Article

Not peer-reviewed version

How to Parallelize “Non-Parallelizable” Minimization Functions

Dmitry Lukyanenko * , Sergei Torbin , Valentin Shinkarev

Posted Date: 14 November 2024

doi: 10.20944/preprints202406.0571.v4

Keywords: minimization; parallel algorithm; parallelization; MPI

Article

How to Parallelize "Non-Parallelizable" Minimization Functions

Dmitry Lukyanenko *, Sergei Torbin and Valentin Shinkarev

Department of Mathematics, Faculty of Physics, Lomonosov Moscow State University, Moscow 119991, Russia

* Correspondence: lukyanenko@physics.msu.ru (D.L.)

Abstract: The paper proposes an algorithm for parallelizing calculations that arise when using highly optimized minimization functions available in many computing packages. The main idea of the proposed algorithm is based on the fact that although the "inner workings" of the minimization function used may not be known to the user, it inevitably uses in its work auxiliary functions that implement the calculation of the minimized functional and its gradient, which are usually realized by the user and in many cases can be parallelized. The paper discusses in detail both the parallelization algorithm and its software implementation using MPI parallel programming technology, which can act as template for parallelizing a wide set of applied minimization problems. An example of software implementation of the proposed algorithm is demonstrated using the Python programming language, but can be easily rewritten using the C/C++/Fortran programming languages.

Keywords: minimization; parallel algorithm; parallelization; MPI

MSC: 65Y05, 68W10, 65-04

---

1. Introduction

When solving many applied problems, there is often a need to minimize certain functional. Recently, a variety of minimization software packages have become available to scientists, providing access to highly-optimized minimization functions. For example, in one of the most popular libraries among the scientific community, SciPy [1] of the Python programming language, multiparameter minimization functions are available, which can be divided into two classes: local minimization functions and global minimization functions. Local minimization functions include those that implement: the Nelder-Mead algorithm [2], the modified Powell algorithm [3], the conjugate gradient algorithm [4], the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm [5–8], the Newton conjugate gradient algorithm [9], the limited memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS-B) algorithm with box constraints [10], a truncated Newton (TNC) algorithm [11], the Constrained Optimization BY Linear Approximation (COBYLA) algorithm [12], Sequential Least Squares Programming (SLSQP) [13], a trust-region algorithm for constrained optimization [14,15], the dog-leg trust-region algorithm [16], the Newton conjugate gradient trust-region algorithm [17], the Newton Generalized Lanczos Trust Region (GLTR) algorithm [18,19], a nearly exact trust-region algorithm [20]. Global minimization functions include those that implement: the Basin-hopping algorithm [21], the "brute force" method [22], the differential evolution method [23], simplicial homology global optimization [24], dual annealing optimization [25], the DIRECT (Dividing RECTangles) algorithm [26,27].

The availability of such minimization software packages has significantly increased the efficiency of scientific work, since it has allowed scientific groups that do not specialize in numerical minimization methods not to waste time on software implementation of the minimization algorithms necessary in scientific work. However, it should be noted that the most popular minimization packages involve sequential calculations. This leads to the following problem: the computational complexity of many modern application problems requires such large amounts of calculations that these calculations cannot be performed on personal computers in a reasonable time.

This problem could be solved as follows. With the development of computing capabilities, parallel computing and parallel programming technologies have become widespread, allowing computationally complex problems to be solved in a reasonable time. The problem of long calculations is solved by parallelizing the calculations between different computing nodes of a computing system. In particular, such an approach to minimizing functionals is used in the PETSc (Portable, Extensible Toolkit for Scientific Computation) package [28], an implementation of which also exists for Python in the form of the petsc4py package [29]. However, this package contains a limited set of minimization functions (see "Tao solvers"), which may not contain the function needed by a particular scientist and solving a fairly specific problem. And packages that contain the necessary minimization function may contain only its sequential version. Thus, many scientists come to the following dilemma: either agree to an extremely long calculation, or try to implement minimization algorithms independently and then parallelize them. In both cases, the time required to solve the scientific problem can increase significantly.

It should be noted that many minimization software packages have recently begun to add parallel computing to their implementation on computer systems with shared memory. In practice, such systems are usually a computer with a single multi-core processor. As a result, within one computing node (for example, a personal computer), the running time of the algorithm can be reduced, but the maximum possible acceleration of the program is still limited by the capabilities of this computing node. Therefore, it is still relevant to develop software implementations that have the ability to use computing systems with distributed memory.

In connection with the above, a question arose: is it possible to parallelize the calculations that arise when using standard minimization functions, bypassing a detailed study of the computational algorithms that are implemented inside these functions? That is, is it possible to use the available minimization functions "as is"? Research has shown that this can be done. The structure of any minimization function is such (see Figure 1) that the main calculations involve calculating the functional and, possibly, its gradient (in the case of using first-order minimization methods). It is the functions that implement the calculations of the functional/gradient that are implemented by scientists in any case independently and can often be parallelized. If this is possible, it is possible to use the available minimization functions "as is". This work is devoted to how this can be done.

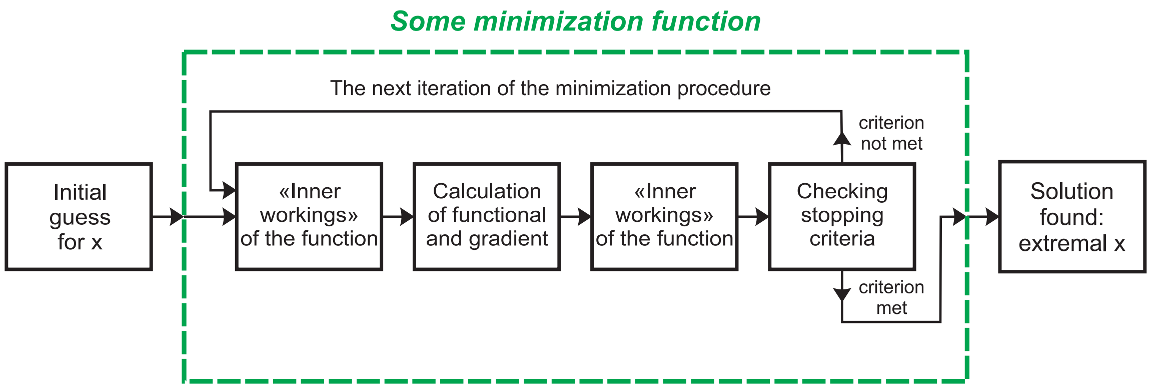
Some minimization function
The next iteration of the minimization procedure
criterion not met
Initial guess for x
«Inner workings» of the function
Calculation of functional and gradient
«Inner workings» of the function
Checking stopping criteria
criterion met
Solution found: extremal x

Figure 1. Typical structure of minimization functions.

