Parallel MO-PBIL: Computing Pareto Optimal Frontiers Efficiently with Applications in Reinsurance Analytics

Leah Brown*, Anirudha Ashok Beria†, Omar A. C. Cortes‡ Andrew Rau-Chaplin*, Duane Wilson*, Neil Burke*, and Jürgen Gaiser-Porter§
*Risk Analytics Lab
Dalhousie University
Halifax, Nova Scotia, CA
lbrown@cs.dal.ca, arc@cs.dal.ca, dwilson@gmail.com, Neil.Burke@cs.dal.ca
†International Institute of Information Technology
Gachibowli, Hyderabad, India
anirudha.beria@students.iiit.ac.in
‡Informatics Academic Department
Instituto Federal de Educação, Ciência e Tecnologia do Maranhão
São Luis, MA, Brazil
omar@ifma.edu.br
§Global Analytics
Willis Group
London, UK
gaiserporterj@willis.co

***Abstract*—In this paper we propose MO-PBIL, a parallel multidimensional variant of the Population Based Incremental Learning (PBIL) technique that executes efficiently on both multi-core and many-core architectures. We show how MO-PBIL can be used to address an important problem in Reinsurance Risk Analytics namely the Reinsurance Contract Optimization problem. A mix of vectorization and multithreaded parallelism is used to accelerate the three main computational steps: objective function evaluation, multidimensional dominance calculations, and multidimensional clustering. Using MO-PBIL, reinsurance contract optimization problems with a 5% discretization and 7 or less contractual layers (subcontracts) can be solved in under a 1 minute on a single workstation or server. Problems with up to 15 layers, which previously took a month or more of computation to solve, can now be solved in less than 10 minutes.**

***Keywords*-Evolutionary Heuristic Search; Population Based Incremental Learning; Reinsurance Analytics**

I. Introduction

Parallel algorithms for evolutionary heuristic search methods are becoming increasingly important in order to leverage hardware advances and perform complex optimizations quickly and scalably in competitive domains such as computational finance [4]. In this paper, we study an important variant of the Reinsurance Contract Optimization problem [5], [7], viewed from the perspective of a primary insurance company. Namely, given a reinsurance contract consisting of a fixed number of layers (subcontracts) and a simulated set of expected loss distributions (one per layer), plus a model of reinsurance costs, identify optimal combinations of placements (i.e. percentage shares) such that for a given expected return the associated risk value is minimized. The solution to this high-dimensional multi-objective optimization problem is a Pareto frontier that quantifies the available trade-offs between expected risk and returns, as illustrated in Figure 1.

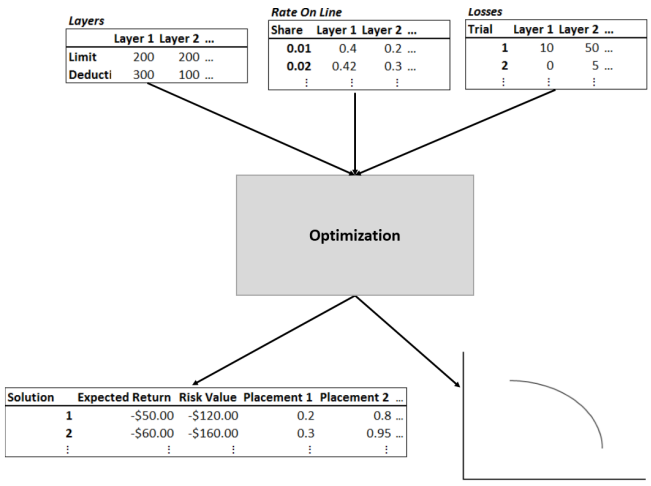

Figure 1. The studied reinsurance contract optimization problem: Inputs and Outputs

This problem, which was first addressed in [5], is central to the placement of reinsurance risk in the marketplace. Mistry et al. took a straightforward enumeration approach and addressed the performance challenge by harnessing a distributed memory cluster to perform the search in parallel. Unfortunately, while this approach allows one to solve

low dimensional problems (e.g. 5 layers or less), assuming a fairly coarse discretization (e.g. 10% increments), in a reasonable amount of time, it is of limited use for practical industrial problems where the number of layers may be between 7 and 15 and finer discretization (e.g. 1%) is required. For instance, a contract with 7 layers and 5% discretization takes more than a week to be solved by the enumeration approach on a sequential machine as we can see from the timing chart given in Figure 2, which was obtained by estimation after some executions of the enumeration method implemented in R.

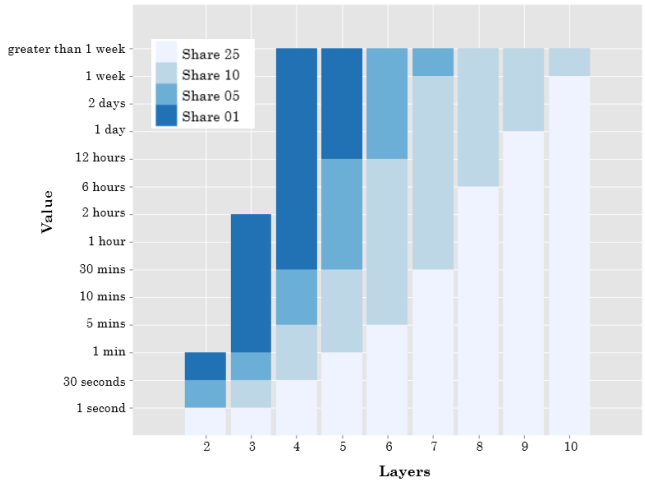

Figure 2. Time required by the enumeration

This problem was further studied by Cortes et al. in [7]. The main advance here was to discard the enumurative search in favour of an evolutionary heuristic search method called Population Based Incremental Learning [13], or PBIL. This work advanced the state-of-the-art by demonstrating that an evolutionary search approach, called Discretized PBIL (DiPBIL), could generate results of acceptable quality and solve interesting industrially relevant problems on a single multi-core workstation (rather than a 36 node cluster) in a timely manner. For example, contract optimization problems with a 5% discretization and 7 or less contractual layers could be solved in less than an hour. However, the DiPBIL approach suffered from a number of weaknesses. In converting the multi-objective contract optimization problem into a mono-objective, it reduced the quality of the results. Secondly, the Pareto frontier had to be built point-by-point, with the method basically searching for the risk/return tradeoff for each level of expected return. This led to both unnecessary computational overhead and a search procedure that spreads its work evenly over the frontier rather than dedicating more computation to the more important regions of the curve.

