstprivate(i) shared(N, swarm, fswarm, stats) private(j, FX)
      {
         Evaluate_Particle(&swarm[i], &FX, N);
#pragma omp critical
         {
            stats->fev = stats->fev + 1;
         }
         fswarm[i] = FX;
      } /* task */
   }
}

for (i = 0; i < M; i++) {
   if (local[i] == 1) {
#pragma omp task untied firstprivate(i) private(GRMS,fev,gev) shared(bestpos, N, fbestpos, xll, xrl)
      {
         local_optimization(&bestpos[i], N, &fbestpos[i], &GRMS, xll, xrl, &fev, &gev);
#pragma omp critical
         {
            stats->lopts = stats->lopts + 1;
            stats->localsearch = stats->localsearch + 1;
            stats->fev = stats->fev + fev;
            stats->gev = stats->gev + gev;
         }
      } /* task */
   }
}
#pragma omp taskwait
```

4 Software Description

The proposed p-MEMPSODE software can be built either as a standalone executable or as a library that exports a user callable interface. Like its serial predecessor, all algorithmic parameters are controlled by a rich variety of options, while all its components are written in ANSI C.

The software also contains a set of sample objective functions, including an interface to the Tinker [50] package for molecular mechanics calculations. During optimization, p-MEMPSODE prints informative messages on the screen and, upon termination, appropriate output files are created. For installation instructions and detailed examples we refer the reader to the extensive readme file in the software distribution.

4.1 User-Defined Subroutines

The user must provide the following subroutines:

(1) `void Objective_F (double x[], int n, double *f)`
Returns the value of the objective function evaluated at x.

`x`	(input)	Array containing the evaluation point.
`n`	(input)	Dimension of the objective function.
`f`	(output)	Objective function value.

(2) `void Bounds_F(double l[], double r[], int n)`
Returns the double-precision array `l` with the lower bounds and the double-precision array `r` with the upper bounds of the variables.

`n`	(input)	Dimension of the objective function.
`l`	(output)	Array containing the lower bounds.
`r`	(output)	Array containing the upper bounds.

For the first order derivatives, the gradient is also needed:

(3) `void Objective_G (double x[], int n, double g[])`
Returns the gradient vector evaluated at x.

`x`	(input)	Array containing the evaluation point.
`n`	(input)	Dimension of the objective function.
`g`	(output)	Gradient vector.

4.2 Installation and Use: Standalone Version

Assuming that the above user-defined subroutines are included in file `fun.c`, the standalone executable can be built using the `make` utility:

```
make OBJECTIVE=fun
```

Then the user can define the number of OpenMP threads that will be used to carry out parallel computation by issuing the command:

```
export OMP_NUM_THREADS=8
```

Subsequently, execution is initiated from the command line:

`mempsode`	`-d 20`	`-a pso`	`-l 2`	`-m 3`	`-s 100`	`-f 10000`
	dimension of the problem	choice between UPSO and DE	memetic scheme	local search	swarm size	maximum function evaluations

The above commands execute p-MEMPSODE on 8 OpenMP threads using UPSO (option `-a pso`) for the 20-dimensional provided objective function (option `-d 20`), swarm size 100 (option `-s 100`), maximum number of function evaluations 10000 (option `-f 10000`), the second memetic scheme (option `-l 2`) and the third local search procedure (BFGS with analytic derivatives) (option `-m 3`).

Algorithm specific parameters can be defined using the command line arguments `-B` for BFGS and `-T` for MDS, which are presented below along with their default values:

(1) BFGS parameters: `-B="param1=value param2=value ..."`

`feps=1.e-8`	Function value termination criterion tolerance.
`xeps=1.e-8`	X-termination criterion tolerance.
`geps=1.e-8`	Gradient norm termination criterion tolerance.
`rho=1.e-4`	Line search ρ parameter.
`sigma=0.9`	Line search σ parameter.
`maxiter=300`	Maximum number of iterations.
`maxfevals=1000`	Maximum number of function evaluations.
`lsiter=30`	Maximum number of line search iterations.

(2) MDS parameters: `-T="param1=value param2=value ..."`

`mu=2.0`	μ parameter.
`theta=0.5`	θ parameter.
`maxiter=300`	Maximum number of iterations.
`maxfevals=1000`	Maximum number of function evaluations.

A complete list of all available command line options is provided in the distribution or through the command:

```
mempsode -h
```

During the optimization procedure the software provides printout information (iterations, function evaluations, minimum objective function value) and upon termination a detailed message containing the minimum, the total number of function evaluations, the number of local searches etc. In addition, the number of tasks executed by each of the OpenMP threads can be retrieved.

4.3 Installation and Use: Library Version

The p-MEMPSODE software can be also built as a library by issuing the command:

```
make lib
```

This command creates the file `libmempsode.a`, which contains a set of interface routines. To render them usable, the user must provide appropriate function calls, as illustrated below:

```
#include "mempsode.h"
...
init_mempsode();
set_mempsode_iparam("dimension", 20);
set_mempsode_cparam("algorithm", "pso");
set_mempsode_iparam("memetic", 2);

/* Use BFGS with analytic derivatives */
set_mempsode_iparam("local-search", 3);

/* Set maximum BFGS iterations */
set_mempsode_cparam("bfgs-params", "maxiter=300");

set_mempsode_iparam("swarm-size", 100);
set_mempsode_iparam("max-fun-evals", 10000);
mempsode();
get_mempsode_min("minval", &val);
...
```

The above code sets various p-MEMPSODE parameters, calls the main optimization routine, and finally retrieves the best function value found. Assuming that file `main.c` contains the above code fragment and file `fun.c` contains the objective function, they can be linked with the p-MEMPSODE library as follows:

```
gcc main.c fun.c -L/path/to/mempsode -I/path/to/mempsode -lmempsode -fopenmp
```

The complete list of interface routines as well as various installation options are described in the software distribution.

Table 1
Experimental setting for the Rastrigin test function

Setting	Probability (ρ)	Dimension (n)
E1	0.01	30
E2	0.01	50
E3	0.01	100
E4	0.05	30
E5	0.05	50
E6	0.05	100
E7	0.1	30
E8	0.1	50
E9	0.1	100
E10	0.2	30
E11	0.2	50
E12	0.2	100

5 Sample Applications

In order to examine the parallel efficiency of p-MEMPSODE, we provide three categories of sample applications. The first one considers the global minimization of multimodal objective functions. For this purpose, four widely used multimodal test function were chosen. Since parallel algorithms are more useful in cases of computationally demanding objective functions, an artificial delay was added to each function call.

The other two applications refer to molecular conformation problems using pairwise and Tinker potentials. The parallel p-MEMPSODE software was tested on a multicore server with 16-core AMD Opteron CPUs and 16GB of RAM. The software was compiled under Linux 2.6 with GNU gcc 4.7 and OpenMP 3.1.

5.1 Application on Artificial Test Functions

To measure the parallel efficiency of our implementation under different parameter configurations we used a set of standard multimodal test functions (see Table 2). These test problems are widely used in the literature to assess the quality of global optimization algorithms. Since we are interested in the parallel speedup, we added artificial delays of 1ms and 10ms to each function call. Preliminary experiments indicated that using delays larger than 10ms does not affect the speedup measurements, hence we excluded them from the analysis. We note that p-MEMPSODE retrieves the global minimizer in all cases reported, provided a large number of available function evaluations.