The structure of this work is as follows. Section 2 describes the structure of the proposed parallel algorithm, formalizes the pseudo-code of this algorithm, and also describes the structure of the corresponding Python code that implements this algorithm using the mpi4py package. The mpi4py package for organizing the communication of various computing processes allows the use of MPI message passing technology, which is currently the main programming tool in parallel computing on systems with distributed memory. For convenience, the Python code is presented in the form of pseudocode that does not contain operations and function arguments that are unimportant for the perception of the algorithm. Section 3 describes the full version of the software implementation of the parallel algorithm, which also uses the latest version of the MPI standard — MPI-4 [30,31]. Section 4 presents the results of test calculations demonstrating the strong scalability properties of the proposed software implementation. Section 5 discusses possible ways to modify the proposed algorithm, taking into account the most frequently encountered situations in practice.

2. Parallel Algorithm

According to Figure 1, any minimization function, when executed, repeatedly calls functions that are specified by the user and implement the calculation of the functional and its gradient. Given that the minimization function used is sequential, it will only be called by the master process. Therefore, functions that are specified by the user and implement the calculation of the functional and its gradient will be split into two pairs of functions: 1) functions that are called by the master process only within the minimization function, and 2) auxiliary functions that are called by all processes.

Functions that are called only by the master process must first implement the distribution of a “flag” to all other processes, based on which the remaining processes can conclude what needs to be done next: 1) calculate their part of the functional, or 2) calculate their part of the gradient. The data needed for calculations is then distributed across all processes. After this, the master process launches an auxiliary function, through which part of the functional or part of its gradient is calculated, for which the master process is responsible for calculating. Then, data from intermediate calculations performed in parallel is collected from all processes, and the final result of calculating the functional and its gradient is aggregated.

After the minimization function completes its work on the master process, the master process sends a “flag” to all other processes, based on the value of which the other processes can conclude that it is necessary to stop working.

The other processes run an endless “while” loop, inside which each process waits to receive a “flag”, based on the value of which it concludes what needs to be done next: 1) calculate its part of the functional, 2) calculate its part of the gradient, 3) finish the job. Further, in the case of the first two options, data for calculations is expected and after receiving it, an auxiliary function is launched, through which the calculation of its part of the functional or its gradient is implemented. Next, the results obtained are sent to the master process.

This algorithm is illustrated in Figure 2 and formalized in the form of the following pseudocode (see algorithm 1), in which the master process has `rank = 0`.

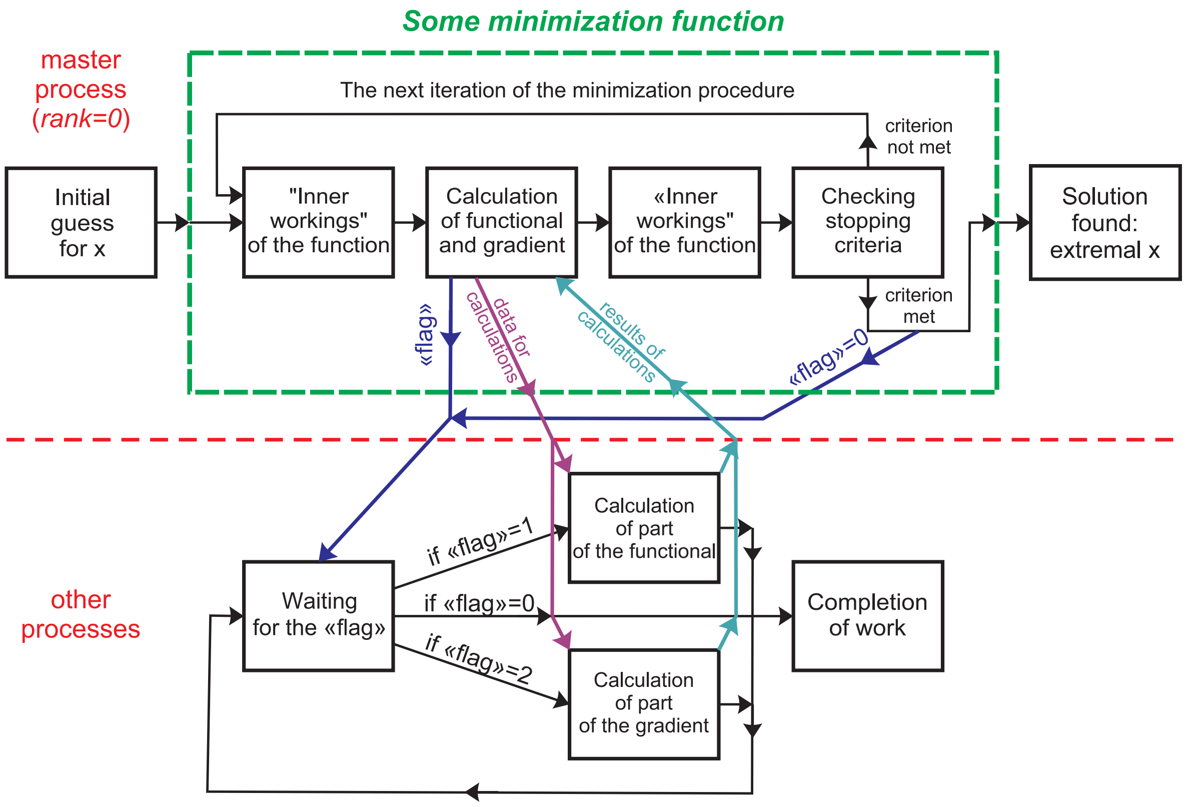
Some minimization function
master process (rank=0)
The next iteration of the minimization procedure
criterion not met
Initial guess for x
"Inner workings" of the function
Calculation of functional and gradient
«Inner workings" of the function
Checking stopping criteria
Solution found: extremal x
«flag»
data for calculations
results of calculations
criterion met
«flag»=0
other processes
Calculation of part of the functional
if «flag»=1
Waiting for the «flag»
if «flag»=0
Completion of work
if «flag»=2
Calculation of part of the gradient

Figure 2. Block diagram of a parallel algorithm.

```
if $rank$=0 then
    $x \leftarrow x_{init}$
    some minimization function
        while stopping criterion not met do
            ...some "inner workings"...
            if functional_calculation started then
                Bcast($flag = 1$) from $rank$=0 to all
                Scatter($x \rightarrow x_{part}$) from $rank$=0 to all
                fun$_{\text{part}}$ $\leftarrow$ functional_part($x_{part}$)
                Reduce(fun$_{\text{part}}$ $\rightarrow$ fun) to $rank$=0 from all
            end
            ...some "inner workings"...
            if gradient_calculation started then
                Bcast($flag = 2$) from $rank$=0 to all
                Scatter($x \rightarrow x_{part}$) from $rank$=0 to all
                grad$_{\text{part}}$ $\leftarrow$ gradient_part($x_{part}$)
                Gather(grad$_{\text{part}}$ $\rightarrow$ grad) to $rank$=0 from all
            end
            ...some "inner workings"...
        end
    end
    Bcast($flag = 0$) from $rank$=0 to all                // extremal $x$ has been found
else
    while True do
        Bcast($flag$) from $rank$=0
        if $flag$=1 then
            Scatter($x \rightarrow x_{part}$) from $rank$=0
            fun$_{\text{part}}$ $\leftarrow$ functional_part($x_{part}$)
            Reduce(fun$_{\text{part}}$ $\rightarrow$ fun) to $rank$=0
        if $flag$=2 then
            Scatter($x \rightarrow x_{part}$) from $rank$=0
            grad$_{\text{part}}$ $\leftarrow$ gradient_part($x_{part}$)
            Gather(grad$_{\text{part}}$ $\rightarrow$ grad) to $rank$=0
        if $flag$=0 then break
    end
end
```

Algorithm 1: Pseudocode of a universal parallel algorithm. Frames highlight pairs of blocks that are executed in parallel. Distribution of data from MPI master process with `rank=0` to other MPI processes is carried out using the MPI routine of collective communication of processes `Scatter`(), but depending on the features of the minimized functional, the distributing data can be implemented with using one of the MPI routines `Bcast()`, `Scatterv`(), `Reduce_scatter`(), etc.