In this paper, in order to overcome the problems inherent in the DiPBIL approach and to increase the dimensionality of solvable contract optimization problems, we propose an approach based on Multi-Objective PBIL (MO-PBIL) and describe a parallel version of this optimization technique that runs efficiently on both multi-core and manycore architectures. The algorithm mixes vectorization and multi-threaded parallelism to accelerate the three main computational steps, namely objective function evaluation, multi-dimensional dominance calculations, and multidimensional clustering. Using this new method reinsurance contract optimization problems with a 5% discretization and 7 or less contractual layers can be solved in less than 1 minute on a single workstation, while problems with up to 15 layers (that previously would have taken a month or more of computation to solved) can be run in less than 10 minutes.

The remainder of this paper is structured as follows: In Section II, we introduce multi-objective search and the MO-PBIL approach. We then introduce a parallel version of MO-PBIL that can efficiently exploit modern multi-core and many-core architectures (Section III). Afterwards, Section IV presents how the parallelization was implemented. An extensive experimental evaluation of the proposed method on a variety of hardware is presented in Section V. Finally, Section VI concludes and describes the planned future work.

II. MO-PBIL: A Multi-objective Optimization Technique

In this section we review the required multi-objective optimization fundamentals and describe the sequential MO-PBIL algorithm that will be parallelized.

Sometimes generating a Pareto set for a particular problem using classical methods might be infeasible due to the time required to compute it, especially when real world problems are being solved. In this context, evolutionary algorithms appear as an alternative to discover approximate solutions. Many real world problems are hard to optimize because they are involve tradeoffs between two, three, or even many objectives. In these cases, effective optimization must take into account multi-objective functions.

A. Multiobjective Fundamentals

As the name suggests, a multi-objective optimization problem considers multiple conflicting objective function [11]. Typically, the objective functions are in conflict, otherwise the solution is only a point in the search space (i.e. there is no Pareto set). In other words, for a multi-objective optimization problem there exist no single global solution. Given an arbitrary optimization problem with k objective functions to be maximized and assuming that a solution to this problem can be represented by a vector in a search space X with m elements, a function $f : X \rightarrow Y$ is used to evaluate the quality of a solution by mapping it into an objective vector in an objective space $Y \in \Re$ as depicted in Figure 3.

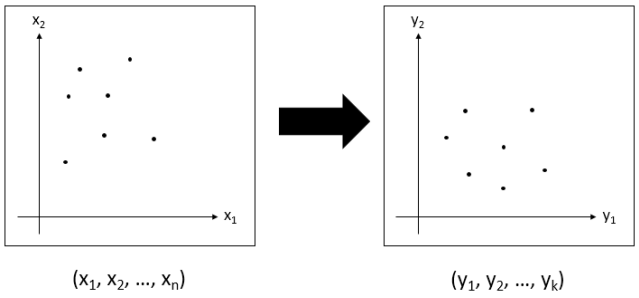

Figure 3. Moving from the search space to objective space

In this context, a multi-objective problem is defined as presented in Equation 1, where f is a vector of objective functions, m is the dimension of the problem and n the number of objective functions.

$$Min\ y = f(x) = (f_1(x_1, ..., x_m), ..., f_n(x_1, ..., x_m)) \quad (1)$$

As mentioned previously, there is no single global solution, thus it is necessary to determine which solutions are better for the problem being solved. We can use the concept of optimality, where given two vectors $x, x* \in \Re$ and $x \neq x*$, x dominates $x*$ (denoted by $x \prec x*$) if $f_i(x) \leq f_i(x*), \forall\ i$. Hence, a solution x is said to be Pareto optimal if there is no solution that dominates x in all dimensions. In this case, x is called non-dominated solution. So, considering the set of non-dominated solutions $\wp$, we can build the Pareto frontier (pf) according to $pf = \{f_i(x) \in \Re | x \in \wp\}$.

B. The MO-PBIL

PBIL was first proposed by Baluja [12] in 1994. The algorithm's population are encoded using binary vectors and an associated probability vector, which is then updated based on the best members of a population. Unlike other evolutionary algorithms which seek to transform the current population into new populations, a new population is generated at random using the updated probability vector for each generation. Baluja describe his algorithm as a "combination of evolutionary optimization and hill-climbing" [12].

Since Baluja's work, extensions to the algorithm have been proposed for continuous and base-n represented search spaces [13], [15], [14]. In [7] the intervals are replaced for equidistant increments in the lower and upper bounds of the search space, thereby defining a discrete version of PBIL, called DiPBIL. However, DiPBIL still requires that all of the objective functions get transformed into a single objective. In order to address this issue and compute a better Pareto frontier a true multi-objective optimization approach called MOPBIL is presented in Algorithm 1.

Basically, the MO-PBIL divides the search space into n overlapping slabs as illustrated in Figure4. Within each slab,

Input: N_G = number of generations;
Estimate the min and the max of the mean;
Divide the interval [min, max] into n slabs;
while *(N_G not reached)* **do**
  for *s from 1 to slabs* ***in parallel*** **do**
    1. Create the population using probability matrix;
    2. Evaluate objective function on each member of the population;
    3. Determine the non-dominated set;
    4. Cluster the non-dominated set into k clusters and then select k representative individuals;
    5. Insert the k individuals into the population;
    6. Update and mutate the probability matrix;
  end
end
Combine the results of each slab;
Determine the Pareto frontier;

Algorithm 1: Parallel MO-PBIL (Sketch)

the population is created using the probability matrix exactly as proposed in [7]. This set of non-dominated solutions is found using Kung's dominance algorithm [11] and the resulting set is clustered. Finally, k representative solutions are chosen to update the probability matrix. The clustering process is introduced to help to keep the diversity along the generations, *i.e.*, the algorithm tries to chose representatives from k^{th} most different solutions keeping them in the population and also updating the probability matrix. The individual which presents the best risk value is chosen to represent a cluster. In other words, we are trying to drive the search towards the optimal risk values and keep the diversity in the population at the same time. When the number of generations is reached the results within each slab are combined in order to get the final Pareto frontier. Figure 5 graphically illustrates the high level structure of the process.

