libga (MPI)

V 2013.1

Contents

1 **Motivation** 1

2 **Obtaining the Source** 2

3 **Prerequisites** 2

4 **Compiling and Running Examples** 2

5 **Example** 4

6 **Python library notes** 5
- 6.1 Setting up configuration and environment 5
- 6.2 Wrapping the algorithm for use with MPI 6
- 6.3 Running the opzimization process 6
- 6.4 Post-processing 7

7 **C++ library notes** 7

1 Motivation

The libga (MPI) library is actually a set of tools that allow its user to solve optimization problems using genetic algorithm (GA) techniques. Both C++ and Python (>= 2.7) libraries are provided. Only a very limited set of algorithms are supported: for multi-objective problems a variation of the Strength Pareto Evolutionary Algorithm (SPEA2) [ZLT01] has been implemented. The single-objective variant (SGA) does not follow a specific algorithm but it includes the typical steps of a GA, namely evaluation, selection, crossover and mutation. SGAs usually employ an elitism scheme which is also available in libga.
The idea behind this implementation is to offer a relatively simple to use set of tools and

not a comprehensive scientific survey of operators and algorithms found in other libraries. Both the single- and the multi-objective variants (C++ and Python libraries) can be run in a single-threaded environment or as MPI processes. The example scripts and test applications are written for use with MPI. The interfaces of the single-threaded and the MPI versions are almost identical so very little change is needed to go parallel.
The C++ versions expect to deal with real-valued optimization problems only. This is simplifying the GA's implementation and also the communication over MPI. The Python versions are generic enough that the genome type is not important as long as it is array-like. The elements of the genome (i.e. the solution candidates) could be arrays, trees or anything else. However, libga only offers crossover operators for the typical real-valued case. You would have to supply your own custom operators for other genome types. Your custom solution type would have to be able to be "pickled" to be sent via MPI aswell. This is dependent on mpi4py's behavior.

2 Obtaining the Source

The source is hosted on github. Pull with

```
$ mkdir libga_mpi && cd libga_mpi
$ git init
$ git pull git://github.com/mawirth/libga_mpi.git
```

3 Prerequisites

1. Working C++ toolset supporting C++11 (tested with Clang 3.3)
2. CMake (Version >= 2.6)
3. Python (Version >= 2.7)
4. MPICH2
5. mpi4py
6. NumPy
7. SciPy
8. Matplotlib

Table 1: Example scripts

File name	Description
native_build_and_run_kursawe	Builds all C++ test applications in the "build" directory then runs them in an MPI context. The results of each process are merged and displayed in a Matplotlib graph. You can click data points to display more information in the terminal.
native_build_and_run_rastrigin	Builds all C++ test applications in the "build" directory. It runs the rastrigin problem in an MPI context. Results are merged and the fittest (i.e. best fitting) solution is displayed in the terminal. For the rastrigin problem the fitness should be close to zero. The variables themselves are also approaching zero for a global minimum.
native_build_and_run_MOP5	Same as the above but runs the MOP5 (Viennet) problem. Displays the three-dimensional objective space using Matplotlib.
python_run_kursawe	Run the kursawe test problem (Python implementation) in an MPI context. Merges results and displays them in a Matplotlib graph.
python_run_rastrigin	Run the rastrigin test problem (Python implementation) in an MPI context. Merges results and prints the fittest solution candidate to the terminal
python_run_MOP5	Run the MOP5 (Viennet) problem and displays the three-dimensional objective space.

4 Compiling and Running Examples

The "examples" directory in the source distribution includes a number of shell scripts that both build and run a few test problems.

If you do not want to run everything at once execute this in the source directory:

```
$  mkdir build && cd build
$  cmake .. && make
```

This produces three executables "kursawe", "mop5" and "rastrigin".

5 Example

We try to find optimal values for the Kursawe[Kur91] test problem: minimize both $f_1(x)$ and $f_2(x)$ where

$$f_1(x) = \sum_{i=1}^{n-1} \left[-10\ exp \left(-0.2 \sqrt{x_i^2 + x_{i+1}^2} \right) \right] \tag{1}$$

$$f_2(x) = \sum_{i=1}^{n} \left[|x_i|^{0.8} + 5\ sin\left(x_i^3\right) \right] \tag{2}$$

n is the length of the solution vector (typically 5) and $x \in [-5, +5]$.
To execute the kursawe test problem in an mpi context use

```
$  mpiexec -n 8 ./kursawe
```