The Python code that implements this algorithm will be quite compact. Its structure in the form of Python pseudocode is presented below. At the same time, for clarity: 1) some of the arguments of the functions used are omitted, 2) the syntax is simplified.

```
if rank == 0 :
    x_init = random()
    x = optimize.minimize(fun=functional_calculation,
                          jac=gradient_calculation,
                          x0=x_init).x
    flag = 0
    Bcast(flag, root=0)
else:
    while True :
        Bcast(flag, root=0)
        if flag == 1 :
            Scatterv(x, x_part, root=0)
            fun_part = functional_part(x_part)
            Reduce(fun_part, fun, op=MPI.SUM, root=0)
        if flag == 2 :
            Scatterv(x, x_part, root=0)
            grad_part = gradient_part(x_part)
            Gatherv(grad_part, grad, root=0)
        if flag == 0 :
            break
```

Here the functions `functional_calculation()` and `gradient_calculation()`, which are called by the master process inside the minimization function `optimize.minimize()` from the SciPy package, have the following structure:

```
def functional_calculation(x) :
    flag = 1
    Bcast(flag, root=0)
    Scatterv(x, x_part, root=0)
    fun_part = functional_part(x_part)
    Reduce(fun_part, fun, op=MPI.SUM, root=0)
    return fun
def gradient_calculation(x) :
    flag = 2
    Bcast(flag, root=0)
    Scatterv(x, x_part, root=0)
    grad_part = gradient_part(x_part)
    Gatherv(grad_part, grad, root=0)
    return grad
```

and the auxiliary functions `functional_part()` and `gradient_part()`, which are called by all processes, have the following general form:

```
def functional_part(x_part):
    fun_part = 0
    for i in range(len(x_part)) :
        fun_part += ...
    return fun_part
def gradient_part(x_part):
    grad_part = empty(len(x_part))
    for i in range(len(x_part)) :
        grad_part[i] = ...
    return grad_part
```

Note. Here it is assumed that in order to calculate its part of the functional or its gradient, each MPI process must know not all the components of the next approximation for the vector x, which is contained in the array `x`, but only part of its components , which are contained in the array `x_part`. Situations are quite common when each MPI process must know all the components of the next approximation for the vector x. In this case, you may need to use the MPI routine `Bcast()` instead of the MPI routine `Scatter(v)()`.

3. Description of the Full Version of the Software Implementation of the Parallel Algorithm (Using the MPI-4 Standard)

Next, we will describe the complete Python code that implements the algorithm 1. But first, the following remark must be made. The software implementation structure proposed in the previous section contains collective communications between processes with the same argument list which are repeatedly executed within the inner loop of a parallel computation. In such a situation, it may be possible to optimize the communications by binding the list of communication arguments to a persistent communication request once and, then, repeatedly using the request to initiate and complete messages. The peculiarity of the MPI-4 standard (2021) is that it introduced MPI routines that allow this to be done for operations of collective communications of processes.

For example, the result of running the MPI routine `Bcast()` is equivalent to the sequence of launching the MPI routines `Bcast_init()` "+" `Start()` "+" `Wait()`. Moreover, if the MPI routine `Bcast_init()` is launched multiple times with the same set of arguments, the result of calling `Bcast_init()` will be the same. This makes it possible to move this action "out of brackets", that is, out of the loop through which it is called repeatedly.

That is, in order to use more advanced versions of the MPI routines for collective communication of processes from the MPI-4 standard, it is necessary to perform the following sequence of changes.

1. Before the main loop `while`, it is necessary to generate persistent communication requests using MPI routines of the form `request[] = Bcast_init()` for all routines of collective communications of processes `Bcast()`, `Scatterv()`, `Reduce()` and `Gatherv()`, which are called multiple times with the same set of arguments.
2. Replace MPI routine calls with a sequence of function calls `Start(request[])` "+" `Wait(request[])`.

So, taking into account the comments made, the parallel software implementation of the algorithm 1 will take the following form (in this case, the program code will be commented block by block, but all blocks with numbered lines will form a single program code).

```
1  from mpi4py import MPI
2  from scipy import optimize
3  from numpy import array, empty, zeros, arange, float64, int32
4
5  comm = MPI.COMM_WORLD
6  numprocs = comm.Get_size()
7  rank = comm.Get_rank()
```

Lines 1–3 import the necessary functions. When importing MPI (line 1), the MPI part of the program is initialized. Based on the results of lines 5-7, each MPI process, on which this program code is executed, knows 1) the total number `numprocs` of processes participating in the calculations, and 2) its identifier `rank` in the communicator `comm`, which contains all the processes on which the program runs.

```
8  N = 1000
9
10 if rank == 0 :
11     ave, res = divmod(N, numprocs)
```

```
12      counts = array([ave + 1 if p < res else ave
13                      for p in range(numprocs)], dtype=int32)
14      displs = array([counts[:i].sum()
15                      for i in range(numprocs)], dtype=int32)
16  else:
17      counts, displs = None, None
18
19  N_part = array(0, dtype=int32)
20  comm.Scatter([counts, 1, MPI.INT], [N_part, 1, MPI.INT], root=0)
```

In line 8, the number of components of the model vector x is determined, and then in lines 11–15, on the master process with `rank = 0`, auxiliary arrays `counts` and `displs`, necessary for MPI routines `Scatterv()` and `Gatherv()` (which will be used to distribute components of the vector x among processes or to collect intermediate calculation data from all processes), are prepared. Thus, the array `counts` contains information about the number of elements sending to each process (or received from each process). Those, array element `counts[k]` contains the value $N_{part\,(k)}$ — number of components of part x_{part} of vector x, for which MPI process with `rank = k` is responsible for processing. The algorithm for determining the elements of `counts` is such that the maximum difference between any two elements of this array is 1, but the sum of all elements of this array is equal to N:

$$\sum_{k=0}^{\texttt{numprocs}-1} \texttt{counts[k]} \equiv \sum_{k=0}^{\texttt{numprocs}-1} N_{part\,(k)} = N.$$

It should be noted that the values of the array elements `counts` and `displs` will be needed further only on the process with `rank = 0` (master process). These values will not be used by other processes. But taking into account the fact that the arrays `counts` and `displs` will be arguments to the functions `Scatterv()` and `Gatherv()`, which are called on all processes, then the corresponding Python objects formally must be initialized. This is done on line 17 for processes with `rank >= 1`.