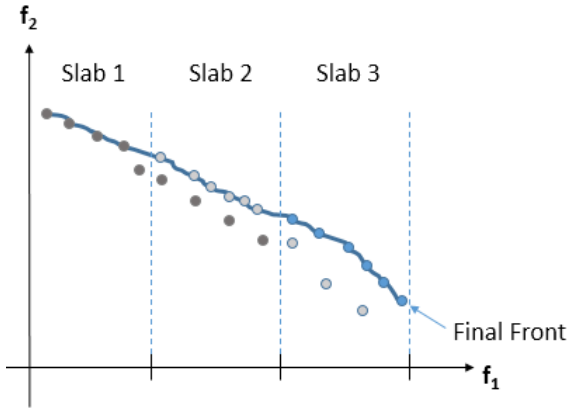

Figure 4. Dividing the work into slabs

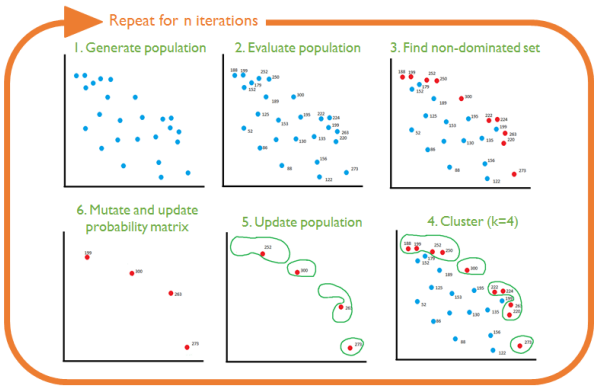

Figure 5. How the MOPBIL creates points and update the probability matrix.

III. Parallel MO-PBIL

We parallelized MO-PBIL using OpenMP and the `#pragma omp parallel for` directive since much of the work in the MO-PBIL algorithm used looping structures to iterate through the array structures. While Algorithm 1 sketches the basic MO-PBIL method and its primary source of independent parallel work, namely slabs, there are several other sources of parallelism that can be exploited. In our implementation, which is described in detail in Algorithm 2, we generally tried to parallelize over the longest loops in order to be able to effectively utilize large processor core counts.

Our implementation also split the evaluation function into two parts to address the problem that arose when the calculated mean (expected return) fell outside the current slab boundaries. By breaking the evaluation function into two parts, one to evaluate the mean of each member of the population and the other to evaluate the risk, the algorithm could discard any "Out of Range" points generated between these steps.

The evaluation of mean values was performed by iterating over the population size, since each member of the population could be independently evaluated and the population provided a high loop count for parallelization. For the risk evaluation, the "In Range" population was divided into equal parts so that each thread would have equal work, avoiding the work imbalance otherwise caused by "Out of Range" points. This lead to a solution in which the parallel evaluation of risk was performed over the "In Range" points rather than the slabs. Finally, each of the evaluation functions calculated the linear extrapolation (i.e. duplicating work), since it was found that the cost of computing the linear extrapolation multiple times was less than the overhead of storing them for later use. In addition to parallelizing the main algorithm, each of the evaluation functions was also parallelized as shown below.

Parallel MO-PBIL;
Inputs: $sliceSize$, $q1$, $q2$, $nSlabs$, $nBest$, $LR2$, $mRate$, $mShift$, $numObj$, $ppln$, $iter$;
Outputs: The Pareto frontier with the optimal placements;

Estimate min and max of the mean;
Divide the interval [min, max] into $nSlabs$ slabs.;
for *(each slab in **parallel**)* **do**
  Initialize the probability matrices;
end
for *(i from 1 to $iter$)* **do**
  for *s from 1 to slabs* **do**
    for *(each layer **in parallel**)* **do**
      Generate cumulative probability matrix;
    end
    (1) Generate $ppln$ from probability matrix;
    for *i from 1 to $pplnSize$ **in parallel*** **do**
      (2) Evaluate mean of ppln member p;
    end
    Discard ps whose means are outside slab;
    Divide valid population between threads;
    for *(each p in division **in parallel**)* **do**
      (2) Evaluate risk of p;
    end
  end
  for *(s from 1 to slabs **in parallel**)* **do**
    (3) Determine non-dominated set (Kung's alg.);
    (4) Cluster non-dominated set into k clusters and best $nBest$ members (k-Means algorithm);
    (5) Update ppln using best members from (4);
    (6) Update and mutate the probability matrix p;
  end
end
Combine results from each slab;
Determine Pareto frontier;

Algorithm 2: Parallel MO-PBIL (Details)

MO-PBIL Algorithm: Means Evaluation;
Inputs: ppln member p, exp. ret. val. of p's slab erv;
Outputs: p's mean m, $isInSlab$;

for *(i each layer in p **in parallel**)* **do**
  Find linear interpolation;
end
Calculate the mean of p, m;
Calculate $isInSlab$ using erv

Algorithm 3: Parallelization of Means Evaluation

MO-PBIL Algorithm: Risk Evaluation;
Inputs: population member p, p's mean m;
Outputs: p's risk values at $q1$ and $q2$ from main alg.;

for *(i each layer in p* ***in parallel)*** **do**
  Find linear interpolation;
end
Find risk value at$q1$ (randomized k-means selection algorithm Find risk value at $q2$ (randomized k-means selection algorithm return m, $isInSlab$

Algorithm 4: Parallelization of Risk Evaluation

IV. Speeding up Parallel MO-PBIL

A single C++ code base was developed and run on both a standard Intel Xeon multi-core processor and a machine consisting of a host Xeon multi-core processor with a coprocessor based on a Intel Phi many-core chip [10].