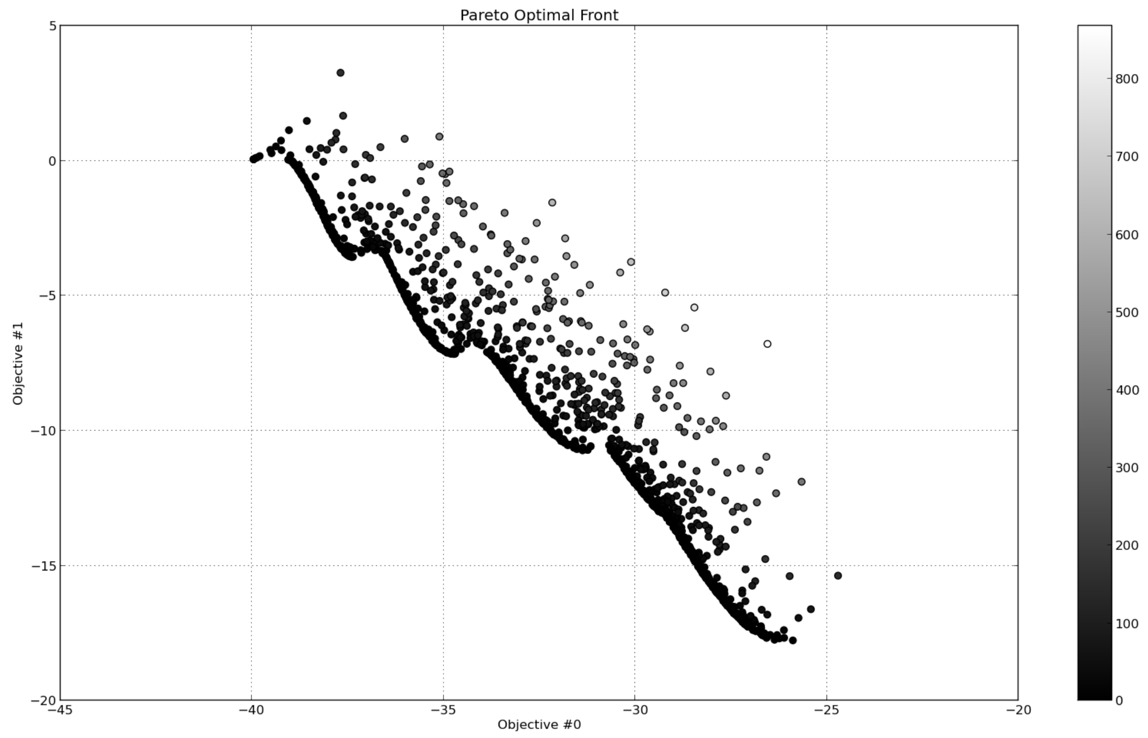
This will write eight files called “kursawe_data_0”, ...,”kursawe_data_7”. Each process has written its own set of results. A couple of Python scripts are provided which can be used to post-process the results, effectively merging them and recalculating fitness information.

```
$  python ../py/merge_to_npz.py ‘‘kursawe_data_*’’ 2 test.npz
```

This combines all information in the numpy binary file “test.npz”.
Note that “kursawe_data_*” is a wildcard expression. Any valid bash expression for filtering file names is accepted here. The argument “2” tells the merge script that the objective space of the kursawe test problem is two-dimensional. When in doubt, the scripts all display a short example usage string when invoked with the wrong (or zero) number of arguments.
You can then proceed and display all of this data using Matplotlib:

```
$  python ../py/paretoplot.py test.npz
```

This produces a graph very similar to this image:



You can click each data point and view the solution's variables, objective values and fitness in the terminal. If too many data points are displayed and you want to reduce the amount of points to just the pareto optimal front you can pre-filter the plot's input data:

```
$ python ../py/merge_to_npz.py ''kursawe_data_*'' 2 test.npz
$ python ../py/filter.py test.npz test.npz
$ python ../py/paretoplot.py test.npz
```

6 Python library notes

libga is designed in a way that supports a certain workflow for working on new optimization problems. If you are tackling a new problem oftentimes writing it down in Python is a good way to start exploring things. In the following section you will find a guide for the most important steps for running the opzimizaion algorithm in Python. We will use the Kursawe test function again.

6.1 Setting up configuration and environment

First, you have to define a few settings for running the optimization algorithm:

```
population  = 100    # number of solution candidates
                     # (including archive)
variables   = 5      # variables per solution
generations = 200    # algorithm should terminate after n generations
```

We define a lambda function that can be used to evaluate our objectives. Because both the SGA and the SPEA2 algorithms are very generic, they only work with lists i.e. the genome data (the set of all candidate solutions) is just a list of lists. In our case - the Kursawe test problem - we want a list of floating-point vectors. None the less we can convert this list of vectors into a numpy matrix. See ”test_problems.py” for an example implementation of the Kursawe problem.

```
kursawe_fn = lambda g: test_problems.test_kursawe(np.asarray(g))
```

Both SGA and SPEA2, at least their Python versions, need an initial genome. We generate a random start:

```
genome = constraints[:,0]+(constraints[:,1]-constraints[:,0])
       * np.random.random(variables) for i in range(population)]
# assert len(genome) == population
```

Note, that we cannot create a Numpy matrix here, since SPEA2 and SGA both need a list. We do not want to make a lot of assumptions about the genome to keep the algorithm as generic as possible. This also allows for different kinds of genome types like, for instance, sets of trees.
We pass this genome data to an Spea2 instance:

```
alg = spea2.Spea2(genome)
```

You could now run this algorithm instance by calling alg.start(). This executes in a single-threaded context and might be the ideal way to get started for testing purposes. Since libga is built in two layers we can extend the capabilities of our test application by composition.