Line 19 allocates memory space for the array `N_part`, which will contain only one value — the number of elements of the vector part x_{part}, for which the MPI process, on which this program code is running, is responsible. It is necessary to recall that formally this number must be a `numpy`-array, since the MPI routines used only work with `numpy`-arrays. In line 20, using the MPI routine `Scatter()`, this array is filled with the corresponding value from the array `counts`, contained only in the process with `rank = 0`. Now the MPI process with `ramk = k` knows its value $N_{part\,(k)}$, contained on this MPI process in the array `N_part`.

Next, an array `x_model` is formed, which contains the values of the model vector x^{model}, with which the solution to the minimization problem will be compared.

```
21  if rank == 0 :
22      x_model = arange(N, dtype=float64)
23  else:
24      x_model = None
25
26  x_model_part = empty(N_part, dtype=float64)
27  comm.Scatterv([x_model, counts, displs, MPI.DOUBLE],
28                [x_model_part, N_part, MPI.DOUBLE], root=0)
```

In lines 21–24 this array is formed only on the master process with `rank = 0` (line 22), and then in lines 27–28, using the MPI routine `Scatterv()`, it is distributed in parts from the process with `rank = 0` across all processes where the corresponding parts are stored in the array `x_model_part`. To do this, in line 26, memory space is preliminarily allocated for these arrays.

Now begins the consideration of the main part of the program, which directly implements the algorithm considered in the work.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 14 November 2024 doi:10.20944/preprints202406.0571.v4

First, it is necessary to generate persistent communication requests for all MPI routines of collective communications of processes that will be called repeatedly.

```
29 requests = [MPI.Request() for i in range(4)]
30
31 flag = empty(1, dtype=int32)
32 requests[0] = comm.Bcast_init([flag, 1, MPI.INT], root=0)
33
34 if rank == 0 :
35     x_temp = empty(N, dtype=float64)
36 else :
37     x_temp = None
38 x_part = empty(N_part, dtype=float64)
39 requests[1] = comm.Scatterv_init(
                    [x_temp, counts, displs, MPI.DOUBLE],
40                           [x_part, N_part, MPI.DOUBLE], root=0)
41
42 if rank == 0 :
43     fun = empty(1, dtype=float64)
44 else :
45     fun = None
46 fun_part = empty(1, dtype=float64)
47 requests[2] = comm.Reduce_init([fun_part, 1, MPI.DOUBLE],
48                        [fun, 1, MPI.DOUBLE], op=MPI.SUM, root=0)
49
50 if rank == 0 :
51     grad = empty(N, dtype=float64)
52 else :
53     grad = None
54 grad_part = empty(N_part, dtype=float64)
55 requests[3] = comm.Gatherv_init(
                  [grad_part, N_part, MPI.DOUBLE],
56                        [grad, counts, displs, MPI.DOUBLE], root=0)
```

Line 29 creates the array `requests`, which will contain the values of the returned parameters, which, in turn, will be used to identify specific persistent communication requests for collective communications. Next, persistent communication requests are generated for the MPI routines `Bcast()` (lines 31–32), `Scatterv()` (lines 34–40), `Reduce()` (lines 42 —48), `Gatherv()` (lines 50–56).

The generation of such persistent communication requests for collective communications is the same, so it will be explained only using the example of the formation of a persistent communication request for collective communication for the MPI routine `Scatterv()` (lines 34–40). Line 35 allocates memory space for the array `x_temp` only on the process with `rank = 0` (this array will contain the values of the vector x, but the meaning of the postfix `_temp` will be explained later when filling this array with specific values). Its contents will not be used by other processes. But taking into account the fact that the array `x_temp` is an argument of the MPI routines `Scatterv_init()` (lines 39–40), which are called on all processes, the corresponding Python object must formally be initialized. This is done on line 37 for processes with `rank >= 1`. Line 38 allocates memory space for the array `x_part`.

It is necessary to note the following important point in the formation of persistent communication requests. The arguments of the corresponding functions are `numpy`-arrays (for example, in the case considered, the arrays `x_temp` and `x_part`), namely fixed memory areas associated with these arrays. When initializing generated communication requests using the MPI routine `MPI.Prequest.Start()`, all data will be taken/written to these memory areas, which are fixed once when the corresponding persistent communication request is generated. Therefore, it is necessary to first allocate space in memory for all arrays that are arguments to these functions; and during subsequent calculations, ensure that the corresponding calculation results are stored in the correct memory areas.

Next, the auxiliary functions `functional_part()` and `gradient_part()` are defined, which are called by all processes.

```python
def functional_part(x_part) :
    fun_part[0] = 0
    for i in range(N_part) :
        for n in range(N) :
            fun_part[0] += (x_part[i] - x_model_part[i]) ** 2
    return fun_part[0]

def gradient_part(x_part) :
    grad_part = zeros(N_part, dtype=float64)
    for i in range(N_part) :
        for n in range(N) :
            grad_part[i] += 2*(x_part[i] - x_model_part[i])
    return grad_part
```

These auxiliary functions contain calculations of the functional, which for simplicity is given as

$$f(x) = \sum_{i=1}^{N}\sum_{n=1}^{N}(x_i - i)^2, \quad \big(\operatorname{grad} f(x)\big)_i = 2\sum_{n=1}^{N}(x_i - i).$$

Next, the functions `functional_calculation()` and `gradient_calculation()` are defined, which are called by the master process with `rank = 0` inside the minimization function `optimize.minimize()` from the SciPy package.

```python
def functional_calculation(x) :
    flag[0] = 1
    MPI.Prequest.Start(requests[0])
    MPI.Request.Wait(requests[0], status=None)
    x_temp[:] = x
    MPI.Prequest.Start(requests[1])
    MPI.Request.Wait(requests[1], status=None)
    fun_part[0] = functional_part(x_part)
    MPI.Prequest.Start(requests[2])
    MPI.Request.Wait(requests[2], status=None)
    return fun[0]

def gradient_calculation(x) :
    flag[0] = 2
    MPI.Prequest.Start(requests[0])
    MPI.Request.Wait(requests[0], status=None)
    x_temp[:] = x
    MPI.Prequest.Start(requests[1])
    MPI.Request.Wait(requests[1], status=None)
    grad_part[:] = gradient_part(x_part)
    MPI.Prequest.Start(requests[3])
    MPI.Request.Wait(requests[3], status=None)
    return grad
```

The structure of these functions fully corresponds to the structure proposed at the end of the previous section (Section 2). Therefore, it is necessary to note only the features of their software implementation. This will be done using the function `functional_calculation()` as an example.

In lines 72–73, the functionality of the MPI routine `Bcast()` is executed, the persistent communication request for which was generated in line 32. Since the corresponding MPI routine is non-blocking, we must wait until the end of its execution (line 73).

In lines 75–76, the functionality of the MPI routine `Scatterv()` is executed, the persistent communication request for which was generated in lines 39–40. Since the corresponding MPI routine is non-blocking, we must wait for its execution to complete (line 76).