Although the code was recompiled to run on each architecture natively, the basic code optimizations for both machines was largely the same. In fact, the final running code differed primarily in the number of slabs used to compute the Pareto frontier. For the Phi, this meant compiling with the *-mmic* flag and off-loading the executable from the Xeon and running it independently on Phi using the *micnativeloadex* utility. Because both the Xeon and the Phi have relatively wide vector registers (256 and 512 bits wide, respectively), both codes were vectorized to help increase performance. Most of the vectorization was automatically performed by Intel's compiler using a $-O < optimizationlevel >$ compiler flag. For MO-PBIL, level 3, the highest level of optimization was used. It automatically performs vectorization, inlining, constant propagation, dead-code elimination, loop unrolling, loop fusion, block-unroll-and-jam, and if-statement collapse. A *-vec-report* flag was also used to monitor where automatic vectorization took place and where it did not. In addition, several instances of loop exit statements (i.e. *break*, *continue*, etc.) which prevent automatic vectorization were removed and replaced with masked assignments.

The parallelism was implemented using OpenMP by means of the instruction *#pragma omp parallel for schedule(guided) num_threads(NUM_THREADS)* parallelism on both the Xeon and Phi. The main advantage of doing so is there is no need to perform any changes in the code when we move it from Xeon to Phi. The "guided" schedule option which instructs the scheduler to distribute increasingly smaller sizes of work to threads, was selected because it gave the best performance in practice. In addition, some parallelized loops with work independent of each other were fused together. This removed the implicit wait barrier at the end of each *#pragma omp parallel for* that allows all threads to synchronize, which can be an expensive operation. By fusing two loops together, it was possible to remove a synchronization.

Unfortunately, many of the loops had relatively small iteration counts because they were over the number of layers (typically 7-15) or the number of slabs. This raised concerns for parallelism on both architectures; with a smaller number of layers, MO-PBIL could not take advantage of the Xeon's 16 cores, and on the Phi, the 240 threads available could not be fully utilized even for the largest test problems. The largest loops were over the iteration size, but because iterations had to be executed sequentially, parallelization over this was not possible. Despite these challenges significant parallel speedup was achieved, especially on the Xeon platform, as described in the following experimental evaluations.

V. Experimental Results

A. Setup

The experiments were conducted on two different architectures. The first one is an Intel Xeon comprising of two Xeon processors E5-2650 running at 2.0 Ghz with 8 cores and 256 GB of memory. The second one is an Intel Xeon with Phi 5110p Card with 60 cores and 8Gb of memory.

The Intel Xeon Phi Coprocessor [10] is Intel's latest solution to find a middle ground between obtaining lowpowered, GPU-like performance improvements while keeping programming overhead at a minimum. Unlike GPU programming, code compiled for the Intel Xeon Phi Coprocessor can be run on both the Xeon and the Phis architecture. It uses the Multi Integrated Core (MIC) architecture and features 60 processing cores, caches, memory controllers, PCIe client logic, and a very high bandwidth bidirectional interconnection ring. Each core comes complete with a private L2 cache that is kept fully coherent by a global distributed tag directory, uses 512-bit wide vector registers, and is capable of 4-way hyper-threading for up to 240 threads of execution. The memory controllers and the PCIe client logic provide a direct interface to the GDDR5 memory on the coprocessor and the PCIe bus, respectively. All these components are connected together by the interconnection ring [10] as we can see in Figure 6. The coprocessor cannot function as a standalone processor and is therefore dependent on being connected to a host Intel Xeon processor through a PCI Express (PCIe) bus.

All experiments involved running MO-PBIL with 500, 1000, or 2000 iterations, 128 individuals, and 2 objective functions (Expected Loss and Risk) on contract optimization problems involve between 7 and 15 layers (ie. subcontracts). Each experiment is composed of 10 trials, i.e., 10 runs. The elapsed-time presented corresponds to the mean of these trials, although very little variation was observed in practice.

The instances of the reinsurance contract optimization problem being solved attempt to find for a given contract structure the best combination of placements (i.e. percentage shares) in order to transfer the maximum of risk and at the same time receive the maximum of the expected return.

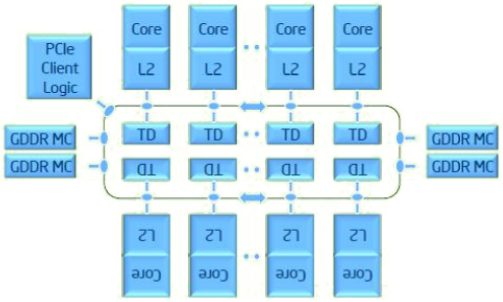

Figure 6. Intel Phi architecture [10]

Equation 2 represents the underlying optimization problem, where *VaR* is a risk metric, R is a $m \times 1$ vector of recoveries and π is a combination of shares. The first one is the quantile representing the 1 in 200 year risk. Roughly speaking these represent the actuarial and investor perspectives, where the intention is to hedge more risk and receive more money back in case of massive claims. For further details about the problem refer to [7] and [18].

$$\begin{aligned} &maximize && f_1(x) = VaR_\alpha(\mathbf{R}(\pi)) \\ &maximize && f_2(x) = E(\mathbf{R}(\pi)) \end{aligned} \tag{2}$$

B. Quality of Solutions

Ideally, the results obtained by the heuristic search technique described in this paper would be compared directly against an exact method to determine the quality of the results obtained. While this was possible for extremely small problems, it was not possible for the high-dimensional contract optimization problems that were our research target. Such problems are not solvable by exact methods in a feasible amount of time. This being the case, three metrics have been used to quantify the quality of solutions obtained. First, the number of non-dominated points found in the Pareto frontier was determined. Second, the hypervolume, which is the volume of the dominated portion of the objective space as presented in Equation 3, was measured, where for each solution $i \in Q$ a hypercube v_i is constructed with a reference point W ($(0,0)$ in this particular case because we are maximizing negative values). Having each v_i we calculated the final hypervolume by the union of all v_i.

