2013 IEEE Congress on Evolutionary Computation
June 20-23, Cancún, México

Parallel Differential Evolution in Unified Parallel C

Pavel Krömer, Jan Platoš, Václav Snášel
IT4Innovations & Department of Computer Science
VŠB-Technical University of Ostrava
17. listopadu 12, Ostrava-Poruba, Czech Republic
Email: {pavel.kromer,jan.platos,vaclav.snasel}@vsb.cz

***Abstract*—Distributed environments and emerging highly-parallel platforms provide a suitable hardware infrastructure for parallel Evolutionary Computation. Partitioned Global Address Space model is a well-known parallel computing model used to implement scalable algorithms for many-core systems and clusters. This study investigates the Unified Parallel C programming language as a tool for implementation of scalable evolutionary algorithms for high-dimensional problems. The design concepts and initial implementation are demonstrated on the Differential Evolution algorithm. The mapping of Differential Evolution concepts to Unified Parallel C features is presented and three variants of parallel Differential Evolution for many-core shared memory systems and clusters of computers with distributed memory are implemented and evaluated in the environment of a small real-world cluster.**

***Index Terms*—Parallel Differential Evolution, Partitioned Global Address Space, Unified Parallel C**

I. Introduction

Partitioned Global Address Space (PGAS) [1] is a simple yet powerful parallel programming model in which a cluster of computers with distributed memory forms a virtual computing environment with single logical shared memory (shared global address space) that is distributed across compute nodes [2]. The PGAS can be used to design parallel applications, often using the data parallel Single Program Multiple Data (SPMD) approach. Processes taking part in the distributed computation have access to the global address space and to the information about locality (partitioning, affinity) of shared data in order to facilitate efficient distributed memory access patterns. PGAS programs do not require explicit implementation of inter-process communication and data transfers (e.g. by message passing) on application level and the data is seamlessly shared among processes via the global address space. The structure of PGAS programs is simple and they can be compared to shared memory programs. On the other hand, wrong communication patterns in which processes unnecessarily access non-local regions of shared memory can be obtained easily due to the simplicity of the model. The PGAS is a popular model for high performance computing (HPC) [3] implemented by a number of programming languages. Most used PGAS languages include Coarray Fortran and Unified Parallel C (UPC) [2], [4], [5].

The UPC [2] is a programming language for parallel and distributed computing based on C that implements the PGAS model. It is actively developed and supported by several research institutions as well as major HPC hardware and software vendors. The close similarity to C allows simple porting of existing C/C++ code to UPC. UPC compilers and libraries are available for a variety of parallel systems and under development for emerging parallel platforms such as hybrid clusters with Graphic Processing Units (GPUs) [6] and Intel Xeon Phi [7] floating point accelerators. The UPC appears to be a viable option for the implementation of portable and scalable evolutionary algorithms for HPC environments.

The Differential Evolution (DE) is a popular evolutionary algorithm that aims to solve optimization problems by an iterative evolution of a population of real-valued vectors (candidate solutions). The population of candidate solutions evolved by the algorithm performs a massively parallel search through the problem domain towards globally optimal solutions. The candidate solutions can be seen as individual points on the fitness landscape of the solved problem that are iteratively and in multiple directions at once moved towards promising regions of the fitness landscape. The implicit parallelism and simple operations of the algorithm make it suitable for parallel implementation and parallel execution and a number of parallel DEs have emerged recently [8], [9], [10]. These implementations, however, focused mostly on parallel DE for shared memory platforms and devices such as many-core CPUs, GPUs, and floating point accelerators. This study presents a UPC implementation of three parallel DE variants for many-core systems, clusters of computers, grids and all other platforms supporting the language.

The rest of this paper is structured in the following way: section II outlines the basic principles of the DE. Section III provides an introduction to Unified Parallel C and section IV describes the implementation of three parallel DE models in UPC. Section V provides an initial evaluation of the DE implemented in UPC in a real-world HPC environment and section VI concludes the study and outlines future work.

II. Differential Evolution

The DE is a versatile and easy to use stochastic evolutionary optimization algorithm [11]. It is a population-based optimizer that evolves a population of real encoded vectors representing the solutions to given problem. The DE was introduced by Storn and Price in 1995 [12], [13] and it quickly became a popular alternative to the more traditional types of evolutionary algorithms. It evolves a population of candidate solutions by iterative modification of candidate solutions by the application of the differential mutation and crossover [11]. In each iteration, so called trial vectors are created from current population by the differential mutation and further modified by various types of crossover operator. At the end, the trial vectors compete with existing candidate solutions for survival in the population.

A. The DE Algorithm

The DE starts with an initial population of N real-valued vectors. The vectors are initialized with real values either randomly or so, that they are evenly spread over the problem space. The latter initialization leads to better results of the optimization [11].