In this case, it is necessary to note the role of the array `x_temp` (line 74). This array will contain the values of the vector x, which could not be redefined, but directly distributed between other processes. But due to the fact that it is often unknown how the minimization function is structured internally, which at each iteration redefines the vector x, it may turn out that a new place in memory is allocated for the next approximation of this vector at this iteration. Therefore, for the purpose of insurance, the values of this array are copied to a pre-allocated fixed location in memory (line 35) in order for the function `Scatterv()`, generated by a persistent communication request, to work correctly.

In lines 78–79, the functionality of the MPI routine `Reduce()` is executed, the persistent communication request for which was generated in lines 47–48. Since the corresponding MPI routine is non-blocking, we must wait for its execution to complete (line 79).

The following is the Python code that implements the main part of the algorithm 1.

```
93  if rank == 0 :
94      x_init = random.uniform(low=0., high=N, size=N)
95      x = optimize.minimize(fun=functional_calculation,
96                            jac=gradient_calculation,
97                            x0=x_init, tol=1e-9,
98                            method='L-BFGS-B').x
99      flag[0] = 0
100     MPI.Prequest.Start(requests[0])
101     MPI.Request.Wait(requests[0], status=None)
102 else:
103     while True :
104         MPI.Prequest.Start(requests[0])
105         MPI.Request.Wait(requests[0], status=None)
106         if flag == 1 :
107             MPI.Prequest.Start(requests[1])
108             MPI.Request.Wait(requests[1], status=None)
109             fun_part[0] = functional_part(x_part)
110             MPI.Prequest.Start(requests[2])
111             MPI.Request.Wait(requests[2], status=None)
112         if flag == 2 :
113             MPI.Prequest.Start(requests[1])
114             MPI.Request.Wait(requests[1], status=None)
115             grad_part[:] = gradient_part(x_part)
116             MPI.Prequest.Start(requests[3])
117             MPI.Request.Wait(requests[3], status=None)
118         if flag == 0 :
119             break
```

The structure of this code fully corresponds to the structure proposed at the end of the previous section (Section 2). Therefore, it is necessary to note only the features of its software implementation.

In lines 100–101 and 104–105, the functionality of the MPI routine `Bcast()` is executed, the persistent communication request for which was generated in line 32.

In lines 107–108 and 113–114, the functionality of the MPI routine`Scatterv()` is executed, the persistent communication request for which was generated in lines 39–40.

In lines 110–111, the functionality of the MPI routine `Reduce()` is executed, the persistent communication request for which was generated in lines 47–48.

In lines 116–117, the functionality of the MPI routine `Gatherv()` is executed, the persistent communication request for which was generated in lines 55–56.

4. Efficiency and Scalability of Software Implementation of the Proposed Parallel Algorithm

To test the efficiency and scalability of the proposed software implementation of the parallel algorithm, the computing section "test" of the supercomputer "Lomonosov-2" [32] of the Research Computing Center of Lomonosow Moscow State University was used. Each computing node in the "test" section contains a 14-core Intel Xeon E5-2697 v3 2.60GHz processor with 64 GB of RAM (4.5 GB per core) and a Tesla K40s video card with 11.56 GB of video memory (not used in calculations).

The parallel version of the program was launched with each MPI process bound to exactly one core. The software packages used were 1) `mpi4py` version 4.0.0.dev0, 2) `numpy` version 1.26.4, 2) `scipy` version 1.12.0, 4) `mpich` version 4.1.1.

The scheme of numerical experiments repeats the scheme described in the work [33]. The program was launched on a number of processes $n \equiv$ `numprocs`, for which the running time T_n of the computational part of the program was recorded (see Figure 3). Using the estimated running time T_1 of the sequential version of the algorithm, the speedup S_n was calculated using the formula $S_n = \dfrac{T_1}{T_n}$ (see Figure 3), and then efficiency $E_n = \dfrac{S_n}{n}$ of software implementation (see Figure 4).

The calculations were carried out for the parameter $N = 8050$ (number of components in the model vector x), for which the running time of the computational part of the serial version of the program was $\sim$ 752 seconds. The results shown in Figure 3 and 4 correspond to the averaged values over a series of runs (100 runs for each value n). At the same time, the graphs also show a corridor of errors $S_n \pm \Delta S_n$ and $E_n \pm \Delta E_n$.

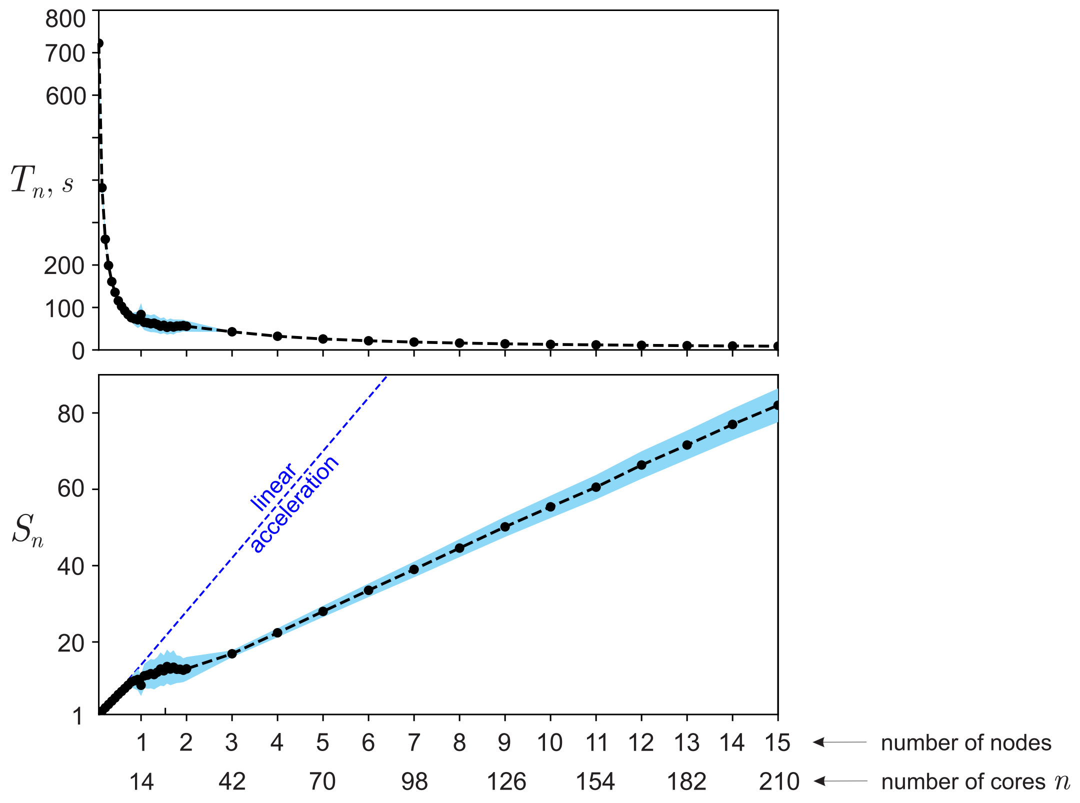

Figure 3. Graphs of the running time and acceleration of the parallel version of the program depending on the number of cores used for calculations (14 cores on one computing node).

