Enhanced Multiobjective Population-Based Incremental Learning with Applications in Risk Treaty Optimization

Omar Andres Carmona Cortes[1](✉) and Andrew Rau-Chaplin[2]

[1] Informatics Department, Instituto Federal Do Maranhão, São Luis, MA, Brazil
omar@ifma.edu.br

[2] Risk Analytics Lab, Dalhousie University, Halifax, NS, Canada
arc@cs.dal.ca

Abstract. The purpose of this paper is to revisit the Multiobjective Population-Based Incremental Learning method and show how its performance can be improved in the context of a real-world financial optimization problem. The proposed enhancements lead to both better performance and improvements in the quality of solutions. Its performance was assessed in terms of runtime and speedup when parallelized. Also, metrics such as the average number of solutions, the average hypervolume, and coverage have been used in order to compare the Pareto frontiers obtained by both the original and enhanced methods. Results indicated that the proposed method is 22.1 % faster, present more solutions in the average (better defining the Pareto frontier) and often generates solutions having larger hypervolumes. The enhanced method achieves a speedup of 15.7 on 16 cores of a dual socket Intel multi-core machine when solving a Reinsurance Contract Optimization problem involving 15 Layers or sub-contracts.

Keywords: PBIL · Multiobjective optimization · Risk · Reinsurance

1 Introduction

Performance plays an important role in search algorithms because the more complex the problem, the harder discovering solutions in feasible time. This assertion is true especially in the industry where timely answers are required regardless the complexity of the search space. Thus, evolutionary algorithms, also known as search heuristics, can be a highly effective choice.

There are many heuristics search methods that can be applied to solve real-world problems, including Particle Swarm Optimization (PSO) [1], Differential Evolution (DE) [2], Genetic Algorithms (GA) [3], Evolution Strategies (ES) [4] and Population-Based Incremental Learning (PBIL) [5]. Whereas most heuristics search methods apply their genetic operations on a population of individual solutions, PBIL executes its operators in a special data structure called probability matrix that is responsible for creating the population in each iteration.

© Springer International Publishing Switzerland 2016
G. Squillero and P. Burelli (Eds.): EvoApplications 2016, Part I, LNCS 9597, pp. 3–18, 2016.
DOI: 10.1007/978-3-319-31204-0_1

Doing so, PBIL tends to be faster than other heuristic search methods, especially in discrete search spaces.

The first version of PBIL was introduced in 1994 in [5]. At that time, the probability matrix was only a vector where each position represented the probability of having a 0 or 1. The closer the probability to 1, the bigger the chance of creating this gene, while the opposite meant the chance of having a 0. Since, extensions have been proposed for continuous and base-n represented search spaces [6–8]. A version for a discrete space in the range [0, 1] was proposed in [9] and compared against DE and PSO in [10], showing that PBIL is often a very attractive heuristic search method. In [11], a multi-objective-based version of PBIL, called MOPBIL, was designed and applied to problems in reinsurance analytics.

An important problem in Reinsurance analytics is the Reinsurance Contract Optimization problem, where given the structure of a multi-layered reinsurance contract, we need to discover the best trade-offs between expected return and risk for the primary insurer. Such optimizations are key to developing the reinsurance risk hedging strategies that are so important in financial risk management [12].

In this paper we propose E-MOPBIL (Enhanced MOPBIL) which contains important enhancements to MOPBIL and apply it to the Reinsurance Contract Optimization problem in order to achieve faster and higher quality solutions. The Reinsurance Contract Optimization problem consists of, given a reinsurance contract formed by a fixed number of layers (subcontracts) and a simulated set of expected loss distributions (one per layer), plus a model of reinsurance costs, identifying optimal combinations of placements (i.e. percentage shares) in order to maximize the expected return while the associated risk value is minimized [11].

The remainder of this paper is organized as follows: Sect. 2 describes the risk optimization problem being studied in this work; Sect. 3 thoroughly explains the MOPBIL algorithm and the enhanced E-MOPBIL algorithm; Sect. 4 shows the comparative experiments; finally, Sect. 5 presents the conclusions and future work.

2 The RCO Problem

The reinsurance contract optimization problem is a particular kind of treaty optimization problem, which consists of a fixed number of contractual layers and a simulated set of expected loss distributions (one per layer), plus a model of reinsurance market costs [9]. Taking this into consideration, the task is to identify optimal combinations of shares (also called placements) in order to build a Pareto frontier that quantifies the best available trade-offs between expected return and risk [13]. In other words, insurance companies aim hedge their risk against potentially large claims, or losses [14]. Having these trade-offs the insurance companies are able to offer them to the reinsurance market.