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

Third, the dominance relationship between Pareto frontiers obtained with increasing iterations, i.e. the coverage, was calculated as depicted in Equation 4. Roughly speaking, the coverage is the ratio between the number of solutions dominates by A divided by the number of solution of set B.

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

For further details about the use of these metrics see [11]. Finally, the resulting frontiers were reviewed by experts for reasonability.

Figure 7 shows the average number of non-dominated points found on the Pareto frontier for contracts with varying number of layers on the Intel Xeon and Intel Phi, respectively. We expected that as we increase the number of layers, and therefore the size of the search space, it would become harder to find solutions on the Pareto frontier. Interestingly, this was not the case. The size of the Pareto frontier diminished between 7 and 11 layers but then began to grow again. We also observed, as we would hope, that as we increased the number of iterations the size of the Pareto frontiers found also increased. Increasing the number of iterations positively effected the number of solutions that the algorithm identified in all observed cases.

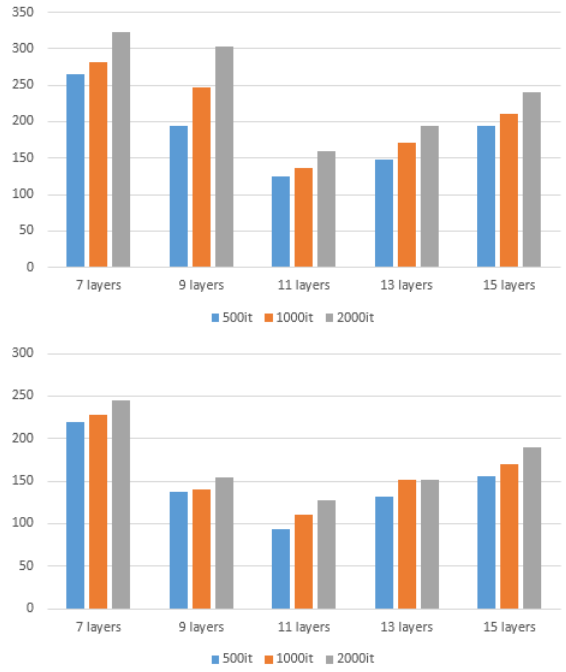

Figure 7. Average number of solution - Iteration vs Layers on Xeon and Phi, respectively.

Figure 8 shows, for both architectures, the size of the dominated hypervolume as a function of the number of layers and the iteration count. We observe that as the number of layers increases so does the size of the dominated hypervolume. Interestingly, increasing the number of iterations has only a modest effect on hypervolume size. Five hundred iterations appears to be enough to capture the basic shape of the Pareto frontier, and while more iterations refine that shape there are no large changes observed.

The coverage observed on both Intel Xeon and Phi architectures is shown in Table I. Each line represents the comparison between iterations, for example, 500-1000

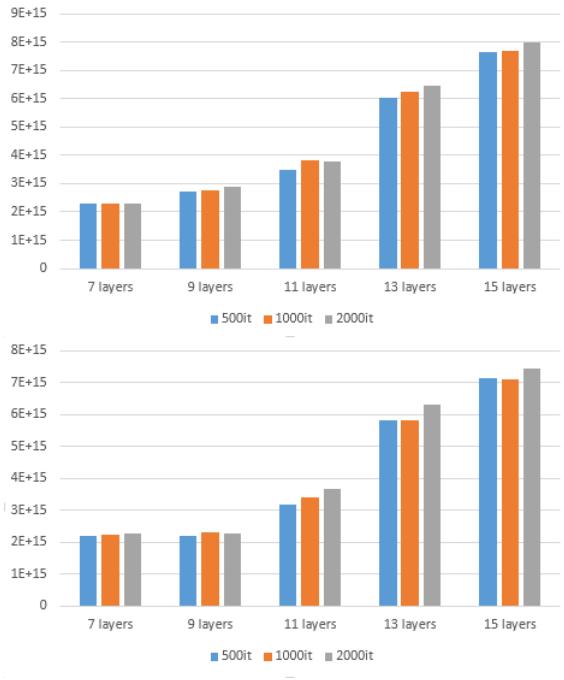

Figure 8. Average hypervolume - Iteration vs Layers on Xeon and Phi, respectively.

depicts the rate of the number of solution obtained by 1000 iterations which are dominated by the solutions reached by 500 iterations. For instance, the number 0.38 indicates that 38% of solutions obtained by 1000 iterations are dominated by the other one. A result equal to 1 would indicate that the first Pareto frontier dominates all points of the second one. On the other hand, an outcome equal to 0 indicates the inverse. Here we observe that in all cases performing more iterations improve the quality of the results,as expected. Either, when the problem becomes more complex the differences start being perceptible with 2000 iterations. However, a careful review of these improvements shows that they tend to be very small in relative numeric terms as shown in Figure 9.

