Derivative-free minimization

18/12/2014

1 Derivative-free minimization

In some application it is required to minimize a function whose derivative is not known or difficult to compute. In such cases a derivative-free minimization method can be employed. In this exercise we consider a very basic method for the minimization of a function $f(x, y)$. This method, starts from an initial candidate (x_0, y_0) and at each step, tries to approach the function's minimum by evaluating the function on p points uniformly distributed on a circle of radius s around the current candidate and by selecting as a new candidate the one, among the p, where the evaluation is smallest. s can be seen as a step length on which depends the speed and accuracy of the method: a small value for s will lead to a slower convergence but a more accurate solution. This is shown if Figure 1.

The method i stopped when no further improvement can be achieved, that is, when $f(x_k, y_k) = f(x_{k+1}, y_{k+1})$. In our exercise we will assume that the function has only one global minimum:

$$(x_{min}, y_{min}) = (0.6971, 0.1317), \quad f(x_{min}, y_{min}) = 1.5670$$

The exercise is about parallelizing the method described above

2 Package content

In the `derivative_free` directory you will find the following files:

- `main.c`: this file contains the main program which first calls a sequential routine `sequential_minimization` which computes the minimum of a (unknown) function using the method described above. The main program then calls the parallel routine `parallel_minimization`. The objective of this exercise is to modify this routine in order to parallelize the minimization method. **Only this file has to be modified for this exercise**.
- `aux.c, aux.h`: these two files contain auxiliary routines and **must not be modified**.

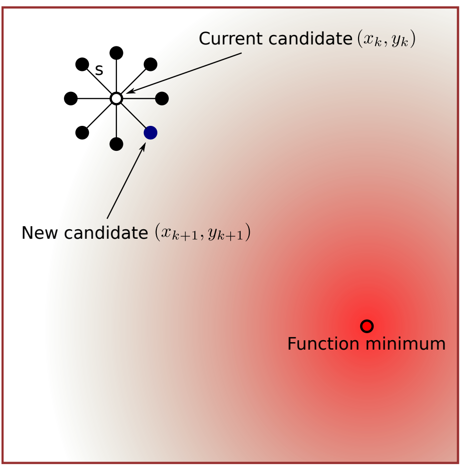

Figure 1: Basic functioning of the derivative-feee minimization

The code can be compiled with the `make` command: just type `make` inside the `derivative_free` directory; this will generate a `main` program that can be run like this:

```
$ ./main p s
```

where `p` is the number of points around the current minimum where the function has to be evaluated and `s` is the step size.

3 Assignment

- At the beginning, the `parallel_minimization` routine is a copy of `sequential_minimization`. Modify this routine in order to parallelize the function minimization method described above. Note that all the p function evaluations performed at each step are independent and can thus be performed concurrently. This is the source of parallelism that has to be exploited to achieve the parallelization. Note that, in principle, it is possible to simply parallelize the loop over the evaluations and protect with a critical section the update of the global minimum. This, however, can result in very poor performance especially if the value of p is very high. For this exercise, instead, a reduction approach will have to be used, where the reduction operation has to be performed by hand (because no equivalent is available in OpenMP).
- ✎ Report the execution times for the two implemented parallel versions with 1, 2 and 4 threads. Analyze and comment on your results.

  Is it possible that the parallel and sequential codes follow a different path? Why?

Report your answer in the `responses.txt` file.

Advice

- If multiple evaluations are computed concurrently by several threads, each thread will only know the minimum among the subset of points on which it evaluated the function. These local minima can be stored (along with the corresponding coordinates) in the `xyz` array. A method has to be developed to identify the global minimum among the `nth` local minima where `nth` is the number of threads participating in the computations. After the evaluations are concurrently executed, one thread computes the global minimum and communicates it to all the other threads along with the corresponding (x, y) coordinates.
- Choose relatively small values for p (say, 8 or 16) and relatively big values for s (say, 0.1) when implementing and validating your parallel version. Choose, instead, relatively large values for p (for example 128 or 256) and relatively small values for s (for example 0.0001 or 0.00001) when evaluating performance.