The final purpose is to maximize the amount of expected return ($) received from the reinsurance company in case of massive claims and maximize the risk hedge to the reinsurance company at the same time. Doing so, the insurance company minimize the loss faced per year due to natural catastrophes.

In this context, (1) represents the problem in terms of optimization, where VaR is a risk metric, $\mathbf{R}$ is a function in term of placements (π) and E is the Expected Value[1]. For further details about the problems refer to [9] and [14].

$$\begin{array}{ll} maximize & f_1(x) = VaR_\alpha(\mathbf{R}(\pi)) \\ maximize & f_2(x) = E[\mathbf{R}(\pi)] \end{array} \tag{1}$$

3 MOPBIL

In this section we thoroughly describe the algorithm MOPBIL which is presented in Algorithm 1. The main inputs of the algorithm are the following parameters: number of iterations (N_G), number of best individuals and the slice size. The number of iterations is common in evolutionary algorithms, being one of the stop criteria can be adopted. The second one regards to the number of individuals who comprises the best population, which will be used to mutate the probability matrix. The last parameter is important to define the discretization of the search space. The smaller the slice size, the bigger the search space.

The algorithm starts estimating both the minimum and the maximum of the mean (expected return) in order to determine the interval of each slab. Actually, this is done by dividing the search space. One of the consequences of doing so is the possibility of parallelized the algorithm; however, in the original version the number of iterations is maintained; therefore, it is difficult to obtain speedup. On the other hand, the main idea behind this is to enhance the quality of solutions.

Then, when the $foreach$ loop starts each slab produces a new population according to the probability matrix. The process is similar to the roulette wheel selection from genetic algorithms which is in essence a Monte Carlo simulation. For example, if the valid shares belong to the set {0, 0.1, 0.2, ..., 1.0}, a column will consist of 11 equal probabilities, thus if a contract is formed by 7 layers the probability matrix will be a matrix consisting of 11 rows and 7 columns with uniform probability. Each position in the matrix is called bucket and will be updated later in the algorithm in the mutation part of the algorithm. The number of buckets varies according to the slice size.

Afterward, new individuals are created according to the probability matrix and evaluated. The fitness is obtained firstly evaluating the expected return. If the expected return is outside from its boundaries $[min, max]$ the risk is automatically set to zero. Given the evaluations, the new population is merged with the archive in order to identify all non-dominated solutions, rank and cluster them. All non-dominated individuals within the valid interval are considered valid and raked as 1. Then, valid dominated solutions are ranked as 2. Finally, individuals outside the limits are ranked ranging from 3 to *pop_size* according to the distance from the valid interval. The clustering is done using the current

[1] In probability theory, the expected value, usually denoted by E[X], refers to the value of a random variable X that we would "expect" to find out if we could repeat the random variable process an infinite number of times and take the average of the values obtained.

```
Input: N_G = number of generations; n_best = number of best individuals;
  slice_size=discretization;
Estimate the min and the max of the mean;
Divide the interval [min, max] into n slabs;
foreach slab do
    while (N_G not reached) do
        Create the population using probability matrix;
        Use the mean to determine the risk value;
        if (mean of one individual is outside the chunk) then
            risk_value = 0;
        else
            risk_value = compute(mean)
        end
        Merge archive and the new population; Determine the non-dominated
          set;
        Cluster the non-dominated set into k clusters;
        Select k representative individuals;
        Select worst individuals;
        Insert the k individuals into the best population;
        Identify the best and worst buckets;
        Update and mutate the probability matrix;
    end
end
Combine the results of each slab;
Determine the Pareto frontier;
```

Algorithm 1. MOPBIL (Sketch)

Pareto frontier. If more than one individual belongs to the same cluster then the best one is selected to the $best_n$ sub-population based on the best risk, *i.e.*, that one which hedge more risk goes to the $best_n$ sub-population; however, if there are not enough non-dominated solutions then the $best_n$ set is filled up with valid dominated solutions ($rank = 2$) or with the best ranked invalid solutions in the worst case scenario.

Having identified the best sub-population it is necessary to determine the best and worse buckets from each individual in order to mutate the probability matrix. The mutation is done in two steps. Firstly, a probability multiplier is computed based on both best and worst buckets according to the Algorithm 2, where roughly speaking if a best bucket coincides with a worst buckets the multiplier increases at a lower rate; otherwise, the multiplier might be larger than 1.

