Open access • Proceedings Article • DOI:10.1109/CEC.2010.5586158

Sferes v2 : Evolvin' in the multi-core world — Source link

Jean-Baptiste Mouret, Stéphane Doncieux

Institutions: Pierre-and-Marie-Curie University

Published on: 18 Jul 2010 - Congress on Evolutionary Computation

Topics: Evolutionary algorithm, Evolutionary computation, Benchmark (computing), Fitness function and Single-core

Sferes_v2: Evolvin' in the Multi-Core World

Jean-Baptiste Mouret, Stéphane Doncieux

Sferes$_{\text{v2}}$: Evolvin' in the Multi-Core World

Jean-Baptiste Mouret and Stéphane Doncieux

***Abstract*— This paper introduces and benchmarks Sferes$_{\text{v2}}$, a C++ framework designed to help researchers in evolutionary computation to make their code run as fast as possible on a multi-core computer. It is based on three main concepts: (1) including multi-core optimizations from the start of the design process; (2) providing state-of-the art implementations of well-selected current evolutionary algorithms (EA), and especially multiobjective EAs; (3) being based on modern (template-based) C++ techniques to be both abstract and efficient. Benchmark results show that when a single core is used, running time of classic EAs included in Sferes$_{\text{v2}}$ (NSGA-2 and CMA-ES) are of the same order of magnitude than specialized C code. When n cores are used, typical speed-ups range from $0.75n$ to $0.9n$; however, parallelization efficiency critically depends on the time to evaluate the fitness function.**

I. Introduction

Most evolutionary computation (EC) users experimented how evolutionary algorithms (EAs) require a lot of computational power. The picture is worse for most EC researchers who have to repeatedly run their algorithm, be it for debugging their algorithm or for establishing statistically valid benchmarks. As a result, in EC research—contrary to many other research fields—faster implementations of the same algorithms *can* make research more effective.

Fortunately for EC research, Moore's observation that the amount of computation achievable on a single chip doubles every two years has been followed with a remarkable precision [1]. Manufacturers of micro-processors are nonetheless reaching physical limits in improving the core of their chips. Desirous to continue to follow Moore's law, they now increase the number of cores available in a single chip [2], thus making parallel computing mainstream. Concretely, there is a large probability that you are now using a multi-core computer as your main development environment; hence, the main question is: are you able to fully exploit this computational power to improve your researches? If not, we introduce in this paper Sferes$_{\text{v2}}$[1]: a C++ framework—a set of classes and compilation tools—that aims at helping EC researchers to implement efficient versions of their ideas in a world of multi-core computers.

As this framework targets scientists, it is assumed that users will have to design new genetic operators, new algorithms, complex fitness, etc. We therefore focused on allowing to easily write new code, with as little efforts as possible. One of the most common approaches to tackle this challenge is to rely on object-oriented programming (see e.g. [4]) but this abstraction model implies substantial run-time costs [5], [6]. Nonetheless, the C++ community now proposes a large set of "modern C++ techniques" (template-based generic programming) [7], [8], [9] that promise to combine the efficiency of hand-tuned C code with the abstraction of object-oriented programming. Besides bringing multi-core programming to EC researchers, one of the goals of Sferes$_{\text{v2}}$ is to make these techniques easy to use in an EC context.

This line of thought leads us to the following main objectives for Sferes$_{\text{v2}}$:

- Be multi-core from the ground-up: include multi-core optimizations from the start of the design process.
- Be up-to-date and multiobjective: provide a few but well selected implementations of "modern" EAs, and especially multiobjective EAs (MOEA) [10].
- Be based on modern C++ techniques to be both abstract and efficient.

Additionally, we set the following goals, which can be summarized by "the framework should be a good and simple piece of software":

- Be extensible: adding new algorithms should be straightforward.
- Be simple: simple experiments (e.g. optimizing real parameters) should be simple to set-up and require the minimum amount of code.
- Put the emphasize on efficient implementations rather than on covering the largest number of algorithms.
- Be small: the source code should be a short as possible to allow new users to quickly master the library and to make maintenance easier.
- Be tested: each important feature should be accompanied by a unit test.
- Be portable to all current Unix flavors (especially GNU/Linux and MacOSX);
- Be open source (GPL-compatible).
- Be easy to interface with current existing code (especially for fitness functions).

The purpose of this article is threefold: (1) introduce the Sferes$_{\text{v2}}$ framework, which can be useful to the EC community; (2) argue our technical choices to fulfill the previously described goals; (3) answer the question: what speed-up/slow-down can we expect in EC by employing modern C++ and by explicitly designing for multi-core systems? Additionally, this paper may suggest some useful ideas to designers of present and future EC software.

Jean-Baptiste Mouret and Stéphane Doncieux are with the Université Pierre et Marie Curie (UPMC) - Paris 6, Institut des Systèmes Intelligents et de Robotique (ISIR), CNRS UMR 7222, F-75005, Paris, France; e-mail: {mouret,doncieux}@isir.upmc.fr

[1]The acronym SFERES stands for "SFERES is a Framework for Encouraging Research on Evolution and Simulation". The version introduced in this paper is the second one, in which the simulation part has been removed. The first version, which shares no code with Sferes$_{\text{v2}}$, has been described in [3].

II. Related Work

Despite the simplicity of the principles underlying EAs, the many variants of the same EA (huge number of potential genetic operators, various representations for solutions, etc.) make most EC programs complex pieces of software. In many ways, designing an EC software is like playing with a Lego™ set in which each brick is a part of the "EC toolbox" and in which some bricks (e.g. fitness function) have to be specially built. This complexity of EC software lead many research teams to mutualize their code by incorporating it in software frameworks that aims at reducing the amount of work required to implement a new EC experiment.