C. Run-time and Speedup on the Intel Xeon

Figure 10 and 11 depict the elapsed time and the speedup observed on the Intel Xeon architecture, with 500, 1000 and 2000 iterations, and layers ranging from 7 to 15. In general, as we increase the number of threads performance improves up to 16, which is the number of physical cores we have available. Between 16 and 32 threads performance actual drops showing that hyperthreading is actually counterproductive in this particular case. The speedup observed ranges from near linear for small thread counts to just a factor of 4 with 16 threads and 11 layer. However, the speedup tends to be larger for harder problems with longer run times and smaller for the fast easy problems, so the use of parallelism is helping us when it counts most. In all cases, the run time

Table I
COVERAGE FOR SOLUTIONS USING 32 THREADS PER LAYER ON XEON AND 128 THREADS ON PHI

Xeon - 16 threads					
#Iter/#layers	7L	9L	11L	13L	15L
500-1000	0.42	0.17	0.29	0.25	0.31
500-2000	0.20	0.10	0.17	0.11	0.17
1000-500	0.53	0.72	0.67	0.63	0.57
1000-2000	0.24	0.18	0.46	0.10	0.30
2000-500	0.67	0.76	0.71	0.85	0.76
2000-1000	0.69	0.68	0.41	0.77	0.67
Phi - 64 threads					
#Iter/#layers	7L	9L	11L	13L	15L
500-1000	0.31	0.42	0.43	0.34	0.45
500-2000	0.27	0.39	0.16	0.20	0.28
1000-500	0.55	0.44	0.42	0.67	0.45
1000-2000	0.25	0.31	0.11	0.31	0.31
2000-500	0.64	0.52	0.70	0.71	0.65
2000-1000	0.62	0.48	0.74	0.54	0.63

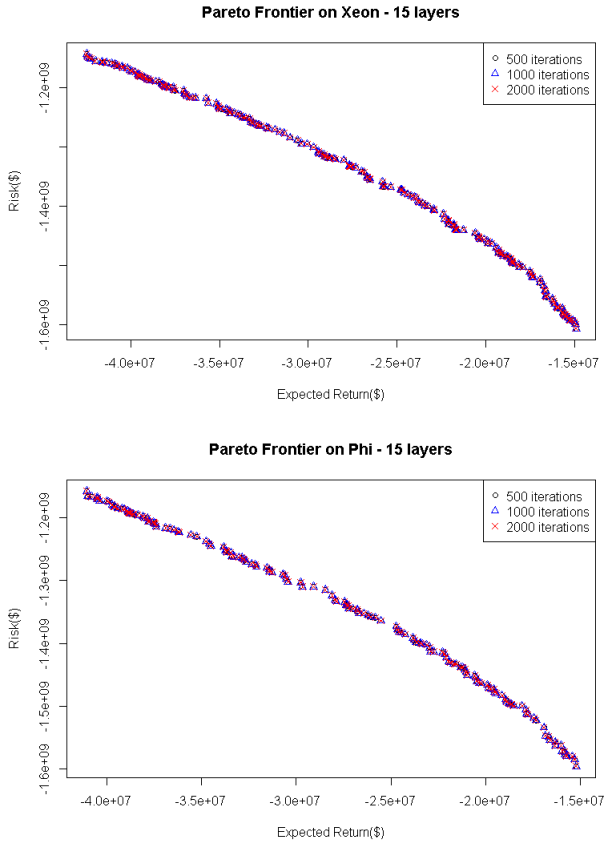

Figure 9. Pareto frontier varying the iteration count on Xeon and Phi with 15 layers

using sixteen threads (on sixteen cores) is less than 100 seconds for 500 iterations, 200 seconds for 1000 iterations and 500 seconds for 2000 iterations. This is fast enough to make practical the analysis of large reinsurance contracts which previously would have been infeasible. Reducing

runtimes from weeks or months to minutes permits the effective evaluation of many scenarios in less time than it would previously have taken to evaluate a single one.

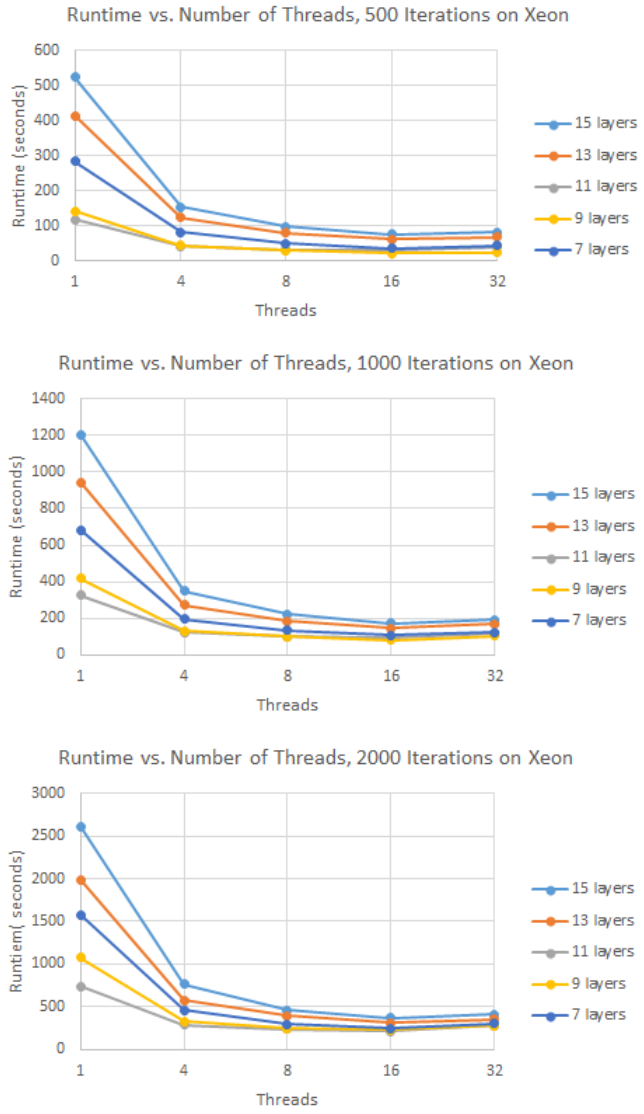

Figure 10. Runtime vs number of threads on Xeon

D. Run-time and Speedup on the Intel Phi

The Intel Phi coprocessor offers potentially 2-3 times more performance than the Intel Xeon processors (i.e. 2.0 Ghz E5-2650s) we had available. However, realizing that additional performance is not always easy and depends heavily on the problem under consideration. While the Intel Phi boasts over 1 TFlop of raw performance it is designed around a high core count (60 1.052 GHz cores each capable of executing four concurrent threads) where each core has a 512 wide vector instruction unit for floating point arithmetic.