Table 2
Artificial test functions

Function	Formula	Minimizer
Rastrigin	$f(x) = 10\,n + \sum_{i=1}^{n} \left(x_i^2 - 10\,\cos(2\pi x_i)\right)$	$f^* = 0,\ x^* = (0, 0, \ldots, 0)^\top$
Ackley	$f(x) = 20 + \exp(1)\ -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right)$	$f^* = 0,\ x^* = (0, 0, \ldots, 0)^\top$
Griewank	$f(x) = \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$	$f^* = 0,\ x^* = (0, 0, \ldots, 0)^\top$
Schwefel	$f(x) = 418.9829 \cdot n - \sum_{i=1}^{n} x_i \sin\left(\sqrt{\lvert x_i \rvert}\right)$	$f^* = 0,\ x^* = (420.9687, 420.9687, \ldots)^\top$

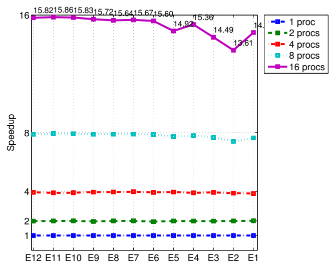
We tested all the LS algorithms implemented in the software. In Table 1 we define 12 different parameter configurations by varying the probability of local search and the dimension of the objective function. The swarm size was set equal to the dimension for each case.

The probability ρ controls the number of parallel LS tasks launched at each iteration in the outer parallelization level. The dimension n also defines the number of concurrent function evaluations in the case of MDS and BFGS with numerical derivatives, at the inner parallelization level. Small probability and dimension result in small number of tasks. Increasing these quantities also increases the number of parallel tasks, which theoretically favors parallel efficiency. We also expect more parallel tasks to be spawned as the swarm size increases.

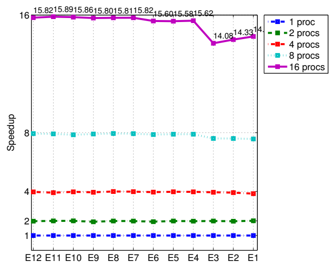
Figure 3 illustrates the achieved speedup when MDS was used as the LS component. The results are arranged in a 3×2 grid. Each row of figures corresponds to a specific swarm size and each column to a specific computational delay level (left column: 1ms, right column: 10ms). Each figure illustrates the speedup for the 12 different parameter settings of Table 1, on 1, 2, 4, 8, and 16 processors. The ideal speedup is the corresponding horizontal line per number of processors.

As we can see, almost perfect speedup is obtained for both delay values, even for the case of 16 processors. As expected, inferior performance is observed when the number of parallel tasks is relatively small (Experiment E1 with swarm size 30), while the best performance is achieved when the number of tasks is high (Experiment E12 with swarm size 100).

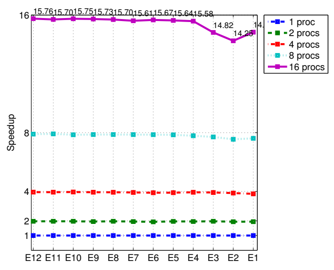
The case of BFGS with numerical derivatives is illustrated in Figure 4. All speedup measurements with varying swarm size (a single row in Figure 4)



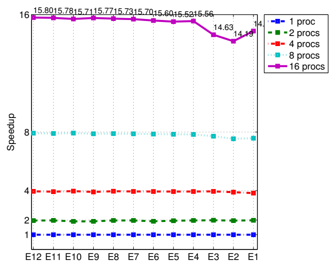
(a) Swarm size 30.



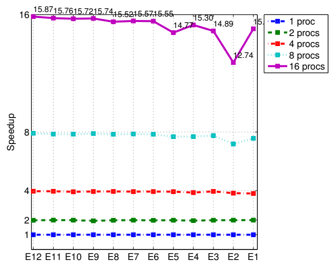
(b) Swarm size 50.



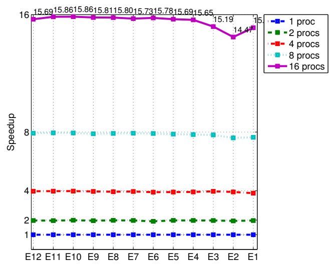
(c) Swarm size 100.



(d) Swarm size 50.

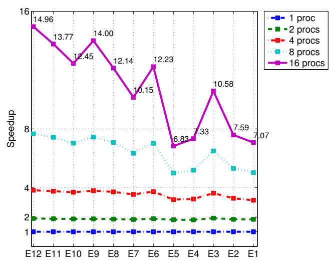


(e) Swarm size 30.

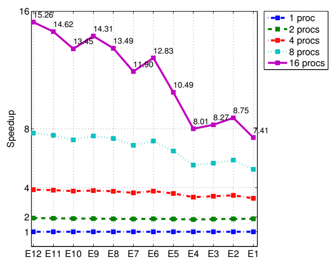


(f) Swarm size 100.

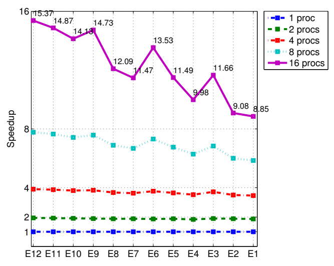
Fig. 3. Results with 1ms delay (first row) and 10ms delay (second row) using MDS.



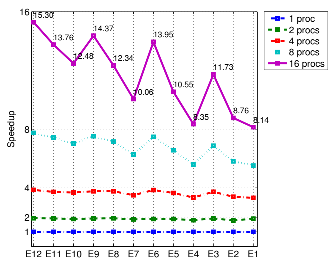
(a) Swarm size 30.



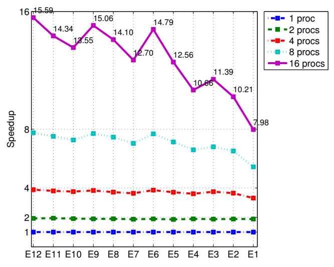
(b) Swarm size 50.



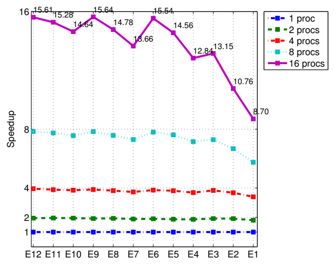
(c) Swarm size 100.



(d) Swarm size 30.

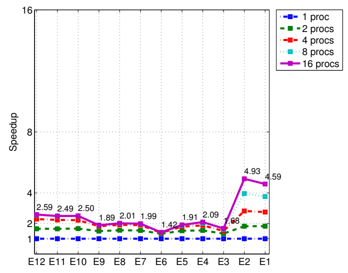


(e) Swarm size 50.

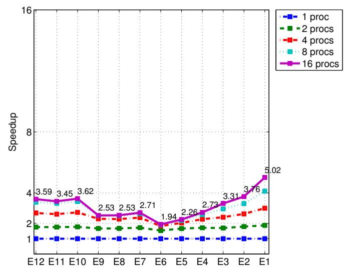


(f) Swarm size 100.

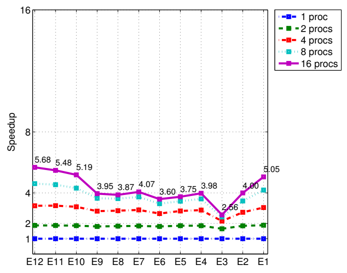
Fig. 4. Results with 1ms delay (first row) and 10ms (second row), using BFGS with numerical derivatives.