For instance, OpenBeagle [11], [12] is an object-oriented C++ framework "designed to provide an EC environment that is generic, user friendly, portable, efficient, robust, elegant and free"[2]. This large framework (about 40,000 lines of code[3]) relies on classic OOP and, to our knowledge, is not specially designed for multi-core (however, it is able to employ several threads by using an optional add-on).

Evolving Objects[4] (EO) [13] is another popular framework in C++. Programmed with template-based C++, it has been recently extended to implement many MOEAs [14]. On top of EO, Paradiseo [15] implements distributed and parallel algorithms. Overall, EO and its extensions appears to be a powerful but large (about 60, 000 lines of code) and complex piece of software that targets more clusters than mainstream multi-core workstations.

Numerous other frameworks for EC have been released on the Internet. Some of them implement parallel algorithms, for instance the DREAM project [16] and MALLABA [17]. However, like Paradiseo, the emphasis is more on parallel computation on large clusters than on providing a simple approach to implement EC on a multi-core workstation.

III. Architecture

A. Main parts

Sferes$_{v2}$ is divided into three main parts: (1) the framework, (2) optional modules (that may be experimental or require complex dependencies) and (3) the user experiments (that may include new genotypes, fitness, ...). Everything is compiled using the waf[5] build tool, supplemented with functions to easily compile user experiments and modules modules.

The framework itself (Fig. 1) is divided into the classic parts of evolutionary algorithms:

1) Evolutionary algorithms (EA): the main evolutionary loops. Different single and multi-objective algorithms are available (see next subsection).
2) Genotypes: data and genetic operators (mutation and cross-over).

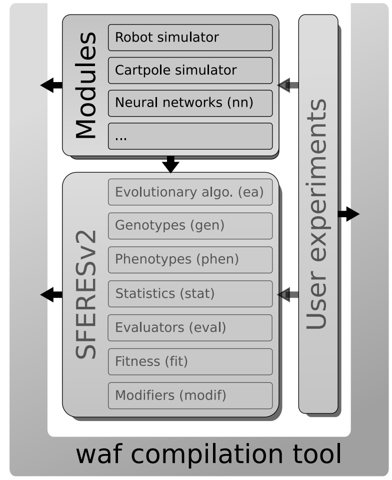


Fig. 1. Architecture of Sferes$_{v2}$. Optional modules can be added to the core framework (SFERESv2); each user experiment is a C++ file compiled with the framework. The waf compilation tool is used to easily compile user-defined experiments and optional modules.

3) Phenotypes: the transformed genotype to be used in the fitness function (e.g. integer parameters from a bit-string or developed neural networks from an indirect encoding).
4) Statistics: observers of the current population (best fitness, Pareto front, ...). They are the only data written on the hard-disk, so they are also the standard way to gather results.
5) Evaluators: evaluators take a list of genotypes, develop them into the corresponding phenotypes and compute the fitness function for each phenotype. Depending on the kind of evaluator, this evaluation can be parallel or sequential.
6) Fitness: abstract classes to program new fitness functions.
7) Modifiers: functors to modify fitness function once the whole population has been evaluated. For instance, this allows to implement fitness sharing [18], behavioral diversity [19] or novelty search [20].

B. Implemented algorithms

Sferes$_{v2}$ is more focused on implementing the most efficient algorithms from the literature than on providing an exhaustive list of algorithms. Such a choice keeps the complexity of the framework and maintenance costs as low as possible: it is easier to optimize and test a few useful algorithms than a lot of algorithms almost never used (and therefore less tested).

At this time, the following algorithms are available (other ones are available as modules):

- NSGA-2 [21]: one of the most efficient multi-objective evolutionary algorithms [22]. Contrary to classic implementations of NSGA-2, which have a run-time complexity of $O(GMN^2)$ [21] (where G is the number of generations, M the number of objectives and N the size of the population), Sferes$_{v2}$ implements the fast

[2] http://beagle.gel.ulaval.ca/

[3] The number of lines of code is computed using David A. Wheeler's "SLOCCount", http://www.dwheeler.com/sloccount/

[4] http://paradiseo.gforge.inria.fr

[5] http://code.google.com/waf: A modern replacement to automake/autoconf/make written in Python.

dominance sorting algorithm introduced in [23], which has a complexity of $O(GN \log^{M-1} N)$. NSGA-2 can also be employed in single-objective optimization (it is then equivalent to a tournament-based evolutionary algorithm).
- ε-MOEA [24]: a recent multi-objective evolutionary algorithm based on ε-dominance. It is often more efficient than NSGA-II [24] but it is less versatile because it is archive-based; for instance, multi-objective diversity preservation approaches [19] are not compatible with archive-based algorithms.
- CMA-ES [25]: one of the most efficient single-objective evolutionary strategies; it is one of the best choices to optimize real numbers with a single objective (MO-CMA-ES [26], the multiobjective CMA-ES, is available as an experimental module).
- a simple rank-based EA: this algorithm is a reference implementation for new evolutionary algorithms in Sferes$_{v2}$.

The following basic genotypes/phenotypes are available:

- Binary strings;
- Real numbers with the following operators:
  - Gaussian mutation;
  - Uniform mutation;
  - Polynomial mutation [27];
  - SBX cross-over [27];
  - Exchange parameters during cross-overs.

Some other genotypes and phenotypes are available as modules. The most notable one is the direct encoding of neural networks [28], a simple encoding to evolve the topology and the parameters of neural networks.