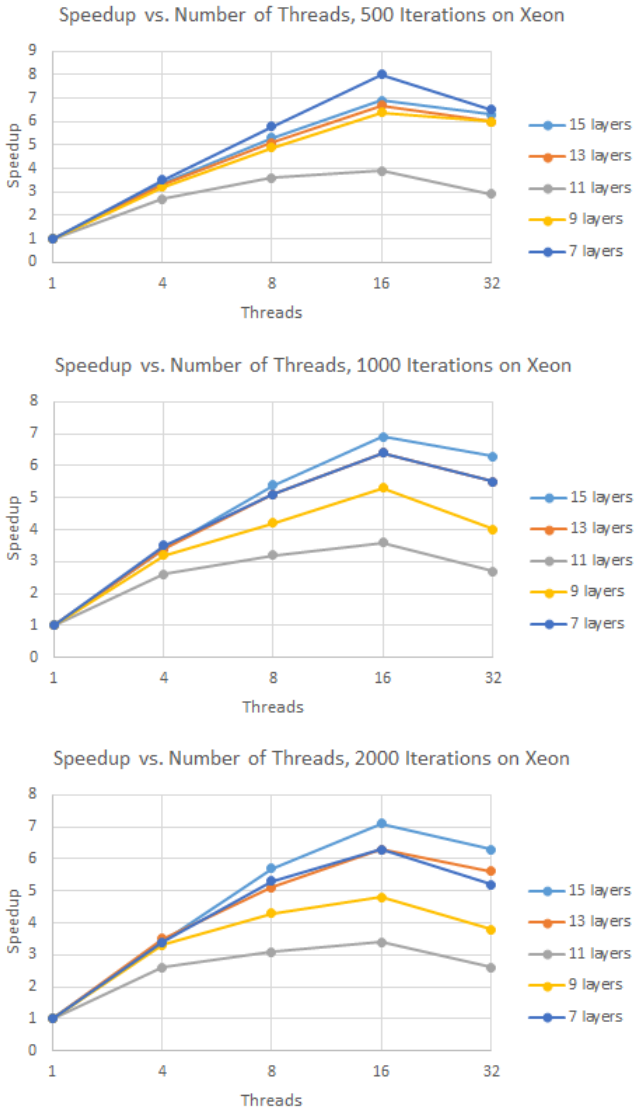

Figure 11. Speedup vs number of threads on Xeon

While in principle MO-PBIL is reasonably well suited to such an architecture, the particular problem under consideration, that is reinsurance contract optimization, poses the following significant challenges: 1) Many of the loops have low iteration counts (for example loops of the number of layers 7-15), and 2) many of the key algorithmic steps such as clustering and the calculation of VaR values are memory bandwidth intensive, rather than floating point intensive.

Figures 12 and 13 show the results for the elapsed time and speedup on Intel Phi. While the basic shape of these curves is fine the absolute values are disappointing. In general, even after extensive profiling and hand optimization the resulting times are significantly slower than observed on the Xeon architecture. While MO-PBIL runs efficiently on the Intel Phi, the reinsurance contract optimization problem

does not present enough independently parallelizable work to really take advantage of the coprocessor. Increasing the number of slabs may address this problem, but has the potential to introduce anomalies in the shape of the Pareto frontier and more overhead. This is an area that we are actively exploring.

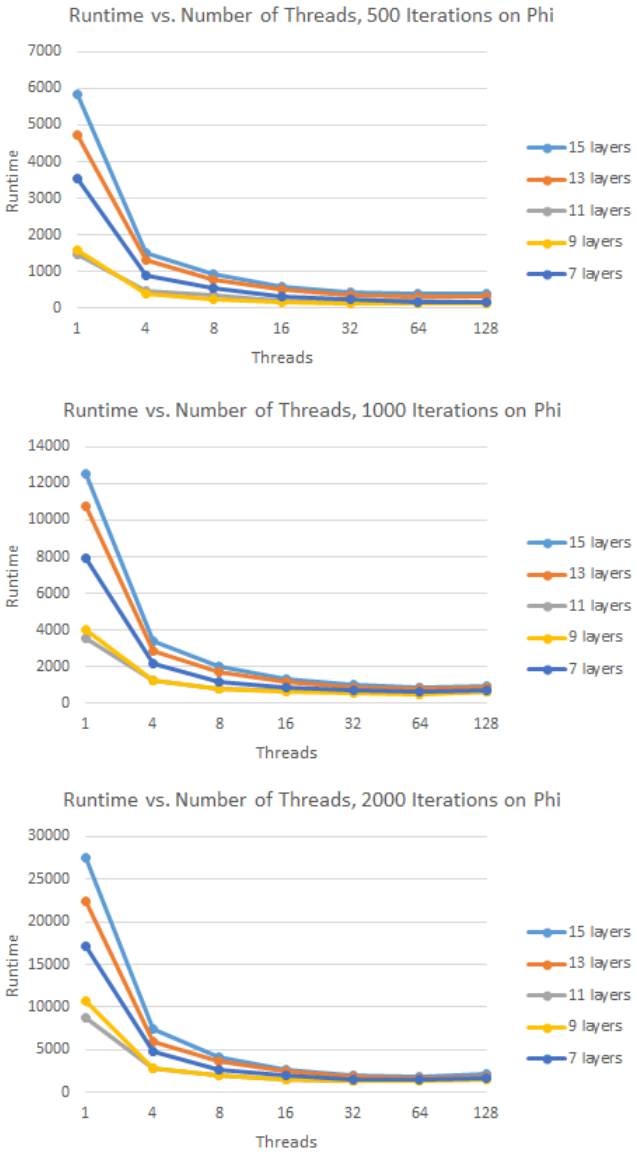

Figure 12. Runtime vs number of threads on Phi

VI. Conclusion

In this paper we have presented a parallel approach for solving a multiobjective contract optimization problem. The experimental results show that the approach when run on a standard Intel Xeon multicore significantly outperforms previous approaches (even those that exploited large distributed