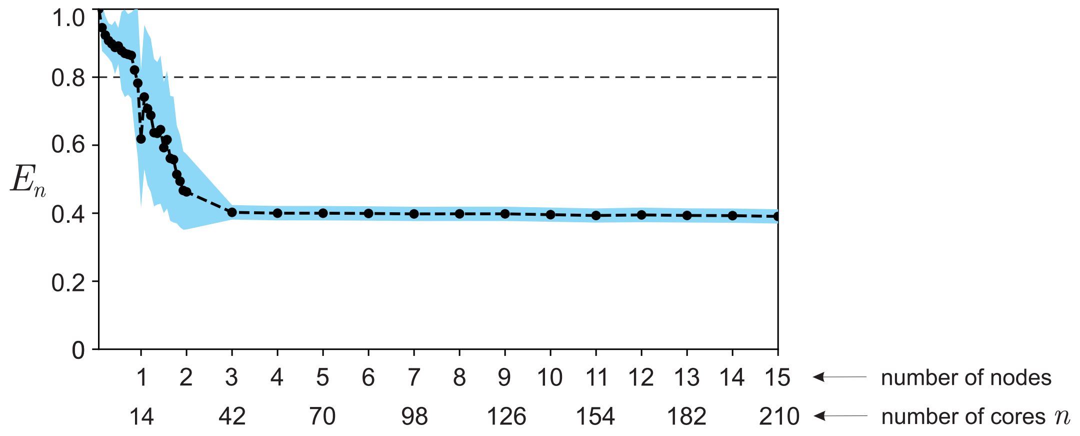

Figure 4. Graphs of parallelization efficiency depending on the number of cores used for calculations (14 cores on one computing node) in case of using `mpi4py` + `mpich`.

Errors for the obtained average values of acceleration and efficiency were calculated using the error formula for indirect measurements. In general, such a formula for the quantity N, depending on direct measurements x_j, $j = \overline{1, m}$, with known standard deviations $\sigma(x_j)$, has the form

$$\Delta N(x_1, \ldots, x_m) = \sqrt{\sum_{j=1}^{m} \left(\frac{\partial N}{\partial x_j} \sigma(x_i) \right)^2}.$$

Thus, for the acceleration calculated by the formula $S_n = \frac{T_1}{T_n}$ the errors are calculated as

$$\Delta S_1 = 0; \quad \Delta S_n = \frac{1}{T_n}\sqrt{\sigma^2(T_1) + \frac{T_1^2}{T_n^2}\sigma^2(T_n)}, \quad n \geq 2,$$

where $\sigma(T_n)$ — estimate of the standard deviation for each measurement T_n.

For efficiency calculated using the formula $E_n = \frac{S_n}{n}$, the errors are calculated as

$$\Delta E_1 = 0; \quad \Delta E_n = \frac{\Delta S_n}{n}, \quad n \geq 2.$$

As can be seen from Figure 4 the efficiency decreases in the interval from $n = 1$ to $n = 28$ and then remains constant. Formally, one could say that from this figure it can be concluded that the parallel software implementation of the algorithm, within certain limits, has good efficiency and good strong scalability over most of the range of the number of cores used for calculations. However, the indicated period of change in efficiency raises questions. Such a result would be easy to explain only in the case of a sharp deterioration in efficiency, starting from $n = 15$. When $n = \overline{1,14}$, only one node is used for calculations, as a result of which there is practically no overhead for the interaction of computing nodes (message transmission over the network is not used). Starting from $n = 15$, 2 nodes already take part in the calculations, as a result of which the communication network is involved and significant overhead may appear for receiving/transmitting messages between computing nodes. Moreover, with an increase in the number of computing nodes used, the share of costs for receiving/transmitting messages should only increase.

The study of this issue revealed the following feature of the combination of software packages used for test calculations. With the current settings of the installed packages and the "Lomonosov-2" supercomputer, when running programs using the `mpirun` utility from the `mpich` package, about 0-4 cores appear on each computing node, operating on average 2.28 times slower than the rest. A similar effect was also reproduced on a benchmark, when we ran the same computational task on multiple nodes/cores (without interaction between processes). The operating time of the entire parallel program is determined by the operating time of the slowest process, and as the number of computing processes n increases (each of which is executed on its own core of a multi-core processor), the probability of using "slow" cores increases, which is why the efficiency decreases relatively smoothly on the first two computing nodes, and then stops at the value ~ 0.4 (i.e., starting from three computing nodes, with a probability close to one, "slow" cores begin to be used).

For values of the number of cores $n = \overline{2,28}$, a graph was constructed showing the distribution of efficiency values over a series of runs for each value of n (100 runs for each value). The resulting distributions (see Figure 5) are not normal, since they contain two pronounced peaks: in the presence and absence of "slow" cores. Therefore, using the average of these values (see Figure 4) to characterize the properties of strong scalability of a software implementation of a parallel algorithm is not entirely correct — it is better to use values corresponding to the maximums of the distribution, since these values correspond to the case of using cores with the same performance (and our algorithm is designed for the same performance of cores).

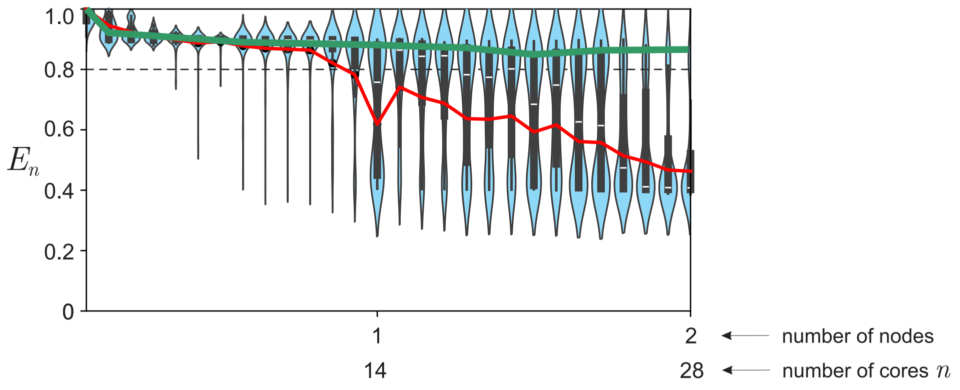

Figure 5. Graph of the efficiency distribution over a series of launches for each value of n. The red curve displays average efficiency values (efficiency drops until there is "guaranteed" that there will be at least one "slow" core among all cores). The green curve displays the dependence of efficiency when taking into account only those values that correspond to the maxima of the distribution, corresponding to "fast" cores.

The support of the Losonosov-2 supercomputer recommended that we recalculate the results using OpenMPI (instead of mpich), since no problems with cores hanging during program operation were found on this MPI implementation. We performed calculations using OpenMPI 5.0.5 (while UCX support was disabled due to tcp timeout errors in the current version of the library). The results are shown in Figure 6.