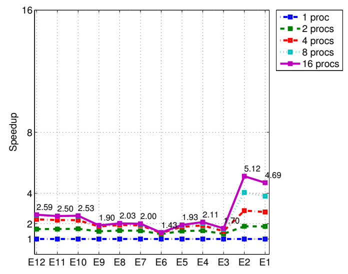
(a) Swarm size 30.



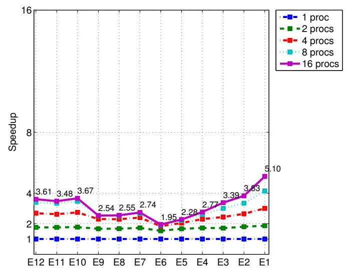
(b) Swarm size 50.



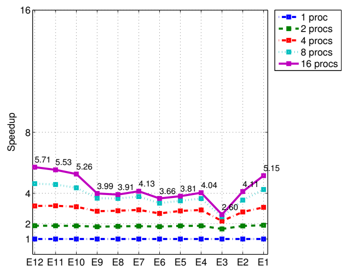
(c) Swarm size 100.



(d) Swarm size 30.



(e) Swarm size 50.



(f) Swarm size 100.

Fig. 5. Results with 1ms delay (first row) and 10ms delay (second row), using BFGS with analytic derivatives.

show increase with increasing swarm size. Furthermore, the speedup increases as a function of probability of local search and dimensionality. For all swarm sizes, experiment E12 (high dimensionality, high probability) exhibits better speedup that E1 (low dimensionality, low probability). The spikes in the 16-processor line for the 100-D cases E3, E6 and E9, indicate that the speedup is mainly affected by the dimensionality of the problem.

In Figure 5 we present results using the BFGS with analytical derivatives. As expected the serial nature of the original BFGS algorithm deteriorates the parallel efficiency of the whole scheme. The number of LS tasks (affected by the swarm size and the probability of LS) is not large enough to keep many processors busy, causing severe load imbalance. Notice that in the extreme case of E12, a mean number of $100 \cdot 0.2 = 20$ LS are created in every iteration. When 16 processors are available, these LS are distributed in two passes. In the first pass one LS per processor will be assigned (16 tasks) and that will leave the second pass with approximately 4 tasks remaining. So in the second pass the majority of the processors (75%) will remain idle. As as result, an average speedup of 4 is achieved when $4, 8$ or 16 processors are used.

However, additional experiments using BFGS with analytical derivatives on larger swarm sizes and dimensions, showed that the speedup is again approximating the ideal. The reason is the increasing number of concurrent local searches that are launched using these settings. In Fig. 6, we report the speedup for dimension $n = 100$, swarm size 1000 and 5000, respectively, and probability of LS $\rho = 0.2$ and 0.5.

The remaining test functions were also studied for the 12 configurations of Table 1, again using artificial delays of 1 and 10ms. The resulting charts are grouped together in a Supplementary Material section in the end this document. The reported speedup values in all cases, are similar to the ones reported for the Rastrigin function. The only significant difference occurs when BFGS with analytical derivatives was applied to the Griewank function (Fig. 16). The speedup reported in this figure is greater than the one reported for the Rastrigin, Ackley and Schwefel functions (Figs. 5, 13,19) and this can be attributed to the increased complexity of the Griewank objective.

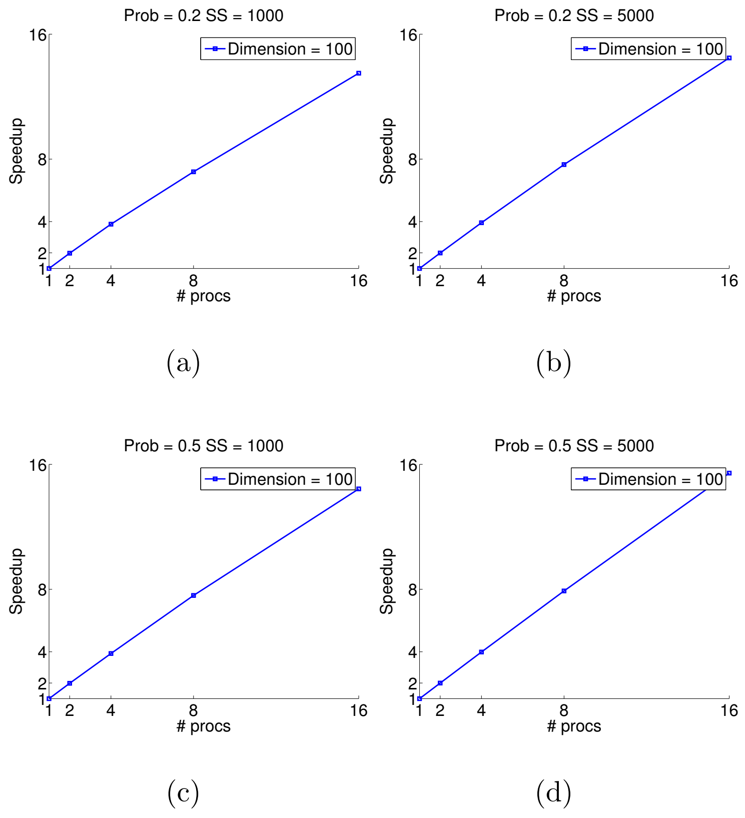


Fig. 6. Speedup using BFGS with analytic derivatives on the 100-dimensional Rastrigin test function.

5.2 Application on Atomic Clusters using Pairwise Energy Potentials

Pairwise energy potentials are mathematical models used to calculate the total energy in a cluster of atoms. Minimizing this energy with respect to the coordinates of the atoms, corresponds to finding a stable conformation of the cluster. In the case of pairwise potentials, the total energy of the system consisting of N_{atoms} atoms can be calculated as:

$$U_{tot} = \sum_{i=1}^{N} \sum_{j<i}^{N} E\left(r_{ij}\right) \tag{19}$$

Here the quantity $E\left(r_{ij}\right)$ is the pairwise interaction energy between the i-th and j-th atoms and

$$r_{ij} = \sqrt{\left(x_i - x_j\right)^2 + \left(y_i - y_j\right)^2 + \left(z_i - z_j\right)^2}$$

is the distance between atom i located at $(x_i, y_i, z_i)^\top$ and atom j located at $(x_j, y_j, z_j)^\top$. Different formulations for $E\left(r_{ij}\right)$ lead to different potentials and hence different conformations in space. Different pairwise energies to describe interactions for various classes of atoms have been proposed over the last years.

In the present work we use p-MEMPSODE to minimize three well known pairwise energy potentials (Lennard-Jones, Morse and Girifalco). The purpose of these benchmarks is to estimate the parallel efficiency of the software. Since the objective function is analytic and first order derivatives are available, we focused on the hybrid scheme that applies BFGS with analytical derivatives.

5.2.1 Lennard Jones Potential

The Lennard-Jones potential [51] is a simple model that approximates the interaction between a pair of neutral atoms or molecules (van der Waals interaction). It is considered a relatively good and universal approximation and due to its simplicity is often used to describe the properties of gases.

