Parallel Particle Swarm Optimization Library in C++11*

Soumyarup Sarkar

May 10, 2013

1 Background and Introduction

1.1 Intuition

Particle swarm optimization (herein denoted "PSO") is an algorithm that optimizes a function by moving around a population of candidate solutions (a "swarm" of "particles") in a search space[3]. Each particle evaluates the objective function at its current location, and determines its movement by taking into account its own current location, its best location, and those of the swarm, with some random perturbations.

Swarm intelligence has numerous applications; for example, if we have a set of robots exploring a surface, and we would like to find the point with minimum altitude, we could perform what is essentially particle swarm optimization using the robots as the particles and their altitudes as "function evaluations."

Standard PSO does not compute gradients of the objective function, and makes effectively no assumptions about it, but it does not guarantee convergence to an optimal solution. There are numerous variants of PSO, and some (such as [2]– which introduces "stochastic PSO") attempt to address this issue by guaranteeing global convergence with probability 1.

1.2 Parallel PSO

[1] introduced a *parallel* particle swarm optimization algorithm. The key idea for parallelization was to set up multiple groups of particles, and process each group independently, on a different processing unit.

More notably, they presented communication strategies between groups that would make it harder for whole groups of particles to get stuck at at local minima, facilitating convergence to a global minimum for the swarm (though they do not guarantee convergence).

Thus, we get both independent groups of particles we can run in parallel, and communication between them that we allow every few iterations, with the expectation that groups won't get stuck at local minima since each particle takes into account its own best position and its group's best position, and is affected by the entire swarm's best position (due to the communication strategy).

*MATP 4820 final project.

2 Our Implementation

We present a limited implementation of the algorithm presented in [1] in `C++11`. Our implementation is parallel and has groups for which iterations occur independently, but we have not implemented any of the communication strategies presented in the paper.

2.1 Minutiae and Tuning

There are several parameters to tune in our implementation, and for many of them, we assign them to the values we use only for empirical reasons.

2.1.1 Initialization

We initialize a fixed number of groups, and then we initialize a fixed number of particles for each group. The number of groups should be greater than or equal to the number of processing units, since each group runs on its own thread. 20—40 particles per group seems to work well.

Initial positions and velocities are selected randomly from a uniform distribution, provided minimum and maximum parameters.

2.1.2 Formulas for Movement

The velocity, position, and best known position for each particle, and the group's best known value, are updated in parallel before the threads are synchronized in order to update the swarm's best known value before the next iteration.

We use the same formulas as in [1] to determine velocity and position. Note that a superscript denotes the iteration number and i and j denote the index of a particle in a group and the index of the group itself, respectively. t is used to denote an iteration number.

$$
\begin{aligned}
V_{i,j}^{t+1} &= W^t \cdot V_{i,j}^t + C_1 \cdot r_1 \cdot (P_{i,j}^t - X_{i,j}^t) + C_2 \cdot r_2 \cdot (G_j^t - X_{i,j}^t) \\
X_{i,j}^{t+1} &= X_{i,j}^t + V_{i,j}^{t+1} \\
f(G^t) &\leq f(G_j^t)
\end{aligned}
$$

Here V denotes the velocity and X denotes the position of a particle. f is the objective function.

P is a particle's personal best, G_j is the j-th group's best position, and G is the swarm's best position.

W is a weight ranging from 0 to 1, which is assigned for each iteration beforehand. It is generally decreasing because particles in the swarm occupy less random positions as the iteration number increases. A uniform distribution from 0.6 to 0.4 works well.

C_1 is a constant for the particle's personal influence on its movement, and C_2 is a constant for the group's influence on the particle's movement. 2—4 for both seems to work well.

r_1 and r_2 are random variables ranging from 0 to 1.

We also include a v_{max} constant that is the maximum we allow the velocity to reach, to prevent a particle from moving too fast and potentially skipping over a better local minumum than the one it is being pulled toward.

2.1.3 Termination Criteria

The algorithm repeatedly evaluates the above formulas using the procedure described at the beginning of the previous section. It terminates when a maximum number of iterations is reached.

2.1.4 Miscellaneous Technical Details

We are using CMake for build automation, and `C++11` for implementation; to get the best performance, we are invoking the maximum optimization setting in the compiler we use (`g++ -O3`), and we are using Google's `tcmalloc`, which offers better performance than the standard `malloc` (which is abstracted away in our case) for multithreaded programs such as ours.

Also, we make heavy use of the `C++` standard library; simple profiling reveals that a lot of time is spent simply generating r_1 and r_2.