C. Main classes

The main classes are organized in a classic object-oriented manner, with abstract classes for each of the previously defined main concepts, which are derived for each particular implementation (NSGA-2, bit-string, ...). Fig. 2 is a simplified UML diagram. Each class is then combined as Lego™ bricks by the user to form a particular experiment (Fig. 3(a)). Fig. 3(b) offers a more dynamic view by showing how each class uses the other ones during the evolutionary loop.

IV. Implementation

A. Technical choices

The efficiency of C++ compilers and the large number of available libraries (both for scientific computing and for physically simulating agents, an important part of the authors' research activities) lead us to choose C++. While the use of languages such as Java has been considered, modern C++ techniques makes C++ the most promising combination of abstraction without any compromise on efficiency.

The Message Passing Interface (MPI) [30] is often the tool of choice when parallelizing evolutionary algorithms. It is, for instance, the underlying interface employed by ParadisEO [15]. MPI processes exchange data (genotypes, fitness) via simple messages (typically strings of bytes) transported on networks or unix sockets. This makes MPI the standard approach to program clusters. However, transforming genotypes to strings and then deciphering messages can be a costly operation, especially for genotypes such as graphs or trees. On multi-core and multi-processor systems, all cores share the same memory and consequently all of them can directly access genotypes. Consequently, since Sferes$_{v2}$ mainly targets multi-core machines, we focused on shared memory parallelization. At least three main options are available:

- Posix threads (or multi-platform abstractions of threads such as boost::thread);
- OpenMP [31];
- Intel TBB[6][32].

Posix threads are standards but they are not easy to use in C++ (because it is a C API based on function pointers) and some desirable properties, for instance automatically selecting the number of threads according to the number of cores, are not built-in. Moreover, they don't propose high-level functions to easily parallelize a loop or a sort.

OpenMP [31] is the industry-standard API for shared-memory parallel computing. It employs annotations (#pragma) in the source code to describe how to parallelize loops. It has been more designed for C than C++ and requires an OpenMP-enabled compiler (GCC compiles OpenMP code only since a few years).

Intel TBB [32] is a recent alternative freely (GPL-compatible) provided by Intel to program multi-core systems. It relies on modern C++ techniques and provides an easy way to parallelize loops and sort procedures with many automatic optimizations (such as finding the optimal number of cores and automatically dividing data between them). Overall, the programming style of TBB and its features suits well the Sferes$_{v2}$ programming approach; that is why we chose to use this library.

MPI is also available as an optional add-on to run optimization of very costly fitness functions on clusters.

Having chosen the C++ language and a parallelization layer, efficient C++ libraries have to be selected to write genotypes on the file system (to save the results) and to efficiently manipulate matrix/vectors. Using existing libraries instead of designing our own has the obvious advantage of keeping the source code of the framework as small as possible. The following libraries were chosen:

- Boost[7][8], [9], a set of peer-reviewed libraries that provide to Sferes$_{v2}$:
  - memory management (via shared pointers);
  - serialization, to write best genotypes on the hard disk;
  - system abstraction (creating directories, etc.);
  - unit test framework.