During the optimization, the DE generates new vectors that are scaled perturbations of existing population vectors. The algorithm perturbs selected base vectors with the scaled difference of two (or more) other population vectors in order to produce the trial vectors. The trial vectors compete with members of the current population with the same index called the target vectors. If a trial vector represents a better solution than the corresponding target vector, it takes its place in the population [11].

There are two most significant parameters of the DE [11]. The scaling factor $F \in [0, \infty]$ controls the rate at which the population evolves and the crossover probability $C \in [0, 1]$ determines the ratio of bits that are transferred to the trial vector from its opponent. The size of the population and the choice of operators are another important parameters of the optimization process.

The basic operations of the classic DE can be summarized using the following formulas [11]: the random initialization of the ith vector with N parameters is defined by

$$x_i[j] = rand(b_j^L, b_j^U), \quad j \in \{0, \dots, N-1\} \tag{1}$$

where b_j^L is the lower bound of jth parameter, b_j^U is the upper bound of jth parameter and $rand(a, b)$ is a function generating a random number from the range $[a, b]$. A simple form of the differential mutation is given by

$$v_i^t = v_{r1} + F(v_{r2} - v_{r3}) \tag{2}$$

where F is the scaling factor and v_r^1, v_r^2 and v_r^3 are three random vectors from the population. The vector v_{r1} is the base vector, v_{r2} and v_{r3} are the difference vectors, and the ith vector in the population is the target vector. It is required that $i \neq r1 \neq r2 \neq r3$. Differential mutation in 2D (i.e. for $N = 2$) is illustrated in fig. 1. The uniform crossover that combines the target vector with the trial vector is given by

$$l = rand(0, N-1) \tag{3}$$

$$v_i^t[m] = \begin{cases} v_i^t[m] & \text{if } (rand(0,1) < C) \text{ or } m = l \\ x_i[m] & \end{cases} \tag{4}$$

for each $m \in \{1, \dots, N\}$. The uniform crossover replaces with probability $1 - C$ the parameters in v_i^t by the parameters from the target vector x_i. The outline of the classic DE according to [11], [14] is summarized in algorithm 1. There are also many other modifications to the classic DE. Mostly, they differ in the implementation of particular DE steps such

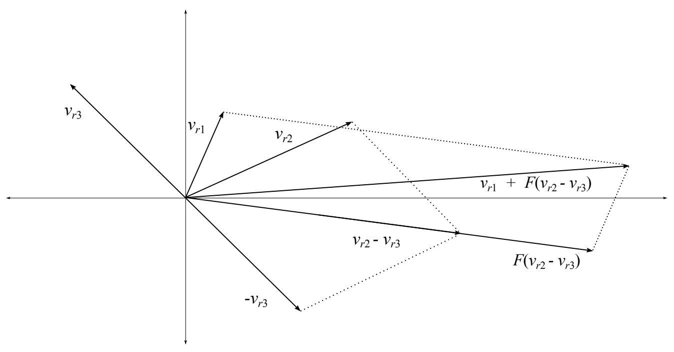

Fig. 1: Differential mutation.

Algorithm 1: A summary of classic Differential Evolution

```
1  Initialize the population $P$ consisting of $M$ vectors using eq. (1);
2  Evaluate an objective function ranking the vectors in the population;
3  while Termination criteria not satisfied do
4      for $i \in \{1, \dots, M\}$ do
5          Differential mutation: Create trial vector $v_i^t$ according to eq. (2);
6          Validate the range of coordinates of $v_i^t$. Optionally adjust coordinates
           of $v_i^t$ so, that $v_i^t$ is valid solution to given problem;
7          Perform uniform crossover. Select randomly one parameter $l$ in $v_i^t$
           and modify the trial vector using eq. (3);
8          Evaluate the trial vector.;
9          if trial vector $v_i^t$ represent a better solution than population vector $v^i$
           then
10             add $v_i^t$ to new population $P^{t+1}$
11         else
12             add $v_i$ to new population $P^{t+1}$
13         end
14     end
15 end
```

as the initialization strategy, the vector selection, the type of differential mutation, the recombination operator, and control parameter selection and usage [11], [14].

The initialization strategy affects the way vectors in the initial population are placed in the problem space. In general, a better initial coverage of the problem space represents a better starting point for the optimization process because the vectors can explore various regions of the fitness landscape from the very beginning [11].

The selection strategy defines how are the target vector, the base vector, and the difference vectors selected. Moreover, it has an effect on the time each vector survives in the population, which can be given either by the age of the vector or by the fitness of the vector. Popular base vector selection strategies are the random selection and methods based on stochastic universal sampling [11], [15]. The random selection without restrictions allows the same vector to be used as a base vector more than once in each generation. The methods based on stochastic universal sampling ensure that all vectors are used as base vectors exactly once in each generation. The selection methods based on the stochastic universal sampling generate a permutation of vector indexes that defines which base vector will be coupled with which target vector [11]. An alternative base vector selection strategy is called the biased base vector selection. The biased base vector selection uses the information about the fitness value of each vector when selecting base vectors. The biased base vector selection strategies include the best-so-far base vector selection (the best vector in the population is always selected as base vector), target-to-best base vector selection (the base vector is an arithmetic recombination of the target vector and the best-so-far vector) [11]. The biased base vector selection schemes introduce a more intensive selection pressure which can, as in other evolutionary techniques, result in faster convergence but it can also lead to a loss of diversity in the population.