After the first step the probability matrix has to be normalized because the sum of a column can be either larger or lower than 1. Secondly, the probability matrix is mutated according to the Algorithm 3. Finally, the matrix has to be normalized again.

```
for (n=1 to rows in prob_matrix) do
    prob_multiplier ← 0 ;
    for (j = 1 to length(best_bucktes)) do
        if (worst_buckets == best_buckets_j) then
            prob_multiplier ← prob.multiplier + (1/length(best.buckets)) *
              ((1-n.learn.rate) + (4 * n.learn.rate)/n.buckets * abs( abs(n -
              best.buckets[j]) - n.buckets/2))
        else
            prob.multiplier ← prob.multiplier + (1/length(best.buckets)) *
              ((1-n.learn.rate2) + (4 * n.learn.rate2)/n.buckets * abs( abs(n -
              best.buckets[j]) - n.buckets/2))
        end
    end
    prob.matrix[n,j] ← prob.matrix[n,i]*prob.multiplier
end
```

Algorithm 2. Computing probability multiplier

```
for j = 1 to n.par do
    for i = 1 to I do
        mut.dir ← 0.0;
        if random(0,1) ≤ M_R then
            if random(0,1) ≤ 0.5 then
                mut.dir ← 1.0;
            end
            p_ij ← p_ij(1 − mut.shift) + mut.dir ∗ mut.shift
        end
    end
end
```

Algorithm 3. Mutation

The mutation process showed in the Algorithms 2 and 3 can be represented by (2), where LF_{ijk} is the ith learning factor, as described in [6], for the kth best result for the jth variable.

$$p_{ij}^{NEW} = \sum_{k=1}^{q} p_{ij} \frac{LF_{ijk}}{q} \tag{2}$$

3.1 Our Proposal: E-MOPBIL

Our first modification is to remove slabs. As a result, we do not have to compute boundaries which in fact is not time-consuming; however, now we do not have to deal with invalid points, *i.e.*, it is not necessary to rank them which demands to compute the distance between points and boundaries. Moreover, using boundaries as we increase the number of slabs we also increase the probability of creating invalid points affecting the quality of solutions in a parallel execution. The second modification regards to creating the best sub-population

($best_n$). Tests have demonstrated that the Pareto frontier formed by merging the new population and the archive on each iteration is enough to build the $best_n$ sub-population. Thirdly, we do not compute the worse buckets because it might reduce the increment on the probability of a promise bucket if they are the same; therefore, we use the Algorithm 4 for computing the probability multiplier. So, the final version is presented in the Algorithm 5.

```
for (n=1 to rows in prob_matrix) do
    prob_multiplier ← 0 ;
    for (j = 1 to length(best_bucktes)) do
        prob_multiplier ← prob_multiplier + (1/length(best.buckets)) *
          ((1-n.learn.rate2) + (4 * n.learn.rate2)/n.buckets * abs( abs(n -
          best.buckets[j]) - n.buckets/2))
    end
    prob.matrix[n,j] ← prob.matrix[n,i]*prob_multiplier
end
```

Algorithm 4. Computing new probability multiplier

```
Input: N_G = number of generations; n_best = number of best individuals;
while (N_G not reached) do
    Create the population using probability matrix;
    Evaluate objective function on each member of the population;
    Merge archive and the new population;
    Determine the non-dominated set;
    Cluster the non-dominated set into k clusters;
    Select k representative individuals;
    Insert the k individuals into the best population;
    Update and mutate the probability matrix;
end
Determine the final Pareto frontier;
```

Algorithm 5. E-MOPBIL (Sketch based on our proposal)

As previously stated, MOPIBIL was designed for getting the best quality in terms of solutions dividing the search space into slabs. Doing so, the algorithm does not lead to an improvement regarding speedup when executed in more than one core. In our approach, we parallelized the iteration loop as illustrated in Algorithm 6, which means that the number of iterations is divided between processor units, thus the process is similar to that one presented in serial E-MOPBIL (Algorithm 5).

In order to make a fair comparison against our proposal of parallelizing the code, we also divided the number of iterations between slabs. Thus, the same number of calls to the evaluation function is done in both algorithms.

Input: N_G = number of generations; n_{best} = number of best individuals;
foreach *(N_G/#n_threads)* ***do*** *in parallel* **do**
  Create the population using probability matrix;
  Evaluate objective function on each member of the population;
  Merge archive and the new population;
  Determine the non-dominated set;
  Cluster the non-dominated set into k clusters;
  Select k representative individuals;
  Insert the k individuals into the best population;
  Identify the best buckets;
  Update and mutate the probability matrix;
end
Combine the results of each chunk of iterations;
Determine the Pareto frontier;

Algorithm 6. Parallel E-MOPBIL

4 Computational Experiments

4.1 Setup and Metrics

All tests were conducted using R version 3.2.1 on a Red Hat Linux 64-bit Operating in an Intel Xeon comprising of two Xeon processors E5-2650 running at 2.0 Ghz with 8 cores and hyper threading and 256 GB of memory. Considering 250 and 500 with a population size equals to 50. The following parameters were used:

- Population size = 50;
- Slice size = 0.05;
- Number of generations = 250, 500
- Best population = 3;
- learn.rate = 0.1,
- neg.learn.rate = 0.075,
- mut.prob = 0.02,
- mut.shift = 0.05,

In order to compare the algorithms we used the following metrics: number of solutions, hypervolume, coverage and generational distance. In the first one, the larger the number of solution the better the results tends to be; however, this metric is not enough to compare Pareto frontiers. All averages are calculated in 30 trials, allowing us to make inferences based on t-test.