[6]http://www.threadbuildingblocks.org
[7]http://www.boost.org

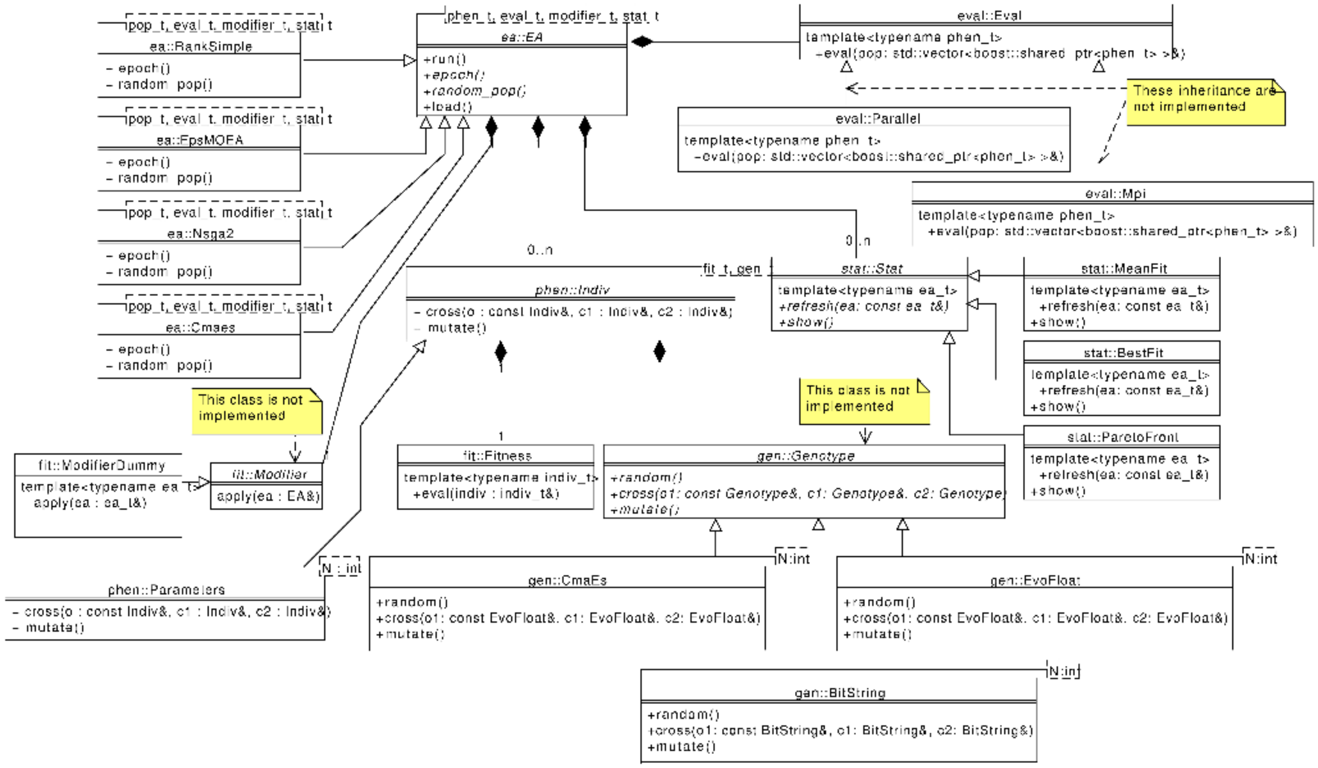
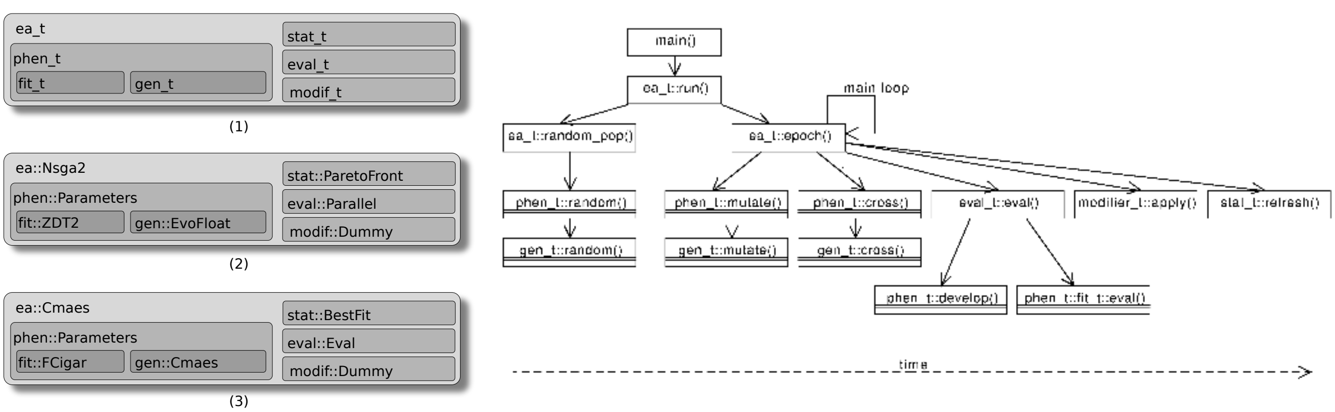


Fig. 2. UML class diagram of the main classes of Sferes$_{v2}$. Some abstract classes are not implemented because template-based C++ [29] does not require them. However, these classes are implicit in the source code and the user should assume they exist. The main algorithms derive from ea::EA. Parallel evaluation is handled by eval::Parallel.



(a) (b)

Fig. 3. (a) The Sferes$_{v2}$ building set (1) General template. The user first selects a fitness function (fit_t) and a genotype type (gen_t); they are then combined with a phenotype type (phen_t); the user then selects a statistics list (stat_t), an evalutator type (eval_t) and a modifier (modif_t); last, each type is combined with an evolutionary algorithm type (ea_t) to obtain a full specialized algorithm for the particular experiment. (2) Example of an experiment that optimize the fitness ZDT2 with NSGA-2 [21] and a parallel (multi-core) evaluator (see section V-A). (3) Example of an experiment that use CMA-ES [25] to optimize the function f_{cigar} (see section V-B). (b) Overview of how Sferes$_{v2}$'s objects interact. The main function first calls the run() method of the defined EA class. This method initializes the population and enters the classic chain of mutation / cross-over / evaluation / modifier. Selection is handled in ea_t::epoch().

```
template <class Derived> struct Base {
  void interface() {
      static_cast<Derived*>(this)->implementation();
  }
  void f() {
    for(int i = 0; i < 3; ++i) interface();
  }
};
struct Derived : Base<Derived> {
  void implementation() { std::cout<<"impl"<<std::endl; }
};
```

Fig. 4. The "curiously recurring template pattern" [34]. This code mimics a virtual call of an abstract method: in the abstract class (Base), a generic algorithm can use methods which are only defined in the derived classes. Extending this pattern to draw a full hierarchy of classes requires more work; Sferes$_{v2}$ relies on the method described in [29].

- Eigen2[8], a fast linear algebra library based on *expression templates* [33]. It also supports SSE vector optimizations.

Thanks to these high-level libraries, the core of sferes is made of only 4000 lines of code (LOC) and the test suite of 1200 LOC (compare to about 60,000 LOC in ParadiseEO [11] or to about 40,000 in OpenBeagle [13]).

B. Static Object Oriented C++

Object-oriented programming (OOP) has many advantages that are desirable for complex high-level code. However, classic OOP (e.g. in C++ or in Java) heavily relies on virtual methods to implement abstraction and polymorphism. Such an approach has a significant run-time overhead [5]:

- selecting the appropriate method to call requires an indirection (the software has to check the virtual table);
- abstract/virtual methods cannot be inlined, resulting in a overhead for very simple methods (e.g. setters/getters).

As a consequence, the use of virtual methods is one of the main performance issues in object-oriented programming. For instance, in an image processing benchmark, a speed-up by a factor 3 has been measured by only getting rid of virtual methods [6]. Nevertheless, the wide success of OOP in scientific computing demonstrates the need of high-level mechanisms such as polymorphism.