2.2 Implementation Details

This is the constructor for our implementation that takes, as part of its input, the aforementioned tuning parameters.

```
Particle_Swarm( \
                // the objective function
                const std::function<double(const std::vector<double> &)> \
                    obj_func,

                // the number of dimensions
                const unsigned int n_dimensions,

                // vectors of minimum and maximum initial positions
                const std::vector<double> &min_init,
                const std::vector<double> &max_init,

                // number of groups, number of particles
                const unsigned int num_groups,
                const unsigned int num_particles,

                // maximum velocity
                const double max_velocity,

                // weights[t] = weight on previous velocity at
                // iteration t
                const std::vector<double> &weights,

                // personal influence
                const unsigned int constant_1,

                // group influence
                const unsigned int constant_2,

                // **NOT IMPLEMENTED**
                const unsigned int loose_corr_strategy,
                const unsigned int strong_corr_strategy,
```

```
        // maximum number of iterations
        const unsigned int max_iter);
```

Calling this constructor initializes the swarm of particles; the `run()` method executes the algorithm, and returns the best particle position x^*, and the function evaluated at that position $f(x^*)$.

3 Sample Usage and Test Cases

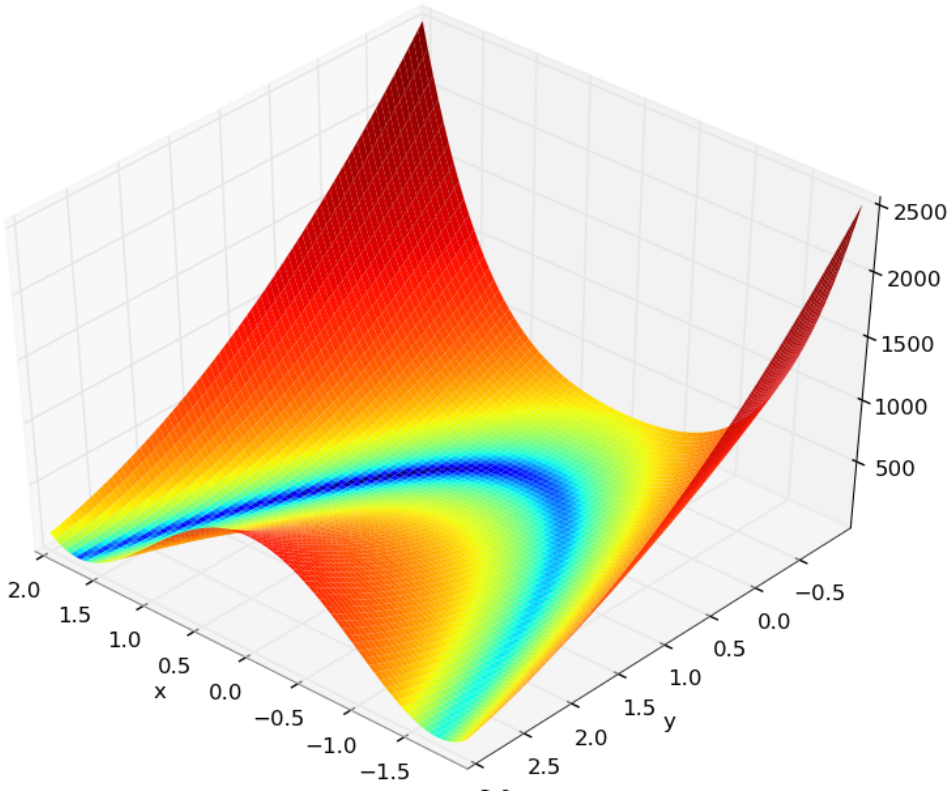
See `demo.cpp` in the source code to see sample usage.

We used non-convex test cases because PSO is best for global optimization problems; convex programs are more efficiently solved by other methods.

3.1 Rosenbrock Function

Our first test case is the Rosenbrock function, that [1] also used. It is non-convex, and has a long valley that contains a global minimum.

$$f(x) = \sum_{i=1}^{n-1} \left(100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right)$$



Rosenbrock Function with 2 Variables. Plotted by Rachelle Villalon for *The Nature of Mathematical Modeling*.

The optimal point $x^\star$ has 1 in each dimension.