Formation of Lennard-Jones clusters have been extensively used as global optimization benchmark problems. The potential energy of the cluster is given by:

$$E(r_{ij}) = 4\,\varepsilon \left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6} \right], \tag{20}$$

where r_{ij} is the distance between atoms i and j. This function, for reduced units ($\sigma = \varepsilon = 1$), is implemented in the file `lj.c` of the p-MEMPSODE source code distribution.

We tested the speedup of p-MEMPSODE on a relatively large molecule of 100 atoms, which results in a problem of dimension $n = 300$. Each call for this specific instance of the Lennard-Jones objective function takes approximately 1 ms on a single core of the multicore server. We tested large swarm sizes (100, 500, and 1000) with probabilities set to 0.1, 0.2, and 0.3, respectively.

Figure 7 depicts the achieved speedups. We can easily notice that when 16 processors were utilized, the speedup ranged from 9.26 up to 11.07. This evidence is in accordance with the experiment results for the Rastrigin function with large swarm size and probability value (see Fig. 6). Again, it is apparent that larger swarms lead to more local searches and hence better speedup.

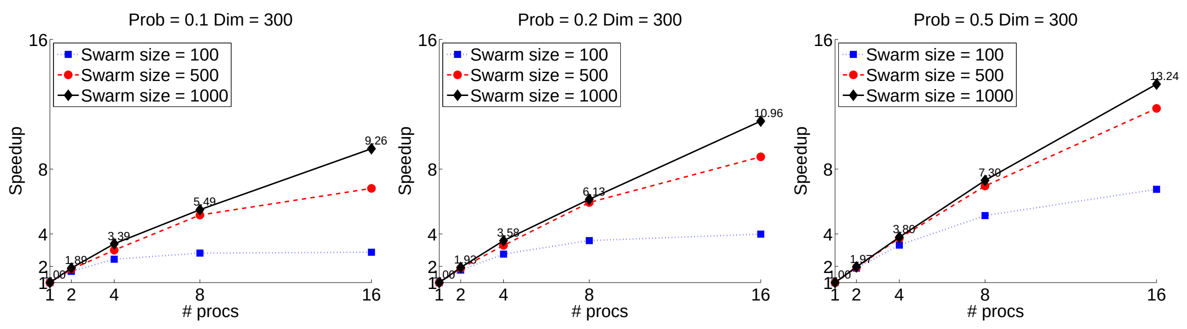


Fig. 7. Speedup for the Lennard-Jones potential: cluster of 100 atoms.

5.2.2 *Morse Potential*

The Morse potential [52] is a convenient model for the potential energy of diatomic molecules. Morse clusters are also considered a particularly tough test system for global optimization. The potential energy of the cluster is given by:

$$E(r_{ij}) = \epsilon \left[e^{-n\beta(r_{ij}-r_0)} - ne^{-\beta(r_{ij}-r_0)} \right] \qquad (21)$$

The parameter r_0 is the distance which corresponds to the minimum of the potential and $\epsilon(n-1)$ is the energy of the potential at its minimum. The parameters β and n define the steepness of the potential.

We measured the speedup of p-MEMPSODE using the same settings as in the Lennard-Jones case (100 atoms). The runtime of this instance of the objective function is approximately $1ms$. The results are shown in Figure 8 and are very similar to those observed for the Lennard-Jones potential.

5.2.3 *Girifalco Potential*

The Girifalco potential [53] was derived as an effective potential of the fullerene-fullerene interaction. A fullerene is a molecule composed of carbon in the form of a hollow sphere or ellipsoid. The formula of this potential is given by:

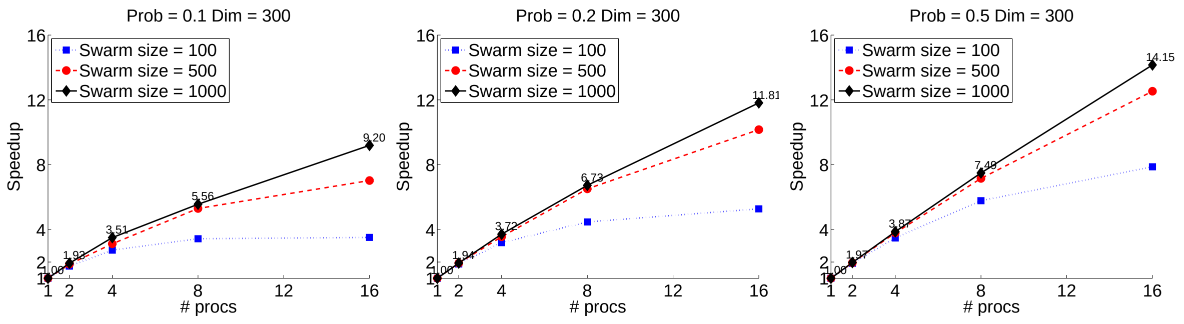


Fig. 8. Speedup for the Morse potential: cluster of 100 atoms.

$$E(r_{ij}) = -\alpha \left[\frac{1}{s_{ij}(s_{ij}-1)^3} + \frac{1}{s_{ij}(s_{ij}+1)^3} - \frac{2}{s_{ij}^4} \right] + \beta \left[\frac{1}{s_{ij}(s_{ij}-1)^9} + \frac{1}{s_{ij}(s_{ij}+1)^9} - \frac{2}{s_{ij}^{10}} \right], \tag{22}$$

where $s_{ij} = \dfrac{r_{ij}}{2a}$, $\alpha = \dfrac{N_c^2 A}{12(2a)^6}$, $\beta = \dfrac{N_c^2 B}{90(2a)^{12}}$. Here N_c is the number of carbon atoms of the fullerene, a defines the minimum position and the quantities $A = 19.97$ and $B = 34809$ are calculated empirically.

In this experiment we used p-MEMPSODE to calculate clusters of 100 fullerenes each one consisting of $N_c = 60$ carbon atoms. The results are shown in Figure 9. Although the same general trend is observed, the speedups are slightly lower in this case. Further analysis revealed that for the Girifalco potential, the BFGS algorithm exhibits larger variance in the number of steps required for convergence compared to the other two cases. That means that higher load imbalance is introduced by the BFGS, affecting the parallel performance of algorithm.