Differential mutation is the key driver of the DE. It creates a trial vector as a recombination of base vector and scaled difference of selected difference vectors. The scaling factor F, which modifies vector differences, can be a firmly set constant, a random variable selected according to some probability distribution, or defined by some other function as e.g. in the self-adaptive DE variants. A variable scaling factor increases the number of vector differentials that can be generated given the population P [11]. In general, a smaller scaling factor causes a smaller steps in the fitness landscape traversal while a greater scaling factor causes larger steps. The former leads to longer time for the algorithm to converge and the latter can cause the algorithm to miss the optima [11].

The recombination (crossover) operator plays a special role in the DE. It achieves a similar goal as the mutation operator in other evolutionary algorithms, i.e. it controls the introduction of new material to the population using a mechanism similar to the $n-$point crossover in the traditional EAs. The crossover probability $C \in [0, 1]$ defines the probability that a parameter will be inherited from the trial vector. Similarly, $C - 1$ is the probability that the parameter will be taken from the target vector. Crossover probability has also a direct influence on the diversity of the population [14]. An increased diversity initiated by larger C means a more intensive exploration and faster convergence of the algorithm at the cost of the robustness of the algorithm [11]. Common DE crossover operators include the exponential crossover and the uniform crossover. Some other crossover operators are e.g. the arithmetic crossover and the either-or-crossover [11], [14].

The DE is a successful evolutionary algorithm designed for continuous parameter optimization driven by the idea of scaled vector differentials. That makes it an interesting alternative to the wide spread genetic algorithms that are designed to work primarily with discrete encoding of the candidate solutions. As well as genetic algorithms, it represents a highly parallel population based stochastic search meta-heuristic. In contrast to the GA, the differential evolution uses the real encoding of candidate solutions and different operations to evolve the population. It results in different search strategy and different directions found by DE when crawling a fitness landscape of the problem domain.

B. Parallel DE

Multiple parallel Evolutionary Computation (EC) models have been proposed in the past [16], [17]. Parallel EC models have been designed for particular evolutionary methods (e.g. Genetic Algorithms) but they can be usually applied to other EC algorithms with just minor modifications. In general, parallel EC with a single panmitic population and parallel multi-populational EC can be distinguished. The panmitic EC with a single shared population can be subject to fine-grained parallel implementation in which are one or more candidate solutions processed by a thread or a team of threads in parallel and only global operations are performed in a synchronized (sequential) manner. The straightforward approach in which the evaluation of candidate solutions is performed in parallel by multiple threads is also known as the *master-slave* model [17] and it has been successfully applied to the DE [18].

Multi-populational EC is known to improve the results of evolutionary search even when used as a part of sequential EC implementation [17]. Multi-populational EC operates with multiple sub-populations (islands, demes) that are evolved side by side [14], [16], [17], [19]. It can be implemented in parallel in a rather coarse-grained manner in which each population is processed by a thread or a team of threads. The evolution of individual populations may be completely independent, following the *independent runs model*. The populations may also interact following certain communication strategy e. g. by allowing migration of best candidate solutions between sub-populations according to selected migration topology at defined times as in the *distributed model* [17] / *islands model* [14].

Cellular EC combines the aspects of panmitic and multi-populational EC. Single global population is in cellular EC divided into overlapping neighborhoods in which the candidate solutions interact [17]. Good solutions are propagated (difused) through the population during the evolution. Another hybrid parallel EC models were developed, often in order to solve specific problems or to utilize special parallel platforms.

The implementation of various EC and differential evolution in parallel environments and on the GPUs in particular has been subject to a number of studies [20], [21], [8]. This study aims to implement and evaluate a DE that would seamlessly scale from shared memory systems to clusters of workstations with distributed memory.

III. Unified Parallel C

Unified parallel C is a parallel extension to ISO C that allows simple creation of parallel programs under the PGAS paradigm [2]. Each UPC program consists of a number of threads (processes) that can take part in a parallel computation by executing the same program logic on different portions of shared data. In contrast to the Single Instruction Multiple Data (SIMD) architecture of the GPUs, UPC threads hosted by many-core Shared Memory Multiprocessors (SMPs) and clusters do not need to execute the same instructions at the same time and usually follow the SPMD parallel processing model.