6.2 Wrapping the algorithm for use with MPI

```
im = island_model.IslandModel(alg)
```

This wraps our Spea2 instance in an IslandModel instance which knows how to communicate over MPI and handles migration of solution candidates from one node to the next. The interface of the most important function “start()” is the same for both Spea2 and the IslandModel class which makes going from single-threaded to parallel very easy.

6.3 Running the opzimization process

```
success = im.start(
    kursawe_fn   # objective function
      # run until we have reached our prefedined maximum of generations
    , lambda : alg.generation < generations
      # crossover method (see gacommon.py for implementation)
    , gacommon.blxa_crossover
      # check whether a solution candidate is valid for our purposes
      # this can also unclude more sophisticated tests, not just
      # range checking.
    , lambda s: gacommon.is_within_constraints(s, constraints)
      # mutation operator
    , lambda s: gacommon.mutate_uniform(s, constraints)
)
```

"success" is true if everything went fine and the algorithm ran until we asked it to terminate (second argument to the start() function). If start() returned false the algorithm stalled at some point which means that no more valid offspring solutions could be found. Please check the constraints or increase the population size in this case (of course, other factors play into that problem aswell).

6.4 Post-processing

The suggested way of saving the results of the optimization process is in the form of Numpy .npz files:

```
np.savez(''kursawe_data_'' + str(MPI.COMM_WORLD.rank) + ''.npz''
    , genome = im.island.genome, ospace = im.island.ospace
    , fitnesses = im.island.fitnesses)
```

Note that we "reach through" the island model instance to get the spea2 instance which actually has the genome data here. We run this python script using mpiexec in n processes to produce n .npz files with result data that have to be merged for further examination.

```
$ mpiexec -n 8 python ./kursawe.py
$ ls *.npz
$ python merge_as_npz.py ''kursawe_data_*'' 2 test.npz
```

Display results using Matplotlib:

```
$ python paretoplot.py test.npz
```

Note that for three-dimensional objective spaces (see the Viennet MOP5 test problem) a script called "paretoplot3d.py" is provided.

7 C++ library notes

When you are comfortable enough with the Python implementation of your test problem, you might think about porting your algorithms to C++ in order to gain speed. The steps to use the C++ version of libga are identical to the steps needed for the Python version. Since the concepts are identical and a well-documented example "src/kursawe.cpp" is provided only the main start() call will be described here.
There are many ways to call the island model's start() method. We will choose a modern lambda-based approach:

```
// create the spea2 instance. min_constraints and max_constraints
// are vectors with constraints information for each variable
// of a candidate soltion
libga_mpi::spea2< float > spea2(population, archive
   , min_constraints, max_constraints, 2);
```

```
// wrap the spea2 instance. just as in the Python version, the island
// model knows how to communicate via MPI.
// migration_prob is the propability of a solution to migrate from
// one algorithm instance (``island'') to the next.
libga_mpi::island_model< decltype(spea2) > im(spea2, migration_prob);
```

```
// run the algorithm. this is the exact same function signature as
// a call to spea2.start().
const bool success = im.start(
   kursawe     // test function
     // crossover operator
   , libga_mpi::xover::blxa< float >()
     // run until the current generation hits ``generations''
   , [generations](std::size_t g) { return g == generations; }
     // check if a solution is valid. in practical problems this is
     // usually a lot more elaborate
   , [&](const float *x, std::size_t n)
     { return is_within_constraints(x, n, min_constraints,
        max_constraints); }
     // mutation operator
   , libga_mpi::mutation::mutate_none< float >()
     // if no new offspring can be generated after n attempts
     // the start() method returns false
   , max_crossover_attempts);
```

```
// save genome data to a text file for further processing with e.g.
// scripts like ``py/merge_to_npz.py''
std::ofstream out(``output'');
out << spea2;
```

References

[Kur91] Frank Kursawe. A Variant of Evolution Strategies for Vector Optimization. In H.-P. Schwefel and R. Männer, editors, *Parallel Problem Solving from Nature — Proc. 1st Workshop PPSN*, volume 496 of *Lecture Notes in Computer Science*, pages 193–197, Berlin, 1991. Springer.

[ZLT01] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm. Technical report, 2001.