This is the code we used to optimize the Rosenbrock function in 16 dimensions using our library, which makes clear what values we used for each parameter:

```
#include <ppso/particle_swarm.hpp>

#include <cmath>
#include <iostream>

class Rosenbrock_function {
public:
  double operator()(const std::vector<double> &x) {
    double ret = 0;
    for (int i = 0; i < x.size() - 1; ++i) {
      ret += std::pow(x[i] - 1.0, 2) + \
        (100.0 * std::pow(x[i + 1] - (x[i] * x[i]), 2));
    }
    return ret;
  }
};

int main(int argc, char *argv[])
{
  std::vector<double> min_p(16, -4.0);
  std::vector<double> max_p(16, 4.0);
  // each of 20000 iterations have weight 0.6
  std::vector<double> wts(20000, 0.6);

  // perform ppso
  auto ret = ppso::Particle_Swarm{
    Rosenbrock_function(),
    16, // number of dimensions
    min_p, // vector of minimum values for each dimension
    max_p, // vector of maximum values for each dimension
    std::thread::hardware_concurrency(), // number of groups (threads)
    40, // number of particles
    0.004, // maximum velocity
    wts, // vector of weights for each iteration
    2, // personal influence constant
    2, // group influence constant

    // NOT IMPLEMENTED:
    4, // every 4 iterations, communication strategy 1
    8, // every 8 iterations, communication strategy 2

    20000 // 20000 iterations
  }.run();

  // print x*
  std::cout << std::fixed << "x* = [ ";
  for (double x: ret.first) {
    std::cout << x << " ";
  }
  std::cout << "]'\n";

  // print f(x*)
```

```
  std::cout << "f(x*) = " << ret.second << std::endl;

  return 0;
}
```

We used 0.6 as the weight at each iteration just for simplicity.

The following is output (abridged for formatting) from running an instance of the above on a laptop computer with an Intel Core i5 processor 430M (which has 4 cores, so we had 4 groups of particles):

```
x* = [ 1.000091 1.000712 1.000050 1.000203
 1.000049 1.000689 0.999813 0.999185 0.998823 0.996556
 0.994044 0.988990 0.981622 0.965176 0.932339 0.869256 ]‘

f(x*) = 0.008684

real    0m5.306s
user    0m5.983s
sys     0m2.973s
```

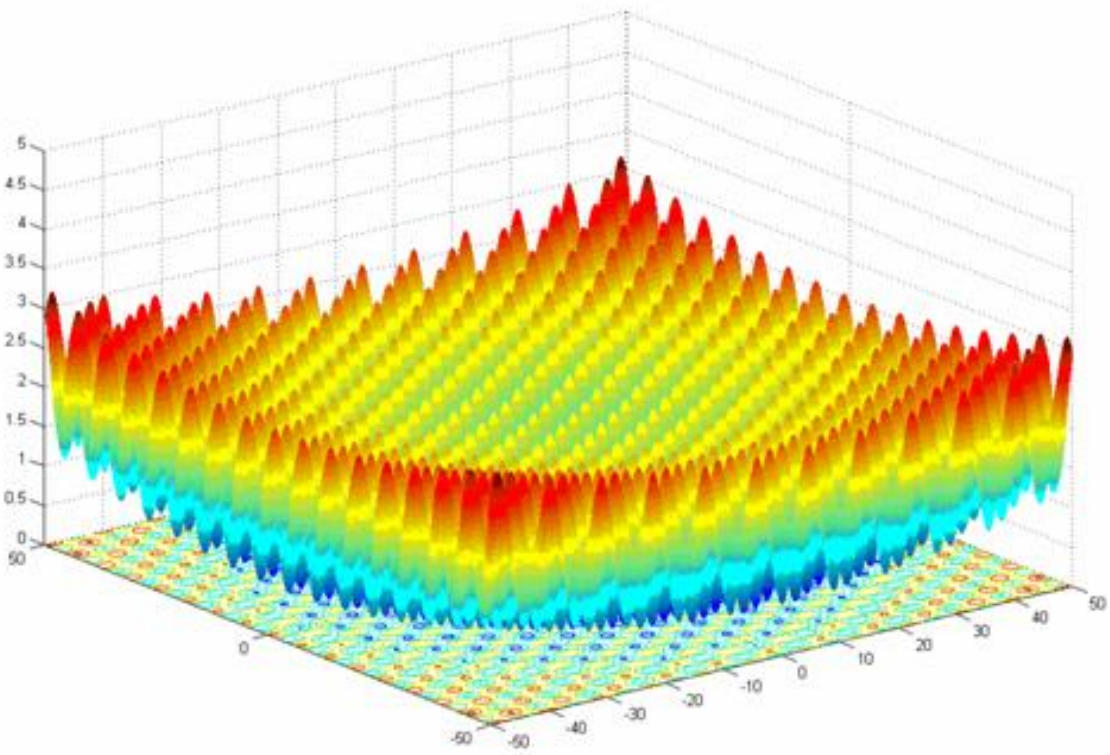
Since the optimal point would have 1 in every dimension, the algorithm performed pretty well. Note that the maximum velocity and the constants are particularly important here, because since groups do not communicate with each other, we need at least 1 group to have a point that is close to the optimal point and not get trapped somewhere else in the valley where the solution lies.