The memory available to a UPC program is structured according to the PGAS paradigm. UPC program has access to *private memory* and *shared memory*. Each UPC thread can use its own private memory that is located on the node that executes the thread and cannot be accessed by other threads, and shared memory providing the global address space. Shared memory is used for data sharing and inter-process communication. Shared memory (physically implemented as distributed shared memory) is allocated on the nodes that take part in the computation. Each thread has assigned a partition of shared memory that is physically local to the thread. Shared data placed in the local part of shared memory can be usually accessed from corresponding UPC thread in a much faster way than data located in non-local parts of shared memory [2]. Access to non-local shared data might involve network transfer and it could be significantly slower than local memory access. UPC program memory structure is illustrated in fig. 2.

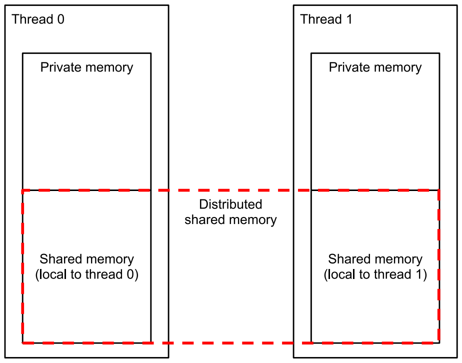
Thread 0
Private memory
Shared memory (local to thread 0)
Distributed shared memory
Thread 1
Private memory
Shared memory (local to thread 1)

Fig. 2: UPC program memory structure.

The data in UPC shared memory is partitioned so that each block of data has *affinity* (i.e. it is local) to certain UPC thread. Shared *scalars* have in the UPC affinity to the first thread (thread 0) so that thread 0 can read and write shared scalars at low cost of a local memory access whereas the access to shared scalars from other UPC threads might trigger network transfer. Shared *arrays* are organized into consecutive *blocks* that are affine to UPC threads in a cyclic manner [2], [22]. UPC threads should primarily process the blocks of shared data that are affine to them because only the access to local parts of shared memory is guaranteed to be efficient on all platforms and given any thread placement [2]. The default block size in UPC is 1, but it can be changed easily. An example of data affinity in a UPC application running with N threads on N hosts with default block size is shown in fig. 3. Shared variable *scalar* is affine to thread 0 and shared vector *array* of the length $2N$ is distributed across UPC threads by blocks of size 1 in a cyclic way. Besides memory sharing, the UPC provides a rich set of functions and language extensions (including collectives, barriers, parallel *for* statement etc.) that can be used to develop parallel applications with ease.

The UPC is supported by a number of compilers developed by different vendors and available on many HPC architectures [2], [22]. This study utilized the portable Berkeley UPC compiler [23]. Berkeley UPC compiler consists of several components including a UPC-to-C translator (this study used the GNU UPC-C translator [24]) and GASNet communication library. The GASNet library, developed at Berkeley Lab, is a language independent high-performance communication

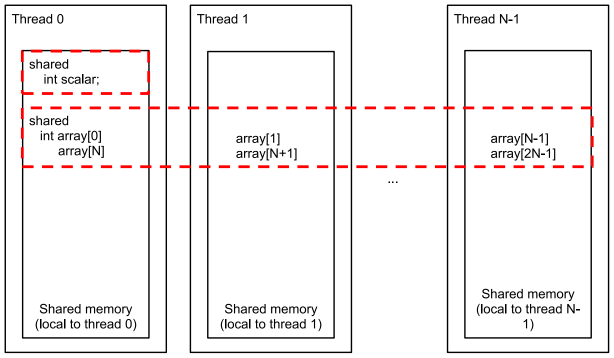

Fig. 3: Shared data affinity in UPC (default block size).

interface for PGAS languages [22], [25]. It provides a set of communication primitives that are used by compilers and it targets various types of networking hardware including high-performance interconnection networks (Gemini Scalable Interconnects, InfiniBand, Myrinet) that can be found in contemporary HPC clusters [22], [26], [27]. The UPC compiler allows selection of target networking environment (conduit) at compile time and a single UPC program can be compiled for a number of different parallel platforms. This feature significantly improves the flexibility and re-usability of UPC programs. Moreover, the GASNet provides a shared memory conduit, i.e. an efficient implementation of communication primitives for shared memory systems. In conclusion, a single UPC program can be compiled for a number of distributed memory platforms as well as for many-core SMPs.

IV. Differential Evolution in UPC

This study presents a design and implementation of three variants of scalable parallel DE for SMPs and clusters of computers with distributed memory in the UPC language. The first variant is a simple parallel implementation of panmitic DE with *shared population*. The iterations of the DE loop (lines 4-14 in algorithm 1) in each generation are performed in parallel by available UPC threads. In the second case, the shared population is divided into independent sub-populations and each of them is processed by single UPC thread following the *independent runs* model. The third variant of parallel DE implemented in course of this work is a simple *island* model created from the previous parallel DE by enabling periodical migration of best candidate vectors between sub-populations.