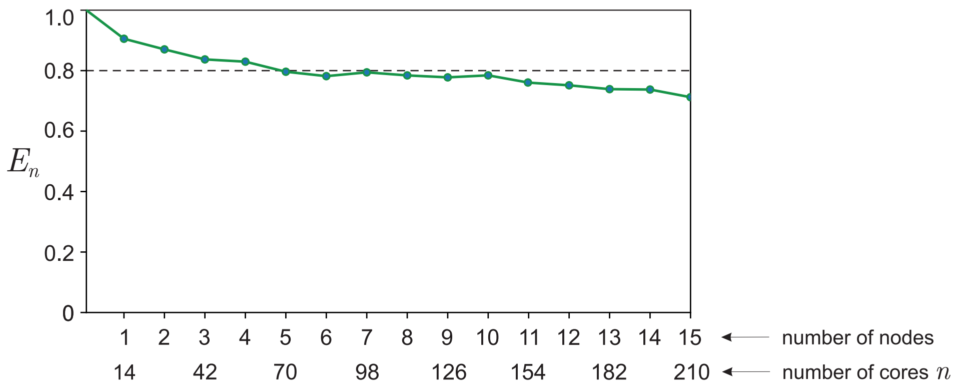

Figure 6. Graph of parallelization efficiency in case of using `mpi4py` + `openmpi`.

5. Discussion

1. A question similar to the one discussed in this article has been raised by other authors. For example, at work [34] general strategies for metaheuristics were classified. According to this article, our work uses the so-called "type 1 parallelism of meta-heuristics", as what the user programs himself is within the optimization loop and the outer control flow is unchanged. The papers [35,36] discuss common approaches to parallelizing existing sequential programs with minimal effort on the part of users. We also assume that the approach we have considered is used by many scientists as an intermediate stage of scientific work, but we are not aware of any relevant detailed descriptions in the literature.
2. When solving many applied problems, the minimized functionals can be arranged in such a way that when each MPI-process calculates its part of the functional/gradient, such functions Python will be used, which use shared memory parallelism within the cores of a node (for example, the dot() function of the package numpy). In this case, it is constructive to use not one core of a multi-core processor as an MPI-node, but a computing node that contains such a multi-core processor (see, for example, work [33]). This approach will reduce the overhead of receiving/sending intermediate calculation data through a communication network using MPI

technology. As a result, the efficiency of software implementation will increase. It is possible to act similarly if there are GPUs on the computing nodes. In this work, we do not consider examples of software implementation of the corresponding approaches, since the corresponding software implementations strongly depend on the type of specific functional being minimized and, as a consequence, on the ability to use built-in functions that use shared memory parallelism (CPU or GPU).

3. Some minimization functions use the BLAS [37], LAPACK [38], Intel MKL etc. libraries. In this case, they can use all the cores of a multi-core processor. As a result, the efficiency of parallelization within a single computing node can drop. It makes sense to disable built-in parallelism in the case when it implies shared memory parallelism within the cores of a node, and the user has access to many computing nodes.
4. The operation of sending a "flag" can be combined with sending data for calculations. We did not do this to avoid unnecessary complexity of the algorithm description.
5. The classes of minimization problems that can be solved are limited only by the available minimization functions in the software package used. For example, in the SciPy library of the Python programming language, there are functions for minimizing linear and nonlinear functionals, with or without constraints, etc. The argument "method" of the function optimize.minimize() used in the program code must be changed accordingly.

6. Conclusion

The approach to parallelization considered in the paper assumes that the computations required to evaluate the minimized functional can be parallelized by the user. In this case, our software implementation can be used as a template for "parallelizing" an arbitrary minimization function available in any software package.

Author Contributions: Conceptualization, D.L. and S.T.; methodology, D.L. and S.T.; software, D.L. and S.T.; validation, D.L., S.T. and V.S.; formal analysis, S.T. and V.S.; investigation, D.L., S.T. and V.T.; data curation, S.T. and V.S.; writing—original draft preparation, D.L.; writing—review and editing, D.L.; visualization, D.L.; supervision, D.L.; project administration, D.L.; funding acquisition, D.L. All authors have read and agreed to the published version of the manuscript.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: The research is carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University [32].

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Virtanen, P.; Gommers, R.; Oliphant, T.E.; Haberland, M.; Reddy, T.; Cournapeau, D.; Burovski, E.; Peterson, P.; Weckesser, W.; Bright, J.; van der Walt, S.J.; Brett, M.; Wilson, J.; Millman, K.J.; Mayorov, N.; Nelson, A.R.J.; Jones, E.; Kern, R.; Larson, E.; Carey, C.J.; Polat, İ.; Feng, Y.; Moore, E.W.; VanderPlas, J.; Laxalde, D.; Perktold, J.; Cimrman, R.; Henriksen, I.; Quintero, E.A.; Harris, C.R.; Archibald, A.M.; Ribeiro, A.H.; Pedregosa, F.; van Mulbregt, P.; SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* **2020**, *17*, 261–272. doi:10.1038/s41592-019-0686-2.
2. Nelder, J.A.; Mead, R. A Simplex Method for Function Minimization. *The Computer Journal* **1965**, *7*, 308–313. doi:10.1093/comjnl/7.4.308.
3. Powell, M.J.D. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The Computer Journal* **1964**, *7*, 155–162. doi:10.1093/comjnl/7.2.155.
4. Hestenes, M.R.; Stiefel, E. Methods of conjugate gradients for solving linear systems. *J Res NIST* **1952**, *49*, 409–436. doi:10.6028/jres.049.044.
5. Broyden, C. A new double-rank minimisation algorithm. Preliminary report. *American Mathematical Society, Notices* **1969**, *16*, 670.