The hypervolume depicts the volume of the dominated part of the curve, therefore, the bigger the hypervolume, the better the Pareto frontier might be. Mathematically, the hypervolume can be computed by (3).

$$hv = volume\left(\bigcup_{i=1}^{|Q|} v_i\right) \tag{3}$$

The coverage represents the percentage of solutions which dominate at least one of the other solutions. Roughly speaking, $C(A, B)$ is the percentage of the solutions in B that are dominated by at least 1 solution in A [15], therefore, if $C(A, B) = 1$ then all solutions in A dominate B, and $C(A, B) = 0$ means the opposite. The coverage can be calculates as shown in (4).

$$C(A, B) = \frac{|\{b \in B | \exists a \in A : a \preceq b\}|}{|B|} \tag{4}$$

In terms of parallel computing the performance was evaluated using speedup, particularly we used the weak speedup depicted in (5) and suggested in [16], where T_1 is the time of the serial version and T_p is the time for running the code in p processor units. We used this kind of speedup because T_1 is the time for running the code in 1 thread, therefore, we do not need to guarantee that T_1 is obtained using the best possible implementation.

$$Speedup = \frac{T_1}{T_p} \tag{5}$$

4.2 Results

Figure 1 shows the final Pareto frontier obtained by E-MOPBIL and MOPBIL using 250 and 500 iterations in a 7-layered problem, in which the closer to zero the better the solutions. Visually, E-MOPBIL presents better results in both configurations which will be confirmed by metrics in Tables 1 and 2.

Table 1 shows metrics for 250 iterations and 7 layers in which we can see that the new algorithm is faster and presents better number of solutions, hypervolume and final number of solutions in the final Pareto frontier. In fact, the E-MOPBIL is 22.8 % faster than the original algorithm. A two-tailed t-test with an $alpha = 0.05$ of significance demonstrated that the differences are significant in the metrics number of solutions and time. Moreover, the coverage metric indicates that E-MOPBIL dominates 83 % of solutions from original MOPBIL against 3.5 % in the opposite direction.

Results regarding 500 iterations and 7 layers are shown in Table 2 where we can see that E-MOPBIL achieved better results in all metrics. Actually, the new algorithm is 21.5 % faster than the original MOPBIL, reaching better results in all metrics as demonstrated by the two-tailed t-test with an $alpha = 0.05$ of significance. The coverage indicates that E-MOPBIL dominates 83 % of solutions from the original algorithm instead of 1.4 % in the way around.

Figure 2 shows the final Pareto frontier for 250 and 500 iterations solving a 15-layered problem. Clearly, our proposal overcomes the original MOPBIL in both configurations. Tables 3 and 4 confirm this assertion.

A two-tailed t-test with an $alpha = 0.05$ of significance demonstrates that the differences between both algorithms are significant in the metrics number of solutions and time in both configurations. Actually, using 15 layers our proposal is 11.5 % faster and discover about twice the number of solutions per trial. In terms of coverage and 250 iterations, E-MOPBIL dominates 90.4 % of solutions

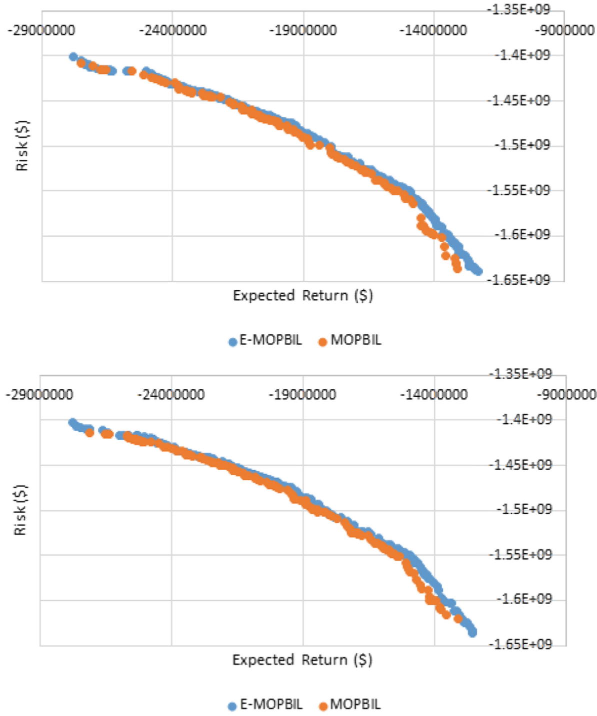

Fig. 1. Comparison between E-MOPBIL and MOPBIL for 250 and 500 iterations with 7 layers