The parallel DEs were implemented in UPC as shown in algorithm 2. The algorithm describes main design principles behind the used parallel DE variants and implementation details are omitted. The population of candidate vectors was implemented as a shared array and distributed across UPC threads with block size equal to problem dimension (i.e. candidate vector length). The final population layout in UPC shared memory is shown in fig. 4 and fig. 5.

In each iteration, UPC threads processed the population of candidate vectors in parallel and accelerated the DE with respect to available hardware resources. Each thread selected random vectors for mutation either from the entire shared

Algorithm 2: An outline of Differential Evolution in UPC

```
1  Create population of candidate vectors in shared memory. One or more candidate
   vectors will be affine to each UPC thread (see fig. 4 for illustration of the
   memory layout);
2  Evaluate an objective function ranking the vectors in the population *in parallel*;
3  while *Termination criteria not satisfied* do
4      Each thread $T$ in parallel
5          for $i = \{1, \dots, M\} |\ v_i \in P^t$ *is affine to* $T$ do
6              Select random vectors $r_1, r_2, r_3 \in \{1, \dots, M\}$ for mutation
               so that:
7              if *shared_population* then
8                  $i \neq r_1 \neq r_2 \neq r_3$;
9              else
10                 $i \neq r_1 \neq r_2 \neq r_3$ and $r_1, r_2, r_3$ is affine to $T$;
11             end
12             Create trial vector $v_i^t$ according to eq. (2);
13             Validate the range of coordinates of $v_i^t$. Optionally adjust
               coordinates of $v_i^t$ so, that $v_i^t$ is valid solution to given problem;
14             Perform uniform crossover. Select randomly one parameter $l$ in
               $v_i^t$ and modify the trial vector using eq. (3);
15             Evaluate the trial vector.;
16             if *trial vector* $v_i^t$ *represent a better solution than population
               vector* $v^i$ then
17                 add $v_i^t$ to new population $P^{t+1}$
18             else
19                 add $v_i$ to new population $P^{t+1}$
20             end
21         end
22     end
23     Synchronize threads;
24     if *island_model* and *migration_gap_reached* then
25         Migrate best vectors between populations;
26     end
27 end
28 Find globally best solution across all threads $T$;
```

population or from the set of candidate vectors affine to that thread (sub-population, island). The latter random vector selection scheme contributed to the minimization of potential network communication when running on distributed memory systems.

Each thread also tracked the fitness of the best solution it has found so far. Because the $DE/rand/1$ variant of DE [11] was used, it was not necessary to share the information about globally best solution with other threads in course of the evolution and the selection of a globally best solution was performed only once at the end of the algorithm. Such a behavior reduced the amount of potential network communication as well.

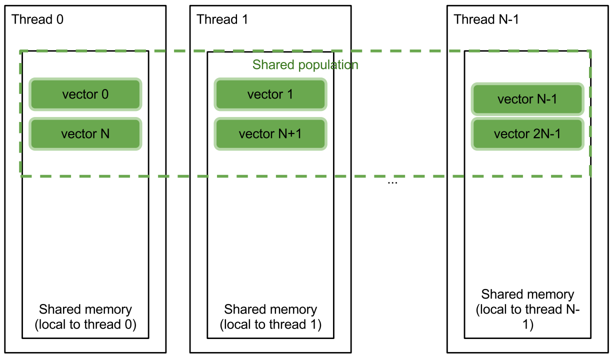
Thread 0
Thread 1
Thread N-1
Shared population
vector 0
vector N
vector 1
vector N+1
vector N-1
vector 2N-1
...
Shared memory (local to thread 0)
Shared memory (local to thread 1)
Shared memory (local to thread N-1)

Fig. 4: Affinity of DE vectors in shared memory.

The data exchange between UPC threads bears no communication overhead on shared memory systems and UPC threads can access candidate vectors affine to other threads with no performance penalty. On distributed memory platforms, however, can the UPC threads access at minimum cost only affine candidate vectors placed in local part of shared population. The stochastic nature of random vector selection in the DE, on the other hand, can result in scenarios in which only a small part of randomly selected vectors comes from non-local parts of shared memory and the performance of the algorithms is not affected significantly.

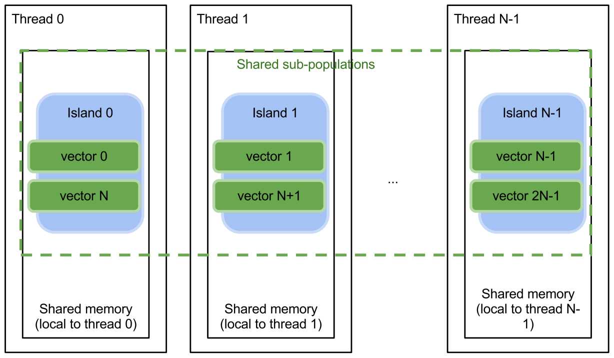

Fig. 5: DE vector layout for the *independent runs* and *island* models.