The success of the Standard Template Library (STL) [35], which manages to combine efficiency and abstraction, inspired many work in *generic programming in C++* [7], [8], [9], [35], a set of C++ techniques that rely on the "template" keyword for abstraction. Among the proposed ideas, the "curiously recurring template pattern" [34] (Fig. 4) mimics polymorphism without the cost of virtual methods. This principle can be extended to a whole hierarchy of classes, leading to "static C++ Object-Oriented Programming" (SCOOP) [29]. In Sferes$_{v2}$, all classes are implemented using this programming style; no virtual methods are employed. This mechanics is hidden in Sferes$_{v2}$ by some pre-processor macros so that users can use static OOP mostly like classic OOP.

[8]http://www.eigen2.org

C. Multi-Core parallelization

Evolutionary algorithms are parallel in essence: the evaluation of each individual is independent from the evaluation of the other ones. In multi-core programming, all cores share the same memory so in first approximation there is no cost to evaluate each individual on a different core. This scheme is close to the classic master-slave distribution [36]; this is what is implemented in Sferes$_{v2}$, mainly because of its simplicity. Nevertheless, it should be noted that:

- This parallelization is more efficient if the computation of the fitness function is slow. This is often the case in real-world EC and for instance in evolutionary robotics but it is not in all benchmarks.
- This parallelization assumes that the cost of distributing the computation (which is low because all cores access the same memory but is not zero because the data have to be split, threads have to be handled and memory bandwidth is shared between cores) is lower than the cost of evaluating the fitness of the whole population. This can be a problem when very small population are used, as in CMA-ES [25].
- This approach to parallelization is designed for generational evolutionary algorithms. Steady state algorithms can also be used but only if a group of individuals is created at each iteration.

The sorting procedures, widely used in EA, can also be parallelized and TBB provides built-in parallel sort procedures. However, this parallelization should be useful only with large populations.

Last, many algorithms use loops that compute a value for each individual that is independent from the value corresponding to other individuals. This is for instance the case of the crowd assignment procedure in NSGA-2 [21]. These loops can be easily parallelized.

D. Compile-time configuration

Evolutionary algorithms have a lot of parameters (e.g. population size, mutation rate, etc.) and a convenient way to set them is needed. A configuration file (e.g. an XML file) is often used. Such a method has, however, several drawbacks:

- some code to read the files has to be written and kept synchronized with the definition of classes; this has to be done each time the framework is extended for a particular EC experiment.
- parameters are unknown at compile time so some checks (e.g. if (mutation_type == x) else ...) have to be done many times whereas they are useless;
- users cannot rely on the compiler to detect type errors nor exploit it to define complex configuration schemes (such as including the same set of parameters in several experiments).

The last drawback encourages many software designers to progressively increase the expressive power of their configuration language until it reaches the expressiveness and complexity of classic programming languages. This observation suggests a radical point of view: why not use the

```
struct Params
{
  struct pop
  {
    static const unsigned size = 200;
    // ...
  };
};
```

Fig. 5. Example of static parameter setting.

TABLE I

MEAN RUNNING TIME AND STANDARD DEVIATION TO OPTIMIZE ZDT2 WITH NSGA-2, WITH REGARD TO TIME SPENT IN FITNESS.

Implem./fit. slowdown	0 μs (s.d.)	500 μs (s.d.)	2000 μs (s.d.)
Deb	2.45s (0.41)	79.26s (0.03)	304.3s (0.04)
Sferes$_{v2}$ (1 core)	2.92s (0.3)	80.09s (0.04)	305.13s (0.04)
Sferes$_{v2}$ (2 cores)	7.75s (0.94)	47.11s (0.81)	159.24s (0.99)
Sferes$_{v2}$ (3 cores)	8.3s (0.7)	35.35s (0.41)	110.26s (0.1)
Sferes$_{v2}$ (4 cores)	7.8s (0.52)	28.79s (0.1)	85.17s (0.06)

programming language used in the framework as a configuration language? Combined with static programming, this also avoids the first two drawbacks: only the minimum amount of code (the declaration of the parameter) has to be written and, provided that these parameters are defined as constants, the compiler can propagate parameters to the whole program, hence removing useless branches and optimizing as much as possible. Incidentally, writing the parameters in C++ allows to specify them in the same file than fitness functions and algorithm specification. This makes it possible to define a whole experiment in a single file.

This approach of parameter setting obviously imposes to recompile the source code for each parameter change, a step that can be substantially long with template-based generic programming. However, given that many EC experiments need several hours of computation, we assumed that a substantial improvement in simplicity and in speed counterbalances the time to compile the program.

From a practical viewpoint, Sferes$_{v2}$ parameters are defined at compile time using a structure that contains only constants. This structure is passed to all Sferes$_{v2}$ classes so they can access the parameters. This method allows to avoid to write read/write code for parameters. It also allows the compiler to propagate constants and settings in the whole source code, resulting in an executable optimized for the specific parameters. Fig. 5 is an example of such a static parameter setting in Sferes$_{v2}$.

V. BENCHMARKS

To evaluate the efficiency of Sferes$_{v2}$, we measured the speed achieved by Sferes$_{v2}$ and reference implementations in two typical scenarios[9]: a two objectives problem optimized by NSGA-II [21] and a single objective one optimized by CMA-ES [25]. The following questions have been tackled:

- Is Sferes$_{v2}$ as fast than hand-tuned C code when a single core is used?
- How does the performance vary when the number of cores is increased?
- How is multi-core efficiency modified by the time to evaluate the fitness?