6. Fletcher, R. A new approach to variable metric algorithms. *The computer journal* **1970**, *13*, 317–322.
7. Goldfarb, D. A family of variable-metric methods derived by variational means. *Mathematics of Computation* **1970**, *24*, 23–26. doi:10.1090/S0025-5718-1970-0258249-6.
8. Shanno, D.F. Conditioning of Quasi-Newton Methods for Function Minimization. *j-MATH-COMPUT* **1970**, *24*, 647–656. doi:10.2307/2004840.
9. Dembo, R.S.; Steihaug, T. Truncated-Newton algorithms for large-scale unconstrained optimization. *Mathematical Programming* **1983**, *26*, 190–212. doi:10.1007/BF02592055.
10. Byrd, R.H.; Lu, P.; Nocedal, J.; Zhu, C. A Limited Memory Algorithm for Bound Constrained Optimization. *SIAM Journal on Scientific Computing* **1995**, *16*, 1190–1208. doi:10.1137/0916069.
11. Nash, S.G. Newton-Type Minimization via the Lanczos Method. *SIAM Journal on Numerical Analysis* **1984**, *21*, 770–788. doi:10.1137/0721052.
12. Powell, M.J.D., A Direct Search Optimization Method That Models the Objective and Constraint Functions by Linear Interpolation. In *Advances in Optimization and Numerical Analysis*; Gomez, S.; Hennart, J.P., Eds.; Springer Netherlands, 1994; pp. 51–67. doi:10.1007/978-94-015-8330-5_4.
13. Kraft, D. *A Software Package for Sequential Quadratic Programming*; Deutsche Forschungs- und Versuchsanstalt für Luft- und Raumfahrt Köln: Forschungsbericht, Wiss. Berichtswesen d. DFVLR, 1988.
14. Byrd, R.H.; Hribar, M.E.; Nocedal, J. An Interior Point Algorithm for Large-Scale Nonlinear Programming. *SIAM Journal on Optimization* **1999**, *9*, 877–900. doi:10.1137/S1052623497325107.
15. Lalee, M.; Nocedal, J.; Plantenga, T. On the Implementation of an Algorithm for Large-Scale Equality Constrained Optimization. *SIAM Journal on Optimization* **1998**, *8*, 682–706. doi:10.1137/S1052623493262993.
16. A hybrid method for nonlinear equations. In *Numerical methods for nonlinear algebraic equations*; Gordon and Breach, 1970; pp. 87–114.
17. Large-Scale Unconstrained Optimization. In *Numerical Optimization*; Springer New York: New York, NY, 2006; pp. 164–192. doi:10.1007/978-0-387-40065-5_7.
18. Lenders, F.; Kirches, C.; Potschka, A. trlib: a vector-free implementation of the GLTR method for iterative solution of the trust region problem. *Optimization Methods and Software* **2018**, *33*, 420–449.
19. Gould, N.I.M.; Lucidi, S.; Roma, M.; Toint, P.L. Solving the Trust-Region Subproblem using the Lanczos Method. *SIAM Journal on Optimization* **1999**, *9*, 504–525. doi:10.1137/S1052623497322735.
20. Conn, A.R.; Gould, N.I.M.; Toint, P.L., Trust Region Methods; Society for Industrial and Applied Mathematics, 2000; pp. 169–200. doi:10.1137/1.9780898719857.
21. Wales, D.J.; Doye, J.P.K. Global Optimization by Basin-Hopping and the Lowest Energy Structures of Lennard-Jones Clusters Containing up to 110 Atoms. *The Journal of Physical Chemistry A* **1997**, *101*, 5111–5116. doi:10.1021/jp970984n.
22. Johansson, R., Optimization. In *Numerical Python: A Practical Techniques Approach for Industry*; Apress: Berkeley, CA, 2015; pp. 147–168. doi:10.1007/978-1-4842-0553-2_6.
23. Storn, R.; Price, K. Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. *Journal of Global Optimization* **1997**, *11*, 341–359. doi:10.1023/A:1008202821328.
24. Endres, S.; Sandrock, C.; Focke, W. A simplicial homology algorithm for Lipschitz optimisation. *Journal of Global Optimization* **2018**, *72*, 181–217. doi:10.1007/s10898-018-0645-y.
25. Xiang, Y.; Gubian, S.; Suomela, B.; Hoeng, J. Generalized Simulated Annealing for Global Optimization: The GenSA Package. *The R Journal* **2013**, *5*, 13–28. doi:10.32614/RJ-2013-002.
26. Jones, D.R.; Perttunen, C.D.; Stuckman, B.E. Lipschitzian optimization without the Lipschitz constant. *Journal of Optimization Theory and Applications* **1993**, *79*, 157–181. doi:10.1007/BF00941892.
27. Gablonsky, J.M.; Kelley, C.T. A Locally-Biased form of the DIRECT Algorithm. *Journal of Global Optimization* **2001**, *21*, 27–37. doi:10.1023/A:1017930332101.
28. Balay, S.; Abhyankar, S.; Adams, M.F.; Benson, S.; Brown, J.; Brune, P.; Buschelman, K.; Constantinescu, E.M.; Dalcin, L.; Dener, A.; Eijkhout, V.; Faibussowitsch, J.; Gropp, W.D.; Hapla, V.; Isaac, T.; Jolivet, P.; Karpeev, D.; Kaushik, D.; Knepley, M.G.; Kong, F.; Kruger, S.; May, D.A.; McInnes, L.C.; Mills, R.T.; Mitchell, L.; Munson, T.; Roman, J.E.; Rupp, K.; Sanan, P.; Sarich, J.; Smith, B.F.; Zampini, S.; Zhang, H.; Zhang, H.; Zhang, J. PETSc Web page. https://petsc.org/, 2024.
29. Dalcin, L.D.; Paz, R.R.; Kler, P.A.; Cosimo, A. Parallel distributed computing using Python. *Advances in Water Resources* **2011**, *34*, 1124 – 1139. New Computational Methods and Software Tools.

30. Dalcin, L.; Fang, Y.L.L. mpi4py: Status Update After 12 Years of Development. *Computing in Science & Engineering* **2021**, *23*, 47–54. doi:10.1109/MCSE.2021.3083216.
31. Rogowski, M.; Aseeri, S.; Keyes, D.; Dalcin, L. mpi4py.futures: MPI-Based Asynchronous Task Execution for Python. *IEEE Transactions on Parallel and Distributed Systems* **2023**, *34*, 611–622. doi:10.1109/TPDS.2022.3225481.
32. Voevodin, V.; Antonov, A.; Nikitenko, D.; Shvets, P.; Sobolev, S.; Sidorov, I.; Stefanov, K.; Voevodin, V.; Zhumatiy, S. Supercomputer Lomonosov-2: Large Scale, Deep Monitoring and Fine Analytics for the User Community. *Supercomputing Frontiers and Innovations* **2019**, *6*, 4–11. doi:10.14529/jsfi190201.
33. Lukyanenko, D. Parallel algorithm for solving overdetermined systems of linear equations, taking into account round-off errors. *Algorithms* **2023**, *16*, 242. doi:10.3390/a16050242.
34. Crainic, T.G.; Toulouse, M., Parallel Strategies for Meta-Heuristics. In *Handbook of Metaheuristics*; Glover, F.; Kochenberger, G.A., Eds.; Springer US: Boston, MA, 2003; pp. 475–513. doi:10.1007/0-306-48056-5_17.
35. Nilsen, J.K.; Cai, X.; Høyland, B.; Langtangen, H.P. Simplifying the parallelization of scientific codes by a function-centric approach in Python. *Computational Science & Discovery* **2010**, *3*, 015003. doi:10.1088/1749-4699/3/1/015003.
36. Aldinucci, M.; Cesare, V.; Colonnelli, I.; Martinelli, A.R.; Mittone, G.; Cantalupo, B.; Cavazzoni, C.; Drocco, M. Practical parallelization of scientific applications with OpenMP, OpenACC and MPI. *Journal of Parallel and Distributed Computing* **2021**, *157*, 13–29. doi:https://doi.org/10.1016/j.jpdc.2021.05.017.
37. Blackford, L.S.; Petitet, A.; Pozo, R.; Remington, K.; Whaley, R.C.; Demmel, J.; Dongarra, J.; Duff, I.; Hammarling, S.; Henry, G.; others. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software* **2002**, *28*, 135–151.
38. Anderson, E.; Bai, Z.; Bischof, C.; Blackford, S.; Demmel, J.; Dongarra, J.; Du Croz, J.; Greenbaum, A.; Hammarling, S.; McKenney, A.; Sorensen, D. *LAPACK Users' Guide*, third ed.; Society for Industrial and Applied Mathematics: Philadelphia, PA, 1999.