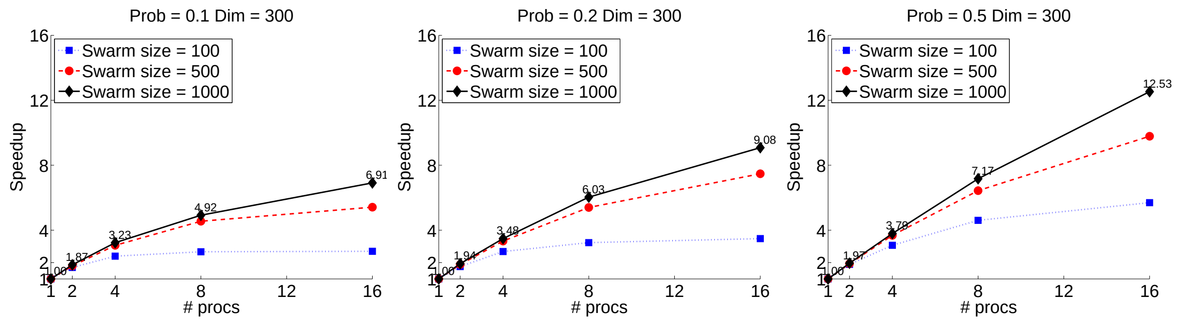


Fig. 9. Speedup for the Girifalco potential: cluster of 100 fullerenes

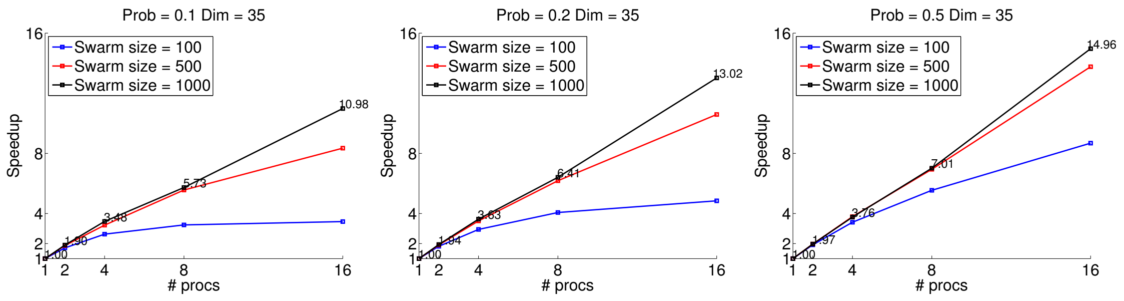


Fig. 10. Speedup for the Tinker potential application.

5.3 Application on Tinker Potential

The Tinker potential energy function was used in [29] to demonstrate the efficiency of the serial MEMPSODE version. In that example, the minimum energy conformation of a gas phase Alanine octamer was found. In the present work, we use the same example to measure the speedup of the p-MEMPSODE software.

The Tinker software package contains a Fortran implementation of the Amber [54] force field used in this study and in [29]. Tinker makes heavy use of common blocks for inter-subroutine communication. Since OpenMP threads share the same address space, concurrent Tinker function evaluations will be erroneous. To deal with this issue, we spawn separate Tinker server processes, one for each OpenMP thread. Each server calculates the same potential within its own memory space and communicates with p-MEMPSODE via Unix-domain sockets. This special implementation reduces the communication overhead while allowing concurrent Tinker evaluations to be issued by multiple OpenMP threads on a single computer system.

6 Conclusions

It this work, the parallel implementation of a hybrid global optimization algorithm that combines population based methods with local searches is proposed. The components of the hybrid algorithm were chosen mainly due to their inherent parallelization capabilities. The parallelization of the presented method results in an irregular, two-level task graph, considering function evaluations as basic computational tasks. The implementation is based on the OpenMP tasking model which provides seamless extraction and efficient execution of nested task-based parallelism. Thorough experimental testing on a server with 16 cores revealed that the proposed parallel implementation harnesses the power of multicore architectures and drastically reduces the execution time of the hybrid optimization algorithm.

Acknowledgments

This work is co-financed by the European Union and Greece Operational Program "Human Resources Development" - NSFR 2007-2013 - European Social Fund.

References

[1] R. Horst and P. M. Pardalos. *Handbook of Global Optimization.* Kluwer Academic Publishers, London, 1995.

[2] T. Bäck. *Evolutionary Algorithms in Theory and Practice.* Oxford University Press, New York, 1996.

[3] G. Rudolph. *Convergence Properties of Evolutionary Algorithms.* Verlag Dr. Kovač, Hamburg, 1997.

[4] K. E. Parsopoulos and M. N. Vrahatis. *Particle Swarm Optimization and Intelligence: Advances and Applications.* Information Science Publishing (IGI Global), 2010.

[5] J. Nocedal and S. J. Wright, editors. *Numerical Optimization.* Springer, 2006.

[6] A. Žilinskas and J. Žilinskas. A hybrid global optimization algorithm for non-linear least squares regression. *Journal of Global Optimization*, 56(2):265–277, 2013.

[7] R. Dawkins. *The Selfish Gene.* Oxford University Press, New York, 1976.

[8] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Technical Report C3P Report 826, Caltech Concurrent Computation Program, California, USA, 1989.

[9] P. Moscato. Memetic algorithms: A short introduction. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 219–235. McGraw-Hill, London, 1999.

[10] H.-G. Beyer. Evolutionary algorithms in noisy environments: Theoretical issues and guidelines for practice. *Comput. Methods Appl. Mech. Engrg.*, 186:239–269, 2000.

[11] E. Bonabeau, M. Dorigo, and G. Théraulaz. *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press, New York, 1999.

[12] C. A. Coello Coello, D. A. Van Veldhuizen, and G. B. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems.* Kluwer, New York, 2002.

[13] A. P. Engelbrecht. *Fundamentals of Computational Swarm Intelligence.* Wiley, 2006.

[14] J. Kennedy and R. C. Eberhart. *Swarm Intelligence.* Morgan Kaufmann Publishers, 2001.

[15] M. W. S. Land. *Evolutionary Algorithms with Local Search for Combinatorical Optimization.* PhD thesis, University of California, San Diego,USA, 1998.

[16] R. Fletcher. *Practical Methods of Optimization.* Wiley, New York, 1987.

[17] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization.* Academic Press, London, 1981.

[18] F. Neri, C. Cotta, and P. Moscato, editors. *Handbook of Memetic Algorithms.* Springer-Verlag, Berlin, 2012.

[19] Reiner Horst, Panos M Pardalos, and H Edwin Romeijn. *Handbook of global optimization*, volume 2. Springer, 2002.

[20] JJ Alonso, P LeGresley, and V Pereyra. Aircraft design optimization. *Mathematics and Computers in Simulation*, 79(6):1948–1958, 2009.

[21] John T Betts. Survey of numerical methods for trajectory optimization. *Journal of guidance, control, and dynamics*, 21(2):193–207, 1998.

[22] C. Audet, S. Le Digabel, and C. Tribes. NOMAD user guide. Technical Report G-2009-37, Les cahiers du GERAD, 2009.

[23] M.A. Abramson, C. Audet, G. Couture, J.E. Dennis, Jr., S. Le Digabel, and C. Tribes. The NOMAD project. Software available at `http://www.gerad.ca/nomad`.

[24] Dario Izzo, Marek Ruciński, and Francesco Biscani. The generalized island model. In *Parallel Architectures and Bioinspired Algorithms*, pages 151–169. Springer, 2012.

[25] Dario Izzo. Pygmo and pykep: Open source tools for massively parallel optimization in astrodynamics (the case of interplanetary trajectory optimization). In *Proceedings of the Fifth International Conference on Astrodynamics Tools and Techniques, ICATT*, 2012.

[26] Kejing He, Li Zheng, Shoubin Dong, Liqun Tang, Jianfeng Wu, and Chunmiao Zheng. Pgo: A parallel computing platform for global optimization based on genetic algorithm. *Computers & Geosciences*, 33(3):357–366, 2007.

[27] Jian He, Layne T Watson, and Masha Sosonkina. Algorithm 897: Vtdirect95: serial and parallel codes for the global optimization algorithm direct. *ACM Transactions on Mathematical Software (TOMS)*, 36(3):17, 2009.

[28] Christian L Müller, Benedikt Baumgartner, Georg Ofenbeck, Birte Schrader, and Ivo F Sbalzarini. pcmalib: a parallel fortran 90 library for the evolution strategy with covariance matrix adaptation. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1411–1418. ACM, 2009.