A. NSGA-2

a) Experimental setup: NSGA-2 [21] is one of the most popular multiobjective evolutionary algorithms. To evaluate its implementation in Sferes$_{v2}$, the ZDT2 function [22], a classic two-objective benchmark function, has been chosen. It is defined as follows:

$$\begin{array}{ll} \text{Minimize} & (f_1(x_1), f_2(\mathbf{x})) \\ \text{Subject to} & f_2(\mathbf{x}) = g(x_2, ..., x_m)h(f_1(x_1), g(x_2, ..., x_m)) \end{array}$$

where:

$$\begin{aligned} m &= 30, \mathbf{x} = (x_1, \cdots, x_m) \in [0,1]^m \\ f_1(x_1) &= x_1 \\ g(x_2, \cdots, x_m) &= 1 + 9 \cdot \sum_{i=2}^{m} \frac{x_i}{m-1} \\ h(f_1, g) &= 1 - (f_1/g)^2 \end{aligned}$$

As additional cores are mostly employed to parallelize fitness evaluation, we can hypothesize that problems with slow fitness functions will benefit more from multi-cores than problems with fast fitness. To investigate the efficiency of Sferes$_{v2}$ with regard to fitness evaluation duration, ZDT2 has been artificially slowed down (via a usleep call) from 0 to 2000 microsecond[10].

Sferes$_{v2}$ is compared in four versions (single thread, two threads, three threads and four threads) to the reference implementation in C provided by K. Deb[11]. Deb's code implements only NSGA-II with binary and real variables; it is not designed to be an abstract framework and it should therefore be faster than a framework.

A population of 200 individuals was employed with a total budget of $100,000$ evaluations. To provide statistically significant results, 30 runs have been launched for each case, on an Intel Quad core Q6600 at 2.40 Ghz.

b) Results: Let first analyze the running time when the fitness evaluation is not slowed down (table I, first column). In the single core case, Sferes$_{v2}$ is slightly slower (19%) than the reference code. However, it should be reminded that the two implementations of NSGA-2 employ different algorithms to sort solutions by dominance. Sferes$_{v2}$ is faster with large populations (because of its better algorithm) while Deb's code is faster with small population (because of faster specialized C code). Hence, despite the abstraction offered by Sferes$_{v2}$, the speed of the two codes is of the same order of magnitude. This validates the use of modern C++ techniques as an alternative to hand-tuned C code. The same experiment

[9]The source code of the benchmarks can be downloaded on the Sferes$_{v2}$ website: `http://www.isir.upmc.fr/~mouret/sferes2`

[10]It should be noted that slowing down with usleep() is an ideal case of slow fitness calculation because there is no memory access during the sleeping time.

[11]`http://www.iitk.ac.in/kangal/codes.shtml`

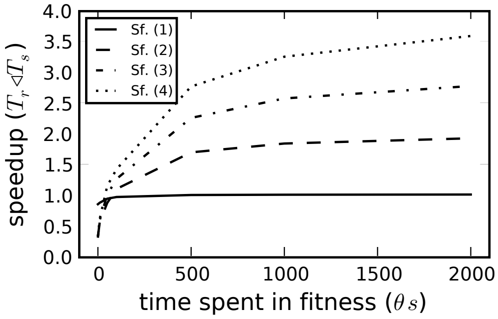


Fig. 6. Speed-up ($\frac{T_s}{T_r}$, where T_s is the time to run the code with $\text{Sferes}_{\text{v2}}$ and T_r the time to run the same optimization with Deb's code) with regard to time spent in fitness and number of cores used. Sf(1) means $\text{Sferes}_{\text{v2}}$ (one core), Sf(2) means $\text{Sferes}_{\text{v2}}$ (2 cores), etc.

has been launched with EO [13] and running time larger than 60 seconds were observed (more than 20 times slower than Deb's code and $\text{Sferes}_{\text{v2}}$). We are currently investigating this difference to double-check that it is not a misuse of EO.

When the number of cores is increased, $\text{Sferes}_{\text{v2}}$ is increasingly slower. This counter-intuitive result means that time spent to parallelize the evaluation (creating threads, splitting data, ...) is larger than time spent to evaluate individuals. ZDT2 only requires a few cycles to be computed, consequently even computing it a few hundred times is easily faster than a few complex system calls.

This picture is reversed with a slower, more realistic, fitness function (table I, columns 2 and 3). In the best case ($2000\mu s$ and 4 cores), $\text{Sferes}_{\text{v2}}$ is 3.6 faster than Deb's code, a speed-up close to the theoritical value (4). When two cores are used, $\text{Sferes}_{\text{v2}}$ is 1.9 times faster than the Deb's code, showing how a now basic dual-core workstation can benefit from a parallel evaluation. To further understand when more than one core should be employed, we computed the speed-up with regard to time spent in fitness evaluation (Fig. 6). Results show that if more than $100\mu s$ are spent in evaluating the fitness function, using more cores is profitable.

B. CMA-ES

c) Experimental setup: CMA-ES [25] is a powerful evolution strategy based on covariance matrix adaptation. It represents a different class of EA than NSGA-2: it is a single objective algorithm, it only handles real-valued parameters, it involves complex matrix manipulations (notably an eigen vector decomposition) at each generation and relies on a small population (less than 10). The small population and the matrix manipulations are expected to decrease the benefits of parallelization.