Table 1. Metrics for 250 iterations and 7 layers

MOPBIL				
	#NS	HV	Time (s)	Final #NS
Avg	63.70	1.66E+15	201.45	114
Std	6.29	1.56E+14	2.43	-
E-MOPBIL				
	#NS	HV	Time (s)	Final #NS
Avg	**119.77**	**1.79E+15**	**162.53**	**218**
Std	20.24	2.66E+14	2.63	-
t	**-14.48**	-2.19	**59.53**	-

from MOPBIL, while in the opposite direction MOPBIL does not domain any solution from our proposal. Considering 500 iterations, E-MOPBIL dominates 87.2 % of solutions against 2.2 % from MOPBIL.

Table 2. Metrics for 500 iterations and 7 layers

MOPBIL				
	#NS	HV	Time (s)	Final #NS
Avg	75.63	1.74E+15	406.96	131
Std	5.33	1.13E+14	2.84	-
E-MOPBIL				
	#NS	HV	Time (s)	Final #NS
Avg	**122.6**	**1.90E+15**	**325.85**	**213**
Std	16.31	2.03E+14	6.55	-
t	**-14.35**	**-3.96**	**62.20**	-

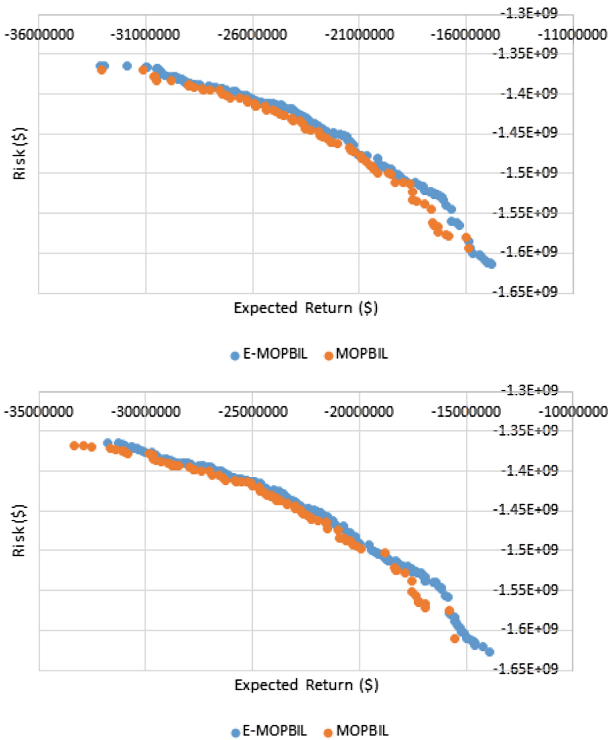

Fig. 2. Comparison between E-MOPBIL and MOPBIL for 250 and 500 iterations with 15 layers

4.3 Parallel Experiments

As previously stated, the original MOPBIL does not properly divided the tasks among workers, which means that all processor elements do the same amount of job. For instance, if we start an optimization using 250 iterations and 4 slabs

Table 3. Metrics for 250 iterations and 15 layers

MOPBIL				
	#NS	HV	Time (s)	Final #NS
Avg	48.30	2.039E+15	408.96	74
Std	5.00	2.60E+14	2.39	-
E-MOPBIL				
	#NS	HV	Time (s)	Final #NS
Avg	**100.67**	**2.12E+15**	**366.64**	178
Std	13.76	2.58E+14	3.15	-
t	**-19.59**	-1.29	**58.61**	-

Table 4. Metrics for 500 iterations and 15 layers

MOPBIL				
	#NS	HV	Time (s)	Final #NS
Avg	52.87	2.23E+15	823.30	87
Std	5.53	2.50E+14	7.58	-
E-MOPBIL				
	#NS	HV	Time (s)	Final #NS
Avg	**100.3**	2.07E+15	**736.47**	**198**
Std	19.47	3.67E+14	26.37	-
t	**-12.83**	1.93	**17.34**	-

(consequently 4 threads) the time will be in the average the same if we execute an optimization with the same number of iterations but using only 1 slab (1 thread). Experiments have demonstrated that time remains about 200 seconds on each trial regardless the number of slabs for 7 layers and 408 seconds for 15 layers, using 250 iterations in both cases.