[29] C. Voglis, K. E. Parsopoulos, D. G. Papageorgiou, I. E. Lagaris, and M. N. Vrahatis. MEMPSODE: A global optimization software based on hybridization of population-based algorithms and local searches. *Computer Physics Communications*, 183(5):1139–1154, 2012.

[30] D. G. Papageorgiou, I. N. Demetropoulos, and I. E. Lagaris. Merlin-3.1. 1. a new version of the Merlin optimization environment. *Computer Physics Communications*, 159(1):70–71, 2004.

[31] Costas Voglis, Grigoris S Piperagkas, Konstantinos E Parsopoulos, Dimitris G Papageorgiou, and Isaac E Lagaris. Mempsode: comparing particle swarm optimization and differential evolution within a hybrid memetic global optimization framework. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, pages 253–260. ACM, 2012.

[32] Costas Voglis, Grigoris S Piperagkas, Konstantinos E Parsopoulos, Dimitris G Papageorgiou, and Isaac E Lagaris. Mempsode: An empirical assessment of local search algorithm impact on a memetic algorithm using noiseless testbed. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, pages 245–252. ACM, 2012.

[33] Costas Voglis. Adapt-mempsode: a memetic algorithm with adaptive selection of local searches. In *Proceeding of the fifteenth annual conference companion on Genetic and evolutionary computation conference companion*, pages 1137–1144. ACM, 2013.

[34] Costas Voglis, Panagiotis E Hadjidoukas, Konstantinos E Parsopoulos, Dimitrios G Papageorgiou, and Isaac E Lagaris. Adaptive memetic particle swarm optimization with variable local search pool size. In *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 113–120. ACM, 2013.

[35] V. Torczon. *A Direct Search Algorithm for Parallel Machines.* PhD thesis, Department of Mathematical Sciences, Rice University, Houston, U. S. A., 1989.

[36] OpenMP Architecture Review Board. Openmp specifications. Available at: `http://www.openmp.org`.

[37] J. Kennedy and R. C. Eberhart. Particle swarm optimization. In *Proc. IEEE Int. Conf. Neural Networks*, volume IV, pages 1942–1948, Piscataway, NJ, 1995. IEEE Service Center.

[38] R. C. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *Proceedings Sixth Symposium on Micro Machine and Human Science*, pages 39–43, Piscataway, NJ, 1995. IEEE Service Center.

[39] P. N. Suganthan. Particle swarm optimizer with neighborhood operator. In *Proc. IEEE Congr. Evol. Comput.*, pages 1958–1961, Washington, D.C., USA, 1999.

[40] M. Clerc and J. Kennedy. The particle swarm-explosion, stability, and convergence in a multidimensional complex space. *IEEE Trans. Evol. Comput.*, 6(1):58–73, 2002.

[41] K. E. Parsopoulos and M. N. Vrahatis. UPSO: A unified particle swarm optimization scheme. In *Lecture Series on Computer and Computational Sciences, Vol. 1, Proceedings of the International Conference of Computational Methods in Sciences and Engineering (ICCMSE 2004)*, pages 868–873. VSP International Science Publishers, Zeist, The Netherlands, 2004.

[42] K. E. Parsopoulos and M. N. Vrahatis. Parameter selection and adaptation in unified particle swarm optimization. *Mathematical and Computer Modelling*, 46(1-2):198–213, 2007.

[43] R. Storn and K. Price. Differential evolution-a simple and efficient heuristic for global optimization over continuous spaces. *J. Global Optimization*, 11:341–359, 1997.

[44] K. Price. Differential evolution: A fast and simple numerical optimizer. In *Proceedings NAFIPS'96*, pages 524–525, 1996.

[45] V. Torczon. On the convergence of the multidimensional search algorithm. *SIAM J. Optimization*, 1:123–145, 1991.

[46] Y. G. Petalas, K. E. Parsopoulos, and M. N. Vrahatis. Memetic particle swarm optimization. *Annals of Operations Research*, 156(1):99–127, 2007.

[47] W. E. Hart. *Adaptive Global Optimization with Local Search.* PhD thesis, University of California, San Diego,USA, 1994.

[48] C. Voglis, K. E. Parsopoulos, and I. E. Lagaris. Particle swarm optimization with deliberate loss of information. *Soft Computing*, 16(8):1373–1392, 2012.

[49] C. Voglis, PE Hadjidoukas, IE Lagaris, and DG Papageorgiou. A numerical differentiation library exploiting parallel architectures. *Computer Physics Communications*, 180(8):1404–1415, 2009.

[50] J.W. Ponder et al. TINKER: software tools for molecular design. *Department of Biochemistry and Molecular Biophysics, Washington University School of Medicine, St. Louis, MO*, 1998.

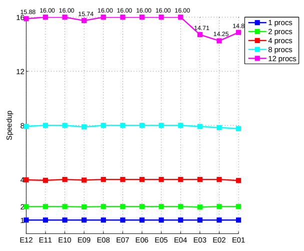
[51] J. Lennard-Jones. On the determination of molecular fields. *Proc. R. Soc. Lond. A.*, 47(6):106–463, 1924.

[52] P. Morse. Diatomic molecules according to the wave mechanics ii. vibrational levels. *Proc. R. Soc. Lond. A.*, 34:57–64, 1929.

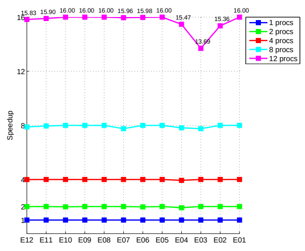
[53] L. Girifalco. Molecular properties of fullerene in the gas and solid phases. *J. Phys. Chem.*, 96:858–861, 1992.

[54] W.D. Cornell, P. Cieplak, C.I. Bayly, I.R. Gould, K.M. Merz, D.M. Ferguson, D.C. Spellmeyer, T. Fox, J.W. Caldwell, and P.A. Kollman. A second generation force field for the simulation of proteins, nucleic acids, and organic molecules. *Journal of the American Chemical Society*, 117(19):5179–5197, 1995.

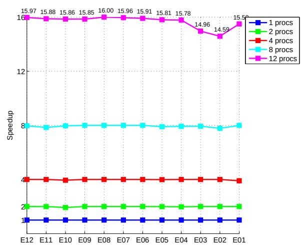
Supplementary Material



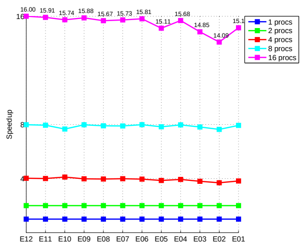
(a) Swarm size 30.



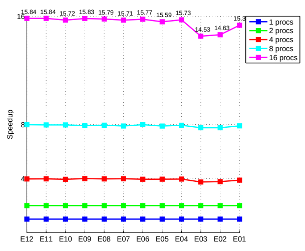
(b) Swarm size 50.



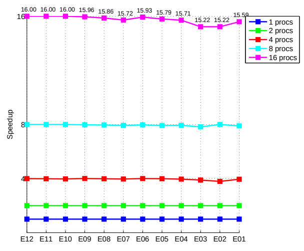
(c) Swarm size 100.



(d) Swarm size 50.

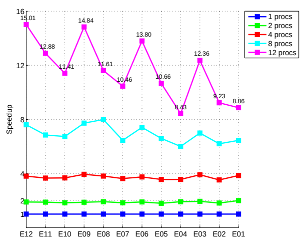


(e) Swarm size 30.