In this work, CMA-ES has been evaluated on the minimization of $f_{\text{cigar}}(x)$, one of the basic benchmark functions described in [25]:

$$f_{\text{cigar}}(x) = x_1^2 + \sum_{i=2}^{n} (1000 x_i)^2$$

where $x \in [0,1]^{22}$ is the candidate solution to be evaluated. The minimization is stopped when $f_{\text{cigar}}(x) < 10^{-12}$. The

TABLE II

MEAN RUNNING TIME AND STANDARD DEVIATION TO OPTIMIZE f_{CIGAR} WITH CMA-ES, WITH REGARD TO TIME SPENT IN FITNESS EVALUATION.

Implem./fit. slowdown	0 μs (s.d.)	500 μs (s.d.)	2000 μs (s.d.)
Hansen	0.73s (0.1)	9.26s (0.17)	34.61s (0.58)
$\text{Sferes}_{\text{v2}}$ (1 core)	0.96s (0.32)	9.47s (0.2)	33.95s (0.71)
$\text{Sferes}_{\text{v2}}$ (2 cores)	0.99s (0.3)	5.66s (0.13)	18.86s (0.39)
$\text{Sferes}_{\text{v2}}$ (3 cores)	0.94s (0.18)	4.46s (0.19)	13.8s (0.21)
$\text{Sferes}_{\text{v2}}$ (4 cores)	0.97s (0.23)	3.81s (0.09)	11.36s (0.22)

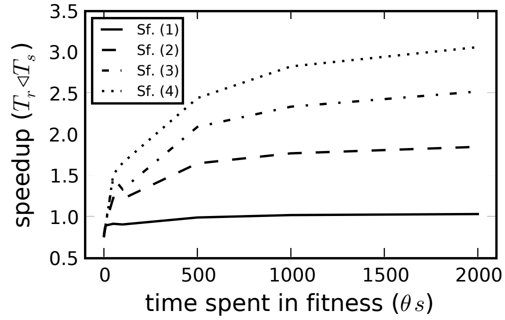


Fig. 7. Speed-up ($\frac{T_s}{T_r}$, where T_s is the time to run the code with $\text{Sferes}_{\text{v2}}$ and T_r the time to run the same optimization with Hansen's code) with regard to time spent in fitness evaluation and number of cores used (Sf(1) means $\text{Sferes}_{\text{v2}}$ (one core), etc.).

parameters are those suggested by Hansen [25]; notably, the population is made of only 7 individuals. $\text{Sferes}_{\text{v2}}$ is compared with Hansen's implementation in C[12].

d) Results: When only one core is used, $\text{Sferes}_{\text{v2}}$ is about 30% slower than Hansen's code (Table II). We think that a 30% slow down is satisfying because $\text{Sferes}_{\text{v2}}$ is more abstract and high-level than the hand-tuned C-code provided by Hansen. The difference in abstraction level is easily illustrated by the length of the two source codes: about 2000 lines of code for Hansen's implementation and 200 lines for $\text{Sferes}_{\text{v2}}$'s implementation. This result validates the use of template-based C++ as an alternative to hand-tuned C code.

Contrary to the experiments with NSGA-2, $\text{Sferes}_{\text{v2}}$ does not significantly slows down when more than one core is employed with a fast fitness. However, adding more cores in this situation appears useless[13]. When time spent in fitness increases (Fig. 7), it becomes more profitable to use several cores. In the best case ($2000\mu s$ spent in fitness), a speed-up of 3.0 is observed with 4 cores and one of 1.8 with two cores.

VI. CONCLUSION

This paper introduced $\text{Sferes}_{\text{v2}}$, a C++ framework for evolutionary computation based on multi-core parallelization and template-based C++. The current implementation and benchmarks results show that $\text{Sferes}_{\text{v2}}$ fulfills the initial goals:

[12] `http://www.lri.fr/~hansen/cmaes_inmatlab.html`

[13] This may be caused by the heuristics of TBB that could consider that the parallelization is inefficient and, consequently, only use one core.

- Sferes$_{v2}$ is only made of $4,000$ lines of code, making it easy to learn;
- when a single core is used, running time are of the same order of magnitude than hand-tuned C code;
- when n cores are used, typical speed-ups range from $0.75n$ (CMA-ES with four cores, slow fitness) to $0.9n$ (NSGA-2 with two cores, slow fitness).

These results validate the use of multi-core programming and template-based C++ to combine efficiency and abstraction in evolutionary computation. However, benchmarks results also revealed that parallelization in Sferes$_{v2}$ was inefficient with very fast fitness functions (evaluation time lower than $100\mu s$). This issue should be further investigated because fast fitness functions are employed by some researchers to test their algorithms.

Sferes$_{v2}$ is currently distributed under the CECILL license[14] (GPL-compatible). It can be downloaded with its documentation on: `http://www.isir.upmc.fr/~mouret/sferes2`

ACKNOWLEDGMENT

The authors thank Sylvain Koos, Jean Liénard, Tony Pinville and Paul Tonelli for their useful feedback.

REFERENCES

[1] E. Mollick, "Establishing Moore's Law," *IEEE Annals of the History of Computing*, vol. 28, no. 3, p. 75, 2006.

[2] D. Geer, "Chip makers turn to multicore processors," *IEEE Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[3] S. Landau, S. Doncieux, A. Drogoul, and J. Meyer, "SFERES : un framework pour la conception de systèmes multi-agents adaptatifs," *Technique et Science Informatiques*, vol. 21, no. 4, pp. 427–446, 2002.

[4] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice Hall PTR, 2000.

[5] K. Driesen and U. Hölzle, "The direct cost of virtual function calls in C++," in *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1996, pp. 306–323.

[6] T. Géraud, Y. Fabre, and A. Duret-Lutz, "Applying generic programming to image processing," in *IASTED International Conference on Applied Informatics*, 2001, pp. 577–581.