The E-MOPBIL divides the number of iterations between threads. Thus, if we run an optimization with 500 iterations and 4 threads, each one will execute 125 iterations. In this context, Fig. 3 shows the speedup reached as we increase the thread count up to 16 threads in which we can see the reached speedup is similar up to 4 threads and from this point on the speedup is better using 500 iterations.

Figure 4 illustrates the speedup reached by the experiment running 250 and 500 iterations in a 15-layered problem, where we can observe that the performance is similar until about 8 threads, after that using 500 iterations tends to present better speedups.

Figure 5 presents the Pareto frontier for 250 and 500 iterations as we increase the thread count using 7 layers. In this case, we can observe a smooth difference in some parts of the Pareto frontier when 1, 2 and 16 threads are used.

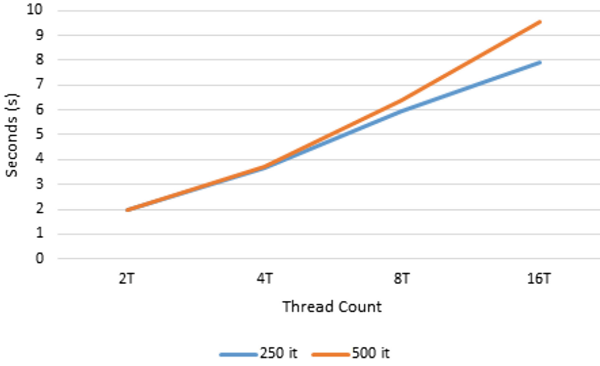

Fig. 3. Speedup achieved by E-MOPBIL for 250 and 500 iterations as we increase the thread count with 7 layers

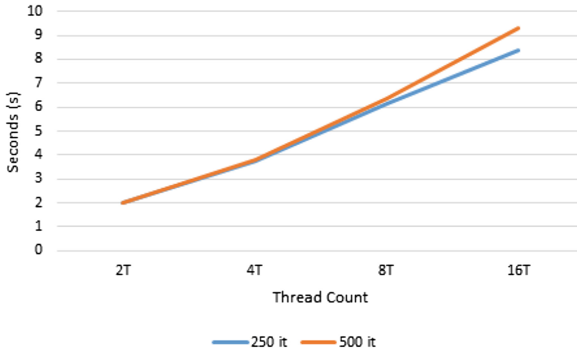

Fig. 4. Speedup achieved by E-MOPBIL for 250 and 500 iterations as we increase the thread count with 15 layers

This difference is expected because we divided the number of iterations between processor units. The coverage metric for 250 iterations and 7 layers indicates that using 1 thread it dominates 51.8 % from the Pareto frontier obtained by 16 threads, and 16.7 % on the contrary. On the other hand, the difference is not meaningful because the variation between these Pareto frontiers, computed by generational distance, is only 0.0006421748.

In order to make a fairer comparison between MOPBIL and E-MOPBIL, we divided the iterations that each slab can do based on the number of threads. Doing so, both algorithms MOPBIL and E-MOPBIL execute the same number of iterations; consequently, they perform the same number of calls to evaluation function. Figures 6 and 7 show the speedup reached by MOPBIL using 7 and 15 layers, respectively, where we can observe that dividing the number of iterations on each slab by the number of processor units almost leads to the ideal speedup.

Even though dividing the number of iteration from MOPBIL into slabs produces better speedups than E-MOPBIL, the quality of the solutions is affected

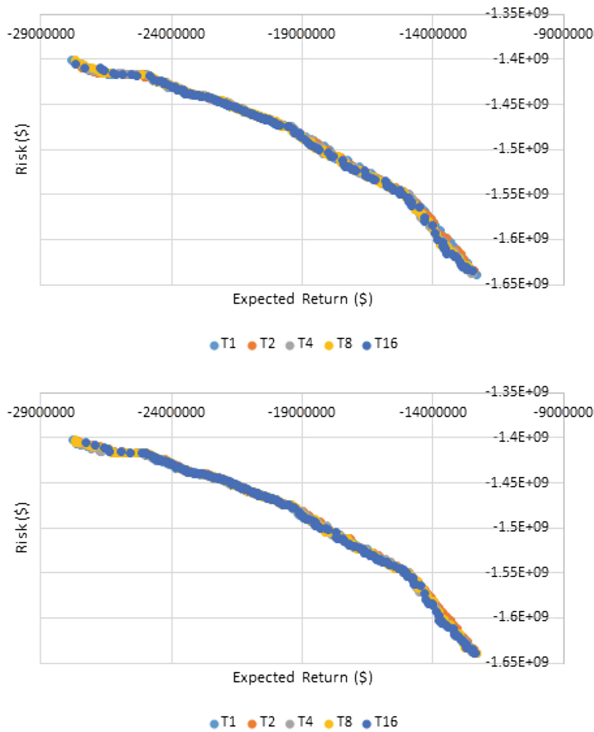

Fig. 5. Pareto frontier for 250 and 500 iterations as we the thread counting with 7 layers