V. Experiments

The three parallel DE models outlined in section IV were implemented in UPC in order to verify the ease of implementation, flexibility, and scalability of UPC-based parallel EC.

The DE code was based on an sequential C++ code which was first ported to C and then to UPC. The porting to C was necessary because the original code base was in C++ and UPC is a C dialect but the transition between C and UPC required only a small programming effort and the parallel implementation was identical to the original sequential code in C to a large extent. Inter-process data sharing and communication was provided automatically by the UPC compiler.

Computational experiments were conducted using a SMP and a distributed memory HPC cluster. The SMP contained 2 AMD Opteron 6176 SE CPUs at 2.3 GHz with 12 cores per CPU (i.e. with 24 cores in total) and 128 GiB of main memory. The HPC cluster was a HP c-class BladeSystem composed of compute nodes with 2 Intel Xeon QuadCore CPUs at 2.5 GHz with 4 cores per CPU and 18 GiB of main memory connected via a 4X DDR IB Mezzanine HCA InfiniBand Interconnect (20Gb/s full-duplex). In the cluster, each UPC thread was placed on a separate compute node to simulate the worst-case thread placement scenario (from the communication point of view) in which access to every non-local shared variable yielded network communication. Moreover, the cluster was at the time of experiments used by other applications so the test conditions were typical for a real-world shared HPC environment.

Three well known real-parameter optimization test functions were used as test problems [28]. Ackley's function (5) with parameter range [-30,30], Griewank's function (6) with parameter range [-600,600], and Sphere function (7) with parameter range [-100, 100] were selected.

$$f_1(\mathbf{x}) = -20 \cdot exp\left(-0.2\sqrt{\frac{1}{n} \cdot \sum_{i=1}^{n} x_i^2}\right) - exp\left(\frac{1}{n}\sum_{i=1}^{n} cos(2\pi x_i)\right) + 20 + e \quad (5)$$

where n is problem dimension, $\mathbf{x} = (x_1, \ldots, x_n)$ is parameter vector, $e \approx 2.71828$ is Euler's number, and $exp(a) = e^a$ is exponential function.

$$f_2(\mathbf{x}) = 1 + \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} cos\left(\frac{x_i}{\sqrt{i}}\right) \quad (6)$$

$$f_3(\mathbf{x}) = \sum_{i=1}^{n} x_i^2 \quad (7)$$

The dimension of the test functions was set to 5000 in order to establish high-dimensional problems with high computational costs, known properties and known optima ($\mathbf{x}_{f_1}^{opt} = \mathbf{x}_{f_2}^{opt} = \mathbf{x}_{f_3}^{opt} = (0, \ldots, 0)$).

Parameters of the DE were selected according to best practices [11], trial-and-error tests, and previous experience with the algorithm. Overview of DE parameters is provided in table I.The size of sub-populations for the *independent runs* and *island* models was set to $\frac{800}{threads}$. Migration gap and migration rate of the island model were set to 20 generations (i.e. vectors migrated between populations after 20 generations) and 10 % of the sub-population size (i.e. 10 % of the best vectors of each sub-population got the chance to migrate to the next sub-population) respectively. Migration topology was ring.

Parameter	Value
Algorithm	$DE/rand/1$
Test problem	f_1, f_2, f_3
Problem dimension	5000
Population size	800
Number of generations	2000
Scaling factor F	0.9
Crossover rate C	0.9

TABLE I: DE parameters.

All experiments were repeated 10 times due to the stochastic nature of the DE and presented values are mean results of the 10 independent runs.

Execution times of the three DE variants implemented in UPC on the shared memory system and on the HPC cluster are shown in table II and table III respectively. The tables show average execution time needed to process 2000 generations for all test functions when using independent runs model (column *Ind. runs*), shared population model (column *Shared pop.*), and island model (column *Island*). The single thread case (first row) refers to sequential execution of the DE which was the same for all models.

The DE has found global optima of all three test functions in all test runs on both platforms. The results in table II and table III show that there were no large differences in execution times when using different parallel DE models. The UPC implementation of all parallel DE models scaled on both test environments in a similar manner and proved to be portable and scalable. Because the performance was the same for all three implemented models, the choice of parallel DE model can be driven purely by application requirements. However, the speedup and scaling reported in this work is rather illustrative and might differ on other parallel environments due to different performance and configuration of hardware resources.

VI. Conclusions

This study proposed and evaluated an implementation of three parallel DE variants for many-core shared memory systems and distributed memory clusters in the UPC programming language. The PGAS programming model and the features of the UPC language together with the capabilities of the GASNet communication library allowed the creation of portable parallel DE implementation that can be reused for multiple parallel systems. Computational experiments performed in two types of parallel environments under real-world conditions have shown that the implementation scaled well and did not require substantial modifications (besides the implementation of vector migration) to the original sequential algorithm. The different parallel DE models implemented in this work were all able to find global optima of selected high-dimensional test functions and the UPC has shown a good potential for simple implementation of EC methods in parallel environments that would enable practical use of EC methods for high-dimensional problems.