[7] A. Alexandrescu, *Modern C++ design: generic programming and design patterns applied*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[8] B. Karlsson, *Beyond the C++ Standard Library*. Addison-Wesley Professional, 2005.

[9] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[10] K. Deb, *Multi-objectives optimization using evolutionnary algorithms*. Wiley, 2001.

[11] C. Gagné, M. Parizeau, and M. Dubreuil, "Distributed BEAGLE: An environment for parallel and distributed evolutionary computations," in *Proceedings of the 17th Annual International Symposium on High Performance Computing Systems and Applications and the OSCAR Symposium*. NRC Research Press, 2003, p. 201.

[12] C. Gagné and M. Parizeau, "Open BEAGLE: A new versatile C++ framework for evolutionary computation," in *Late-Breaking Papers – GECCO'2002*, 2002.

[13] M. Keijzer, J. Merelo, G. Romero, and M. Schoenauer, "Evolving objects: A general purpose evolutionary computation library," in *Artificial evolution: 5th international conference*, vol. LNCS 2310/2002, 2002, pp. 231–244.

[14] A. Liefooghe, M. Basseur, L. Jourdan, and E. G. Talbi, "ParadisEO-MOEO: A framework for evolutionary multi-objective optimization," *Lecture Notes in Computer Science*, vol. 4403, p. 386, 2007.

[15] S. Cahon, N. Melab, and E. Talbi, "ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics," *Journal of Heuristics*, vol. 10, no. 3, pp. 357–380, 2004.

[16] M. G. Arenas, P. Collet, A. E. Eiben, M. Jelasity, J. J. Merelo, B. Paechter, M. Preuß, and M. Schoenauer, "A framework for distributed evolutionary algorithms," in *Proceedings of the PPSN VII conference*. Springer, 2003, pp. 665–675.

[17] E. Alba, F. Almeida, M. Blesa, J. Cabeza, C. Cotta, M. Díaz, I. Dorta, J. Gabarro, C. León, J. Luna, *et al.*, "MALLBA: A library of skeletons for combinatorial optimisation," in *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*. Springer, 2002, pp. 927–932.

[18] D. E. Goldberg and J. Richardson, "Genetic algorithms with sharing for multimodal function optimization," in *Proceedings of the Second International Conference on Genetic Algorithms*, 1987, pp. 148–154.

[19] J.-B. Mouret and S. Doncieux, "Overcoming the bootstrap problem in evolutionary robotics using behavioral diversity," in *Proceedings of IEEE-CEC*, 2009.

[20] J. Lehman and K. Stanley, "Exploiting open-endedness to solve problems through the search for novelty," in *Proceedings of Artificial Life XI*, 2008, pp. 329–336.

[21] K. Deb, S. Agrawal, A. Pratab, and T. Meyarivan, "A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II," in *Proceedings of the PPSN VI Conference*. Paris, France: Springer. Lecture Notes in Computer Science No. 1917, 2000, pp. 849–858.

[22] E. Zitzler, K. Deb, and L. Thiele, "Comparison of Multiobjective Evolutionary Algorithms: Empirical Results," *Evolutionary Computation*, vol. 8, no. 2, pp. 173–195, 2000.

[23] M. T. Jensen, "Reducing the run-time complexity of multiobjective EAs: The NSGA-II and other algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 7, no. 5, pp. 503–515, 2003.

[24] K. Deb, M. Mohan, and S. Mishra, "Evaluating the ε-Domination Based Multi-Objective Evolutionary Algorithm for a Quick Computation of Pareto-Optimal Solutions," *Evolutionary Computation*, vol. 13, no. 4, pp. 501–525, 2005.

[25] N. Hansen, S. Muller, and P. Koumoutsakos, "Reducing the Time Complexity of the Derandomized Evolution Strategy with Covariance Matrix Adaptation (CMA-ES)," *Evolutionary Computation*, vol. 11, no. 1, pp. 1–18, 2003.

[26] C. Igel, N. Hansen, and S. Roth, "Covariance matrix adaptation for multi-objective optimization," *Evolutionary Computation*, vol. 15, no. 1, pp. 1–28, 2007.

[27] K. Deb and R. B. Agrawal, "Simulated binary crossover for continuous search space," *Complex Systems*, vol. 9, no. 2, pp. 115–148, 1995.

[28] J.-B. Mouret and S. Doncieux, "Using behavioral exploration objectives to solve deceptive problems in neuro-evolution," in *Proceedings of GECCO'09*, 2009.

[29] N. Burrus, A. Duret-Lutz, T. Géraud, D. Lesage, and R. Poss, "A static C++ object-oriented programming (scoop) paradigm mixing benefits of traditional oop and generic programming," in *Proceedings of the Workshop on Multiple Paradigm with OO Languages (MPOOL'03)*, 2003.

[30] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, 1999.

[31] L. Dagum and R. Menon, "Open MP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[32] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[33] T. Veldhuizen and M. Jernigan, "Will C++ Be Faster than Fortran?" in *Proceedings of the Scientific Computing in Object-Oriented Parallel Environments*. Springer-Verlag, 1997, p. 56.

[34] J. Coplien, "Curiously recurring template patterns," in *C++ gems*. SIGS Publications, Inc., 1996, p. 144.

[35] M. Austern, "Generic programming and the STL: using and extending the C++ Standard Template Library," *Addison-Wesley Professional Computing Series*, p. 548, 1998.

[36] E. Cantu-Paz, *Efficient and accurate parallel genetic algorithms*. Kluwer Academic Publishers Norwell, MA, USA, 2000.

[14] `http://www.cecill.info/`