If v_{max} is too high, particles will be too susceptable to the influence of their own best positions and the group's previous best positions to keep randomly when more random movement may give a better position. If it were too low, particles would not move far enough even when they'd be moving toward a better position. A simlar argument is made for C_1 and C_2.

3.2 Griewank Function

Our second test case is the generalized Griewank function used in [1],

$$f(x) = \frac{1}{400}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$



Griewank Function with 2 Variables. Plotted by Abdel-Rahman Hedar for Test Functions for Unconstrained Global Optimization.

It is non-convex, and the optimal $x^\star$ has 0 in each dimension.

We used the same parameters as in the last test case (as well as the same number of dimensions); the following is the code for the function itself:

```
class Griewank_function {
public:
  double operator()(const std::vector<double> &x) {
    double ret = 0.0;
    for (double x_i: x) {
      ret += x_i * x_i;
    }
    ret /= 400.0;
    ret += 1;

    double cos_term = 1.0;
    for (size_t i = 0; i < x.size(); ++i) {
      cos_term *= std::cos(x[i] / std::sqrt(i + 1));
    }

    ret -= cos_term;

    return ret;
  }
};
```

This is the output from running an instance of our algorithm on the problem (again, abridged for formatting):

```
x* = [ 0.000005 0.000357 0.000048 -0.000710
       -0.000320 -0.000453 -0.000081 0.001442 0.000039
       0.000320 0.000676 -0.000055 -0.000091 -0.000139
       -0.000714 0.000014 ]‘

f(x*) = 0.000000

real    0m13.584s
user    0m26.783s
sys     0m3.270s
```

This is reasonably close to the optimal point, so the algorithm performed pretty well. It's slower than the previous example because function calls take much longer due to having to take the cosine and square root so often, and also because there are two loops to iterate through each dimension.

Note that the more we expand the domain of the original particles' initial positions, the more likely the "best" group is to converge to a non-optimal point.

Here is the output from an instance of the program with each dimension of the initial population being selected from a uniform distribution between -10 and 10 and nothing else changed:

```
x* = [ -0.000047 -0.000048 0.000082 -5.349114 -0.000242 -0.000276
       -0.000312 0.000319 0.000013 0.000124 -7.021363 -0.000292
       0.000012 0.000870 0.000261 -0.000043 ]‘

f(x*) = 0.730948

real    0m14.609s
user    0m31.077s
sys     0m3.643s
```

As you can see, x_4 and x_{11} are far from their optimal values.

The solution to this would be to use more particles (or more groups and the same number of particles), or (better yet) to implement communication strategies.

4 Points Outside of Domain, Maximization, and Constraints

Note that while we do not include explicit support for function evaluations that do not exist (such as $\log(-10)$), a user-provided fuction could return the maximum value a `double` type could take at points that do not exist.

Similarly, for maximization, the user could simply write a function that negates the original objective function and pass that in as the objective function.

Also, while we do not provide built-in support for constraints, there are several ways to handle them; we could treat infeasible points as the maximum `double` value as described earlier, or we would use a penalty method or something similar.

5 Limitations

Without a communication strategy, our implementation simply behaves like multiple standard PSO runs happening simultaneously, and choosing the best value of x ever encountered by a particle upon completion.

This means that the algorithm is slow to converge, and each group could get trapped at local minima that are far from the global minimum. Implementing communication strategies is the most straightforward way to improve our implementation.

Apart from that, performance of our implementation will be improved as the `C++11` thread library is better optimized and starts to use thread pools (if it doesn't already), since they are not explicitly available using the standard library's abstraction; this potential improvement owes to the need for synchronizing so often, which would be necessary if we were to implement communication strategies.

We will not continue to work on this project because alternatives that are available (such as PSwarm[4], which is parallelized using MPI) are better engineered and handle errors better; our implementation was just an academic toy.

Appendix: Source Code

The code is available at `http://rpi.edu/~sarkas4/MATP4820FinalCode_ppso.zip`.

References

[1] Jui-Fang CHANG, Shu-Chuan CHU, John F RODDICK, and Jeng-Shyang PAN. A parallel particle swarm optimization algorithm with communication strategies. *Journal of information science and engineering*, 21(4):809–818, 2005.

[2] Zhihua Cui and Jianchao Zeng. A guaranteed global convergence particle swarm optimizer. In *Rough Sets and Current Trends in Computing*, pages 762–767. Springer, 2004.

[3] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm intelligence*, 1(1):33–57, 2007.

[4] A Ismael F Vaz and LN Vicente. Pswarm: A hybrid solver for linearly constrained global derivative-free optimization. *Optimization Methods & Software*, 24(4-5):669–685, 2009.