Other evolutionary computation methods can be implemented in UPC in a similar way as the DE. Our future work will focus on the implementation of further evolutionary and swarm algorithms for high performance computing environments.

Acknowledgement

This work was supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070) and by the Bio-Inspired Methods: research, development and knowledge transfer project, reg. no. CZ.1.07/2.3.00/20.0073 funded by Operational Programme Education for Competitiveness, co-financed by ESF and state budget of the Czech Republic. This work is also partially supported by Grant of SGS No. SP2013/70, VŠB - Technical University of Ostrava, Czech Republic.

References

[1] D. Padua, *Encyclopedia of Parallel Computing*, ser. Springer Series in Optical Sciences. Springer, 2012. [Online]. Available: http://books.google.cz/books?id=S6E7cM0Yb9AC

[2] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: Distributed Shared Memory Programming*, ser. Wiley Series on Parallel and Distributed Computing. Wiley, 2005. [Online]. Available: http://books.google.cz/books?id=c0_GVfK7sf8C

	Average DE execution time [s] for f_1			Average DE execution time [s] for f_2			Average DE execution time [s] for f_3		
Threads	Ind. runs	Shared pop.	Island	Ind. runs	Shared pop.	Island	Ind. runs	Shared pop.	Island
1	1784.730	N/A	N/A	1588.670	N/A	N/A	1188.801	N/A	N/A
3	601.980	602.339	612.200	536.752	539.972	551.005	390.365	387.241	394.079
6	300.700	300.986	305.937	270.512	270.700	274.737	196.424	193.767	195.948
12	147.197	146.688	152.368	131.427	131.205	136.502	98.135	96.475	98.374
24	73.645	72.548	75.941	65.598	65.611	68.138	48.827	57.642	49.258

TABLE II: DE execution time on SMP.

	Average DE execution time [s] for f_1			Average DE execution time [s] for f_2			Average DE execution time [s] for f_3		
Threads	Ind. runs	Shared pop.	Island	Ind. runs	Shared pop.	Island	Ind. runs	Shared pop.	Island
1	718.008	N/A	N/A	873.005	N/A	N/A	654.185	N/A	N/A
3	242.483	245.716	237.064	293.618	295.598	307.932	220.793	219.090	214.677
6	117.637	122.398	121.870	152.169	153.377	151.891	121.027	112.965	107.491
12	62.4546	58.792	64.588	74.2633	71.7392	81.941	59.322	53.023	53.261

TABLE III: DE execution time on cluster.

[3] J. González-Domínguez, M. J. Martín, G. L. Taboada, J. Touriño, R. Doallo, D. A. Mallón, and B. Wibecan, "Upcblas: a library for parallel matrix computations in unified parallel c," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 14, pp. 1645–1667, 2012.

[4] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda, "An evaluation of global address space languages: co-array fortran and unified parallel c," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '05. New York, NY, USA: ACM, 2005, pp. 36–47. [Online]. Available: http://doi.acm.org/10.1145/1065944.1065950

[5] M. Miller, *Implementing and Understanding Algorithms in Unified Parallel C*. Michigan Technological University, 2008. [Online]. Available: http://books.google.cz/books?id=Yc8tPwAACAAJ

[6] L. Chen, L. Liu, S. Tang, L. Huang, Z. Jing, S. Xu, D. Zhang, and B. Shou, "Unified parallel c for gpu clusters: language extensions and compiler implementation," in *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, ser. LCPC'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 151–165. [Online]. Available: http://dl.acm.org/citation.cfm?id=1964536.1964547

[7] Silicon Graphics International Corp. , "SGI announces software support for intel xeon phi," Press Release, January 2013, http://www.sgi.com/company_info/ newsroom/press_releases/ 2013/january/intel.html.

[8] P. Krömer, V. Snášel, J. Platoš, and A. Abraham, "Many-threaded implementation of differential evolution for the cuda platform," in *GECCO*, N. Krasnogor and P. L. Lanzi, Eds. ACM, 2011, pp. 1595–1602.

[9] K. Tagawa, "Concurrent Differential Evolution Based on Generational Model for Multi-core CPUs," in *SEAL*, ser. Lecture Notes in Computer Science, L. T. Bui, Y.-S. Ong, N. X. Hoai, H. Ishibuchi, and P. N. Suganthan, Eds., vol. 7673. Springer, 2012, pp. 12–21.