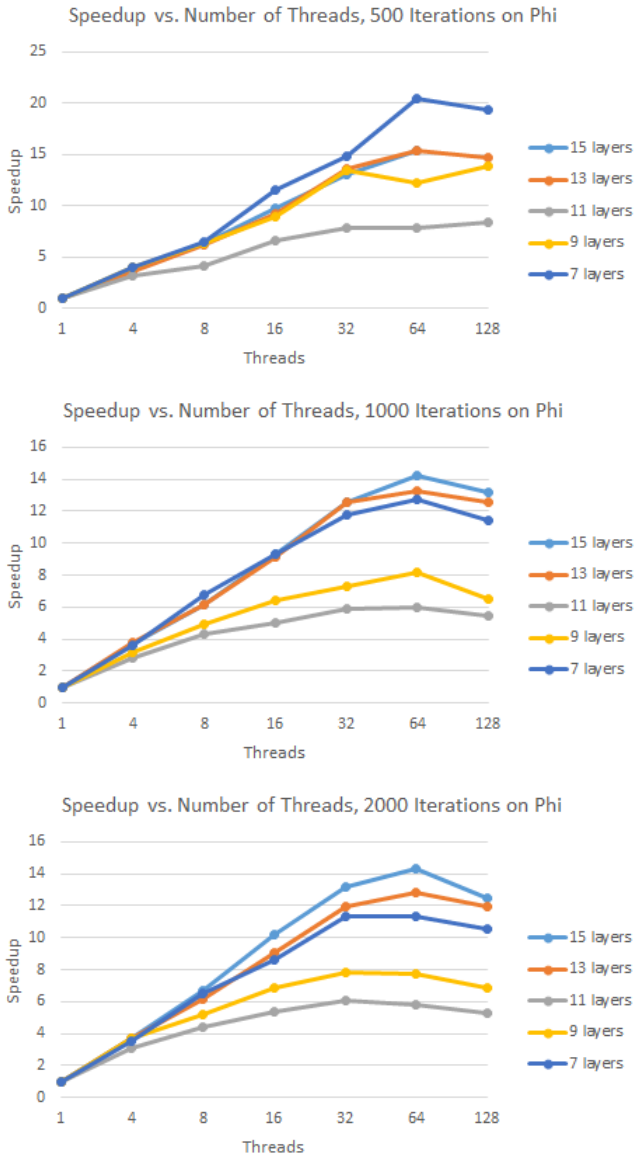

Figure 13. Speedup vs number of threads on Phi

memory clusters) in terms of time, speedup, and efficiency. Practical reinsurance contract optimization problems involving up to 15 layers (subcontracts) can now be solved in under 10 minutes when previously they would have required months of computation, if they were solvable at all.

Acknowledgment

The authors would like to thank the Willis Group for providing the actual data and, CNPq and Flagstone for funding this research.

References

[1] Misra, G. and Kurkure, N. and Das, A. and Valmiki, M. and Das, S. and Gupta, A., "Evaluation of Rodinia Cdes on Intel

Xeon Phi", International Conference on Intelligent Systems Modelling & Simulation (ISMS), pp.415-419, 2013.

[2] TOP 500 Supercomputer Sites. http://www.top500.org.

[3] Potluri, S. and Venkatesh, A. and Bureddy, D. and Kandalla, K. and Panda, D. K., "Efficient Intra-node Communication on Intel-MIC Clusters", 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp.128-135, 2013.

[4] Zircon Computing LLC, "Parallel Computing for Computational Finance Applications: A Case Study Parallelizing NAG with Zircon Software", Goethe University, Frankfurt, available at http://extwww.nag.co.uk/industryarticles/goetheuni_zircon_nag.pdf, 2010.

[5] Mistry, S. (n.d.), et al. Parallel Computation of Reinsurance Models. Unpublished Manuscript.

[6] Bureerat, S., Improved Population-Based Incremental Learning in Continuous Spaces, Soft Computing in Industrial Applications, p. 77-86, Springer, 2011.

[7] Cortes, O. A. C. and Rau-Chapliny, A. and Wilson, D. and Gaiser-Porterz, J. "Efficient Optimization of Reinsurance Contracts using Discretized PBIL", In Proceedings of Data Analytics, London, 2013.

[8] Culler, D. E. and Singh, J. P., "Parallel Computer Architectures: A Hardware/Software Approach", Morgan Kauffman Publisher, 1999.

[9] Klemm, M. and Duran, A. and Tian, X. and Saito, H. and Caballero, D. and Martorell, X., "Extending OpenMP with Vector Constructs for Modern multi-core SIMD Architectues", OpenMP in a Heterogeneous World, LNCS 7312, pp. 59-72, 2012.

[10] Intel Xeon Phi Coprocessor - the Architecture, http://software.intel.com/en-us/articles/, Visit in 02-jul-2013.

[11] Deb. K., "Multi-objective Optimization using Evolutionary Algorithms", John Wiley and Sons LTDA, 2001.

[12] Baluja, S. Population based incremental learning. *Technical Report*, Carnegie Mellon University.

[13] Bureerat, S., Improved Population-Based Incremental Learning in Continuous Spaces, Soft Computing in Industrial Applications, p. 77-86, Springer, 2011.

[14] Yuan, B. and Gallagher, M. Playing in continuous spaces: Some analysis and extension of population-based incremental learning, *CEC2003, CA, USA*, 443-450, 2003.

[15] Servais, M.P., Jager, G. and Greene, J.R. Function optimisation using multi-base population based incremental learning. *PRASA 97, Rhodes University*, 1997.

[16] Mishra, K. K. and Harit, S., "A Fast Algorithm for Finding the Non Dominated Set in Multi objective Optimization", International Journal of Computer Applications 1(25):3539, 2010.

[17] Marler. R. T. and Arora, J. S., "Survey of multi-objective optimization methods for engineering", Structural and Multidisciplinary Optimization, 26, 369395, 2004.

[18] Cai, J. et al. (2008). "Optimal reinsurance under VaR and CTE risk measures", *Insurance: Mathematics and Economics*, 43, 185-196.