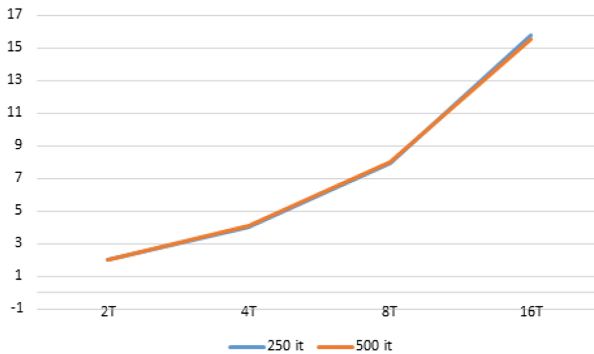

Fig. 6. Speedup achieved by MOPBIL for 250 and 500 iterations as we increase the thread count with 15 layers

as depicted in Fig. 8 in which we compare the Pareto frontier of 7 layers, 250 iterations, and 2 threads. In fact, the coverage indicates that E-MOPBIL domi-

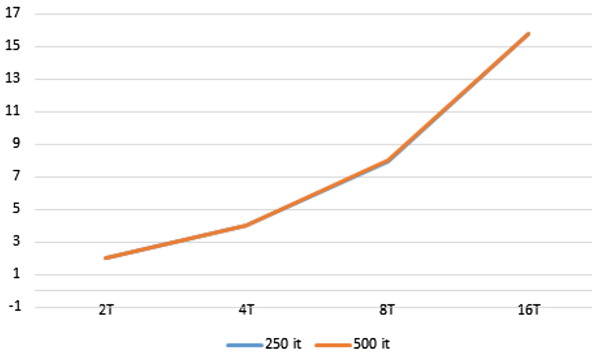

Fig. 7. Speedup achieved by MOPBIL for 250 and 500 iterations as we increase the thread count with 15 layers

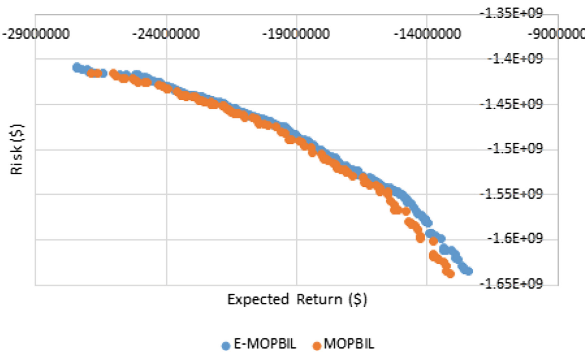

Fig. 8. Comparison against Pareto frontiers using 2 threads, 7 layers, and 250 iterations

nates 92.3 % of solutions from MOPBIL, and that MOPBIL does not dominate any solution from E-MOPBIL.

Although the number of iterations is smaller as we increase the number of threads, a difference still occurring using 16 thread and 500 iterations in a 15-layered problem as illustrated in Fig. 9. Regarding coverage, E-MOPBIL dominates 88.8 % of solutions, while MOBPBIL dominates only 0.6 % of solutions. In other words, the Pareto frontier of E-MOPBIL overcomes that one from the original algorithm in this configuration as well.

Table 5 shows metrics for the comparison between both algorithms using 16 threads, 500 iterations and 15 layers. The results validate that even though this slightly modified version of MOPBIL is faster in terms of execution time, E-MOPBIL produces more solutions, better hypervolume, and consequently a better Pareto frontier.

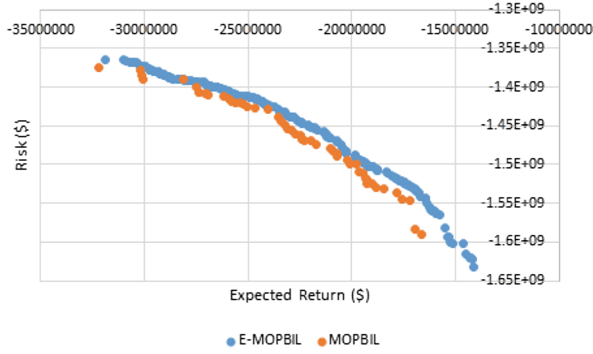

Fig. 9. Comparison against Pareto frontiers using 16 threads, 15 layers, and 500 iterations