[10] H. Wang, S. Rahnamayan, and Z. Wu, "Parallel differential evolution with self-adapting control parameters and generalized opposition-based learning for solving high-dimensional optimization problems," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 62 – 73, 2013, metaheuristics on GPUs. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731512000639

[11] K. V. Price, R. M. Storn, and J. A. Lampinen, *Differential Evolution A Practical Approach to Global Optimization*, ser. Natural Computing Series, G. Rozenberg, T. Bäck, A. E. Eiben, J. N. Kok, and H. P. Spaink, Eds. Berlin, Germany: Springer-Verlag, 2005. [Online]. Available: http://www.springer.com/west/home/computer/foundations?SGWID=4-156-22-32104365-0&#38;teaserId=68063&#38;CENTER_ID=69103

[12] R. Storn and K. Price, "Differential Evolution- A Simple and Efficient Adaptive Scheme for Global Optimization over Continuous Spaces," Tech. Rep., 1995. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.1.9696

[13] R. Storn, "Differential evolution design of an IIR-filter," in *Proceeding of the IEEE Conference on Evolutionary Computation ICEC*. IEEE Press, 1996, pp. 268–273.

[14] A. Engelbrecht, *Computational Intelligence: An Introduction, 2nd Edition*. New York, NY, USA: Wiley, 2007.

[15] J. E. Baker, "Reducing bias and inefficiency in the selection algorithm," in *Genetic algorithms and their applications : proceedings of the second International Conference on Genetic Algorithms*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1987, pp. 14–21. [Online]. Available: http://dl.acm.org/citation.cfm?id=42512.42515

[16] E. Alba, E.-G. Talbi, G. Luque, and N. Melab, *Metaheuristics and Parallelism*. John Wiley & Sons, Inc., 2005, pp. 79–103. [Online]. Available: http://dx.doi.org/10.1002/0471739383.ch4

[17] G. Luque and E. Alba, *Parallel Genetic Algorithms: Theory and Real World Applications*, ser. Studies in Computational Intelligence. Springer, 2011. [Online]. Available: http://books.google.cz/books?id=qIbjNcMJWJsC

[18] D. Tasoulis, N. Pavlidis, V. Plagianakos, and M. Vrahatis, "Parallel differential evolution," in *Evolutionary Computation, 2004. CEC2004. Congress on*, vol. 2, June, pp. 2023–2029 Vol.2.

[19] M. Affenzeller, S. Winkler, S. Wagner, and A. Beham, *Genetic Algorithms and Genetic Programming: Modern Concepts and Practical Applications*. Chapman & Hall/CRC, 2009.

[20] D. Tasoulis, N. Pavlidis, V. Plagianakos, and M. Vrahatis, "Parallel differential evolution," in *Evolutionary Computation, 2004. CEC2004. Congress on*, vol. 2. IEEE, 2004, pp. 2023–2029.

[21] K.-L. Fok, T.-T. Wong, and M.-L. Wong, "Evolutionary computing on consumer graphics hardware," *Intelligent Systems, IEEE*, vol. 22, no. 2, pp. 69–78, 2007.

[22] C. Bell, W.-Y. Chen, D. Bonachea, and K. Yelick, "Evaluating support for global address space languages on the cray x1," in *Proceedings of the 18th annual international conference on Supercomputing*, ser. ICS '04. New York, NY, USA: ACM, 2004, pp. 184–195. [Online]. Available: http://doi.acm.org/10.1145/1006209.1006236

[23] UC Berkeley, *Berkeley UPC - Unified Parallel C*, 2013, http://upc.lbl.gov/. [Online]. Available: http://upc.lbl.gov/

[24] GNU, *GNU Unified Parallel C*, 2013, http://www.gccupc.org/. [Online]. Available: http://www.gccupc.org/

[25] R. Nishtala, Y. Zheng, P. Hargrove, and K. Yelick, "Tuning collective communication for partitioned global address space programming models," *Parallel Computing*, vol. 37, no. 9, pp. 576–591, 2011. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-80052023951partnerID=40md5=4b7497f7caa3ec3e3a3a12cc840bbc20

[26] M. Farreras and G. Almasi, "Asynchronous pgas runtime for myrinet networks," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS '10. New York, NY, USA: ACM, 2010, pp. 4:1–4:10. [Online]. Available: http://doi.acm.org/10.1145/2020373.2020377

[27] J. Jose, M. Luo, S. Sur, and D. K. Panda, "Unifying upc and mpi runtimes: experience with mvapich," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*,

ser. PGAS '10. New York, NY, USA: ACM, 2010, pp. 5:1–5:10. [Online]. Available: http://doi.acm.org/10.1145/2020373.2020378

[28] P. N. Suganthan, N. Hansen, J. J. Liang, K. Deb, Y. P. Chen, A. Auger, and S. Tiwari, "Problem definitions and evaluation criteria for the CEC 2005 Special Session on Real Parameter Optimization," Nanyang Technological University, Tech. Rep., 2005.