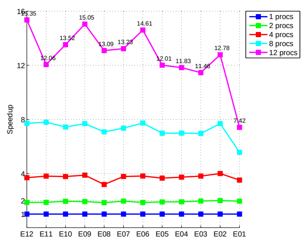


(f) Swarm size 100.

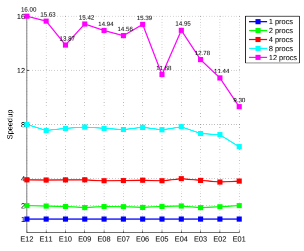
Fig. 11. Ackley function results with 1ms delay (first row) and 10ms delay (second row) using MDS.



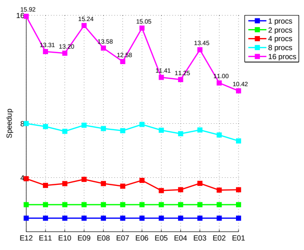
(a) Swarm size 30.



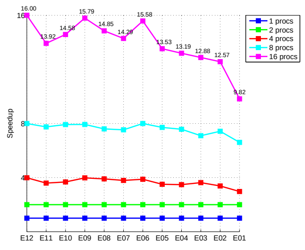
(b) Swarm size 50.



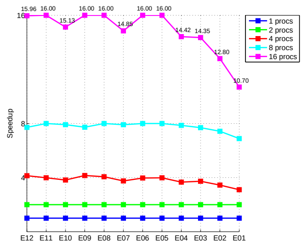
(c) Swarm size 100.



(d) Swarm size 30.

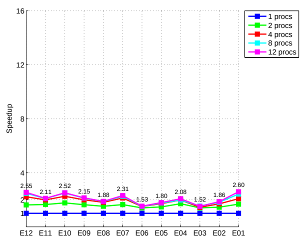
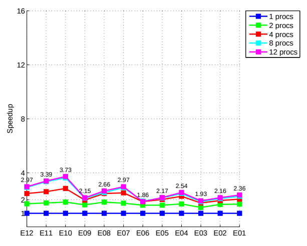
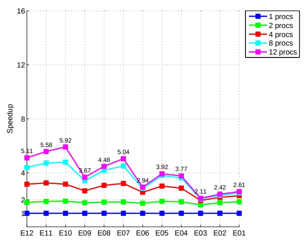


(e) Swarm size 50.

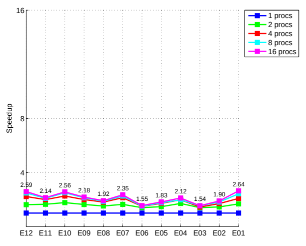
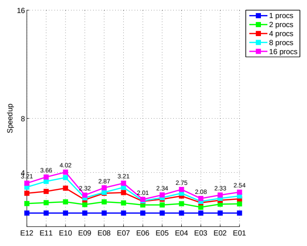
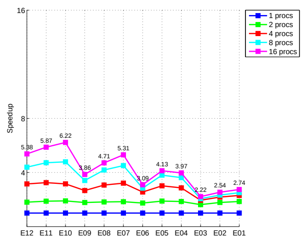


(f) Swarm size 100.

Fig. 12. Ackley function results with 1ms delay (first row) and 10ms (second row), using BFGS with numerical derivatives.

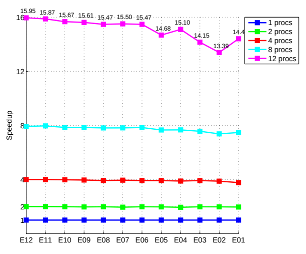
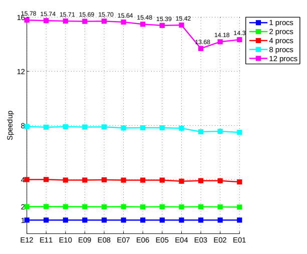
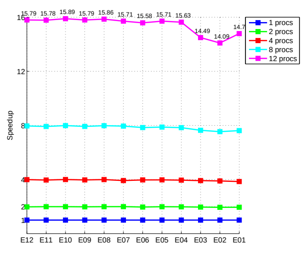




(a) Swarm size 30. (b) Swarm size 50. (c) Swarm size 100.

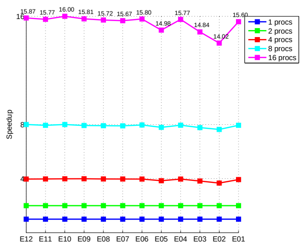
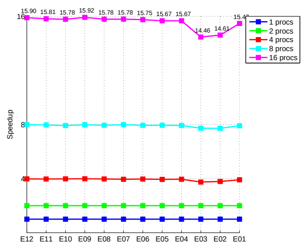
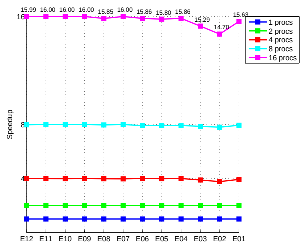




(d) Swarm size 30. (e) Swarm size 50. (f) Swarm size 100.

Fig. 13. Ackley function results with 1ms delay (first row) and 10ms delay (second row), using BFGS with analytic derivatives.

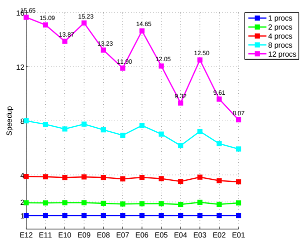




(a) Swarm size 30. (b) Swarm size 50. (c) Swarm size 100.

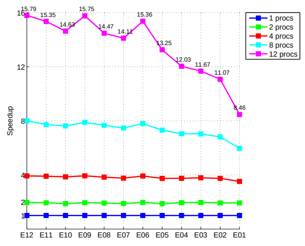




(d) Swarm size 50. (e) Swarm size 30. (f) Swarm size 100.

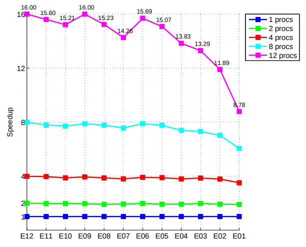
Fig. 14. Griewank function results with 1ms delay (first row) and 10ms delay (second row) using MDS.



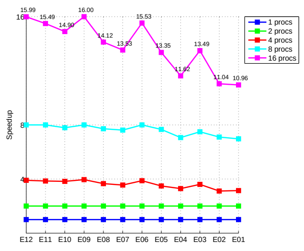
(a) Swarm size 30.



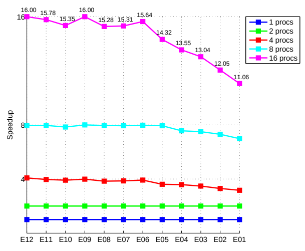
(b) Swarm size 50.



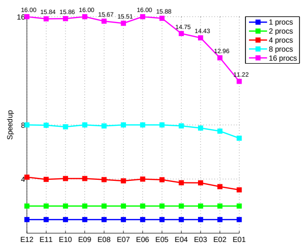
(c) Swarm size 100.



(d) Swarm size 30.

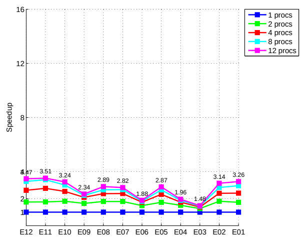


(e) Swarm size 50.