Table 5. Metrics for 500 iterations, 15 layers, and 16 threads

MOPBIL				
	#NS	HV	Time (s)	Final #NS
Avg	29.43	1.61E+15	**52.32**	51
Std	3.90	2.85E+14	0.95	-
E-MOPBIL				
	#NS	HV	Time (s)	Final #NS
Avg	**93.93**	**2.62E+15**	78.97	220
Std	8.46	2.78E+14	3.83	-
t	**-37.90**	**13.86**	**-36.99**	-

5 Conclusions

This paper presented the Enhanced MOPBIL method, E-MOPBIL, and showed how it achieves better results than the original method concerning both performance and quality of solution. E-MOPBIL is up to 21.5 % faster than the original method when run sequentially using a single thread.

Furthermore, MOPBIL did not exhibit any speedup when run in a multicore setting due to its structure and how it executed iterations over distinct segments (slabs) of the parameter space. In E-MOPBIL the user has more flexibility in that they can control how iterations are executed and trade-off small quality losses for up to linear speed-up. This feature is attractive in many settings where timely results are critical.

Acknowledgment. The authors would like to thank CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) and IFMA (Instituto Federal de Educação, Ciência e Tecnologia do Maranhão) for funding this research.

References

1. Kennedy, J., Eberhart, R.: Particle swarm optimization. IEEE Int. Conf. Neural Netw. **4**, 1942–1948 (1995)
2. Storn, R., Price, K.: Differential evolution: A simple and efficient adaptive scheme for global optimization over continuous spaces (1995)
3. Michalewicz, Z.: Genetic Algorithms + Data Structure = Evolution Programs. 3rd edn (1999)
4. Yao, X., Liu, Y., Lin, G.: Evolutionary programming made faster. IEEE Trans. Evol. Comput. **3**(2), 82–102 (1999)
5. Baluja, S.: Population based incremental learning (1994)
6. Servais, M., de Jager, G., Greene, J.R.: Function optimisation using multiple-base population based incremental learning. In: The Eighth Annual South African Workshop on Pattern Recognition, Rhodes University (1997)
7. Yuan, B., Gallagher, M.: Playing in continuous spaces: Some analysis and extension of population-based incremental learning. IEEE Congr. Evol. Comput. IEEE **17**, 443–450 (2003)
8. Bureerat, S.: Improved population-based incremental learning in continuous spaces. In: Gaspar-Cunha, A., Takahashi, R., Schaefer, G., Costa, L. (eds.) Soft Computing in Industrial Applications. AISC, vol. 96, pp. 77–86. Springer, Heidelberg (2011)
9. Cortes, O.A.C., Rau-Chaplin, A., Wilson, D., Cook, I., Gaiser-Porter, J.: Efficient optimization of reinsurance contracts using discretized PBIL. In: The Third International Conference on Data Analytics, pp. 18–24 (2013)
10. Cortes, O.A.C., Rau-Chaplin, A., Wilson, D., Gaiser-Porter, J.: On PBIL, DE and PSO for optimizationof reinsurance contracts. In: Esparcia-Alcázar, A.I., Mora, A.M. (eds.) EvoApplications 2014. LNCS, vol. 8602, pp. 227–238. Springer, Heidelberg (2014)
11. Brown, L., Beria, A.A., Cortes, O., Rau-Chaplin, A., Wilson, D., Burke, N., Gaiser-Porter, J.: Parallel MO-PBIL: Computing pareto optimal frontiers efficiently with applications in reinsurance analytics. In: 2014 International Conference on High Performance Computing Simulation (HPCS), pp. 766–775, July 2014
12. Wang, H., Cortes, O., Rau-Chaplin, A.: Dynamic optimization of multi-layered reinsurance treaties. In: The 30th ACM/SIGApp. Symposium On Applied Computing (2015)
13. Cortes, O., Rau-Chaplin, A., do Prado, P.F.: On VEPSO and VEDE for solving a treaty optimization problem. In: IEEE International Conference on Systems, Man and Cybernetics (SMC), pp. 2427–2432, October 2014
14. Cai, J., Tan, K.S., Weng, C., Zhang, Y.: Optimal reinsurance under VaR and CTE risk measures. Insurance: Mathematics and Economics, pp. 185–196 (2008)
15. Zhang, Q.H.: Moea/d: A multiobjective evolutionary algorithm based on decomposition. IEEE Trans. Evol. Comput. **11**(6), 712–731 (2007)
16. Alba, E.: Parallel evolutionary algorithms can achieve super-linear performance. Inf. Process. Lett. **82**(1), 7–13 (2002)