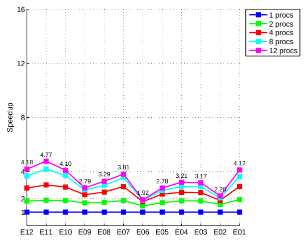


(f) Swarm size 100.

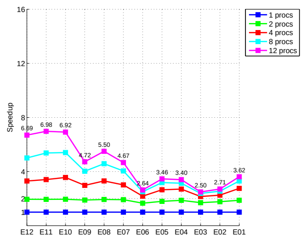
Fig. 15. Griewank function results with 1ms delay (first row) and 10ms (second row), using BFGS with numerical derivatives.



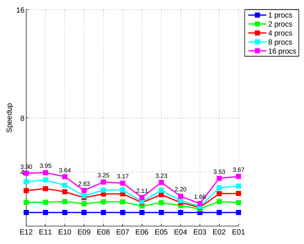
(a) Swarm size 30.



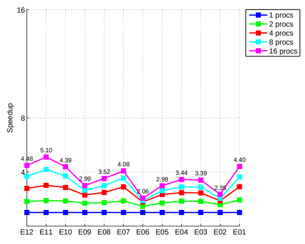
(b) Swarm size 50.



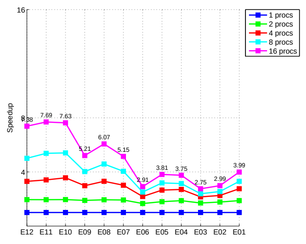
(c) Swarm size 100.



(d) Swarm size 30.

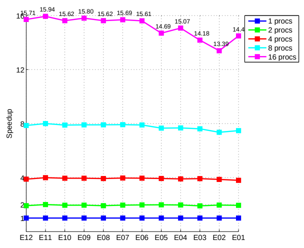


(e) Swarm size 50.

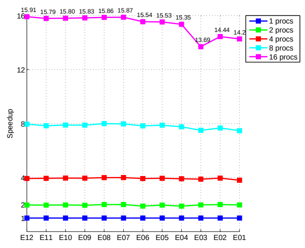


(f) Swarm size 100.

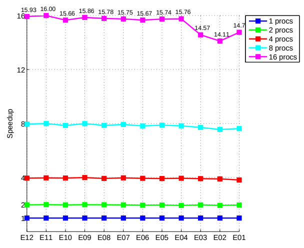
Fig. 16. Griewank function results with 1ms delay (first row) and 10ms delay (second row), using BFGS with analytic derivatives.



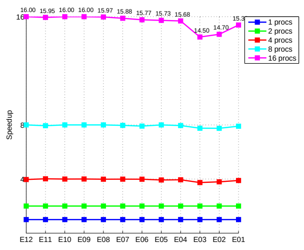
(a) Swarm size 30.



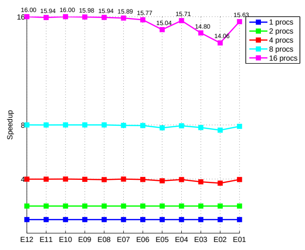
(b) Swarm size 50.



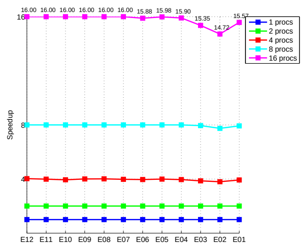
(c) Swarm size 100.



(d) Swarm size 50.

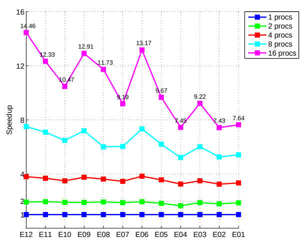


(e) Swarm size 30.

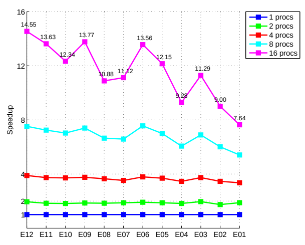


(f) Swarm size 100.

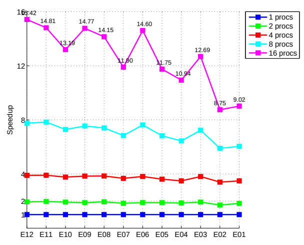
Fig. 17. Schwefel function results with 1ms delay (first row) and 10ms delay (second row) using MDS.



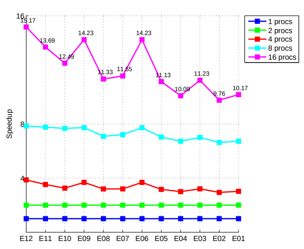
(a) Swarm size 30.



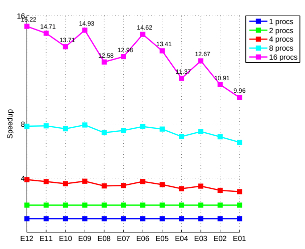
(b) Swarm size 50.



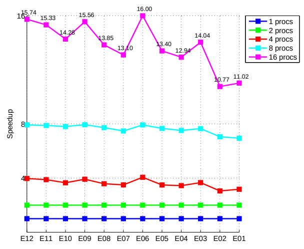
(c) Swarm size 100.



(d) Swarm size 30.

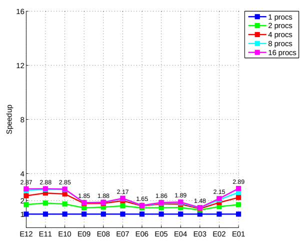


(e) Swarm size 50.

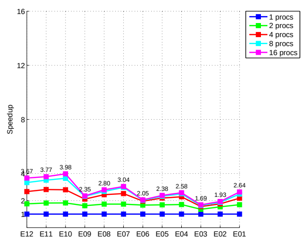


(f) Swarm size 100.

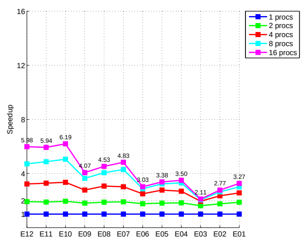
Fig. 18. Schwefel function results with 1ms delay (first row) and 10ms (second row), using BFGS with numerical derivatives.



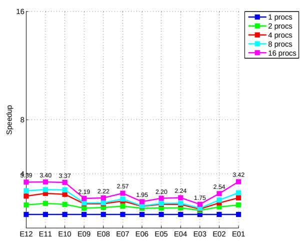
(a) Swarm size 30.



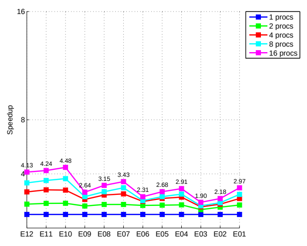
(b) Swarm size 50.



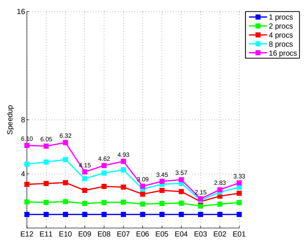
(c) Swarm size 100.



(d) Swarm size 30.



(e) Swarm size 50.



(f) Swarm size 100.

Fig. 19. Schwefel function results with 1ms delay (first row) and 10ms delay (second row), using BFGS with analytic derivatives.