Simplifying the parallelization of scientific codes by a function-centric approach in Python

To cite this article: Jon K Nilsen et al 2010 Comput. Sci. Disc. 3 015003

View the article online for updates and enhancements.
Simplifying the parallelization of scientific codes by a function-centric approach in Python

Jon K Nilsen\textsuperscript{1,2}, Xing Cai\textsuperscript{3,4}, Bjørn Høyland\textsuperscript{5} and Hans Petter Langtangen\textsuperscript{3,4}

\textsuperscript{1} USIT, PO Box 1059 Blindern, N-0316 Oslo, Norway
\textsuperscript{2} Department of Physics, PO Box 1048 Blindern, University of Oslo, N-0316 Oslo, Norway
\textsuperscript{3} Center for Biomedical Computing, Simula Research Laboratory, PO Box 134, N-1325 Lysaker, Norway
\textsuperscript{4} Department of Informatics, PO Box 1080 Blindern, University of Oslo, N-0316 Oslo, Norway
\textsuperscript{5} Department of Political Science, PO Box 1097 Blindern, University of Oslo, N-0317 Oslo, Norway

E-mail: j.k.nilsen@usit.uio.no, xingca@simula.no, bjorn.hoyland@stv.uio.no and hpl@simula.no

Received 17 January 2010, in final form 30 June 2010
Published 17 September 2010

Computational Science & Discovery \textbf{3} (2010) 015003 (24pp)

Abstract. The purpose of this paper is to show how existing scientific software can be parallelized using a separate thin layer of Python code where all parallelization-specific tasks are implemented. We provide specific examples of such a Python code layer, which can act as templates for parallelizing a wide set of serial scientific codes. The use of Python for parallelization is motivated by the fact that the language is well suited for reusing existing serial codes programmed in other languages. The extreme flexibility of Python with regard to handling functions makes it very easy to wrap up decomposed computational tasks of a serial scientific application as Python functions. Many parallelization-specific components can be implemented as generic Python functions, which may take as input those wrapped functions that perform concrete computational tasks. The overall programming effort needed by this parallelization approach is limited, and the resulting parallel Python scripts have a compact and clean structure. The usefulness of the parallelization approach is exemplified by three different classes of application in natural and social sciences.
1. Introduction

Due to the limited computing power of standard serial computers, parallel computing has become indispensable for investigating complex problems in all fields of science. A frequently encountered question is how to transform an existing serial scientific code into a new form that is executable on a parallel computing platform. Although portable parallel programming standards, such as message passing interface (MPI) and open multi-processing (OpenMP), have greatly simplified the programming work, the task of parallelization can still be quite complicated for domain scientists. This is because inserting MPI calls or OpenMP directives directly into an existing serial code often requires an extensive code rewrite as well as detailed knowledge of and experience with parallel programming.

The hope for non-specialists in parallel computing is that many scientific applications possess high-level parallelism. That is, the entire computational work can be decomposed into a set of individual (and often collaborative) computational tasks, each of coarse grain, and can be performed by an existing piece of serial code. Depending on the specific application, the decomposition can be achieved by identifying a set of different parameter combinations, or (fully or almost) independent computations or different data groups or different geometric subdomains. For a given type of decomposition, the parallelization-induced programming components, such as work partitioning, domain partitioning, communication, load balancing and global administration, are often generic and independent of specific applications. These generic components can thus be implemented as reusable parallelization libraries once and for all. This is what we exemplify in the present paper.

It is clear that a user-friendly parallelization approach relies on at least two factors: (i) the existing serial code should be extensively reused; and (ii) the programming effort by the end user must be limited. To achieve these goals, we suggest the use of Python to wrap up pieces of existing serial code (possibly written in other languages) and implement the parallelization tasks in separate and generic Python functions.

Python [1] is, at its core, an extremely expressive and flexible programming language. The language has been extended with numerous numerical and visualization modules such as NumPy [2] and SciPy [3]. The two
requirements of a user-friendly parallelization mentioned above are well met by Python. First of all, Python is good at inter-operation with other languages, especially Fortran, C and C++, which are used heavily in scientific codes. Using wrapper tools such as F2PY [4], it is easy to wrap up an existing piece of code in Fortran and C and provide it with a Pythonic appearance.

Moreover, among its many strong features, Python is extremely flexible with handling functions. Python functions accept both positional arguments and keyword arguments. The syntax of a variable set of positional and keyword arguments (known as ‘(*args, **kwargs)’ to Python programmers) allows the writing of libraries routines that work with any type of user-defined functions. That is, the syntax makes it possible to call a Python function without revealing the exact number of arguments.

It is also straightforward to pass functions as input arguments to a Python function and/or return a function as output. A callable class object in Python can be used as if it were a stand-alone function. Such a construction, or alternatively a closure (known from functional programming), can be used to create functions that carry a state represented through an arbitrarily complex data structure. The result is that one can express the flow of a scientific code as a Python program containing a set of calls to user-defined Python functions. These user-defined functions can be ordinary functions or classes that wrap pieces of the underlying scientific code. This is what we call a function-centric representation of the scientific code. With such a function-centric approach, we can build a general framework in Python for almost automatic parallelization of the program flow in the original scientific code. Later examples will convey this idea in detail.

Performance of the resulting parallel application will closely follow the performance of the serial application, because the overhead of the parallelization layer in Python is only due to a small piece of extra code, as we assume the main computational work to take place in the Python functions that call up pieces of the original scientific code. In the parallelization layer, good performance can be ensured by using efficient array modules in Python (such as numpy [2]) together with light-weight MPI wrappers (such as pypar [5]). For examples of writing efficient Python code segments for some standard serial and parallel numerical computations, we refer the reader to Cai et al [6].

Related work. In C++, generic programming via templates and object-oriented programming has been applied to parallelizing serial scientific codes. Two examples can be found in [7] and [8], where the former uses C++ class hierarchies to enable easy implementation of additive Schwarz preconditioners, and the latter uses C++ templates extensively to parallelize finite element codes. Many scientific computing frameworks have also adopted advanced programming to incorporate parallelism behind the scenes. In these frameworks (see e.g. [9–13]), the users can write parallel applications in a style quite similar to serial programming, without being exposed to many parallelizing details. Likewise for frameworks that are specially designed to allow coupling of different serial and parallel components, such as Cactus [14] and MpCCI [15]. The Python programming language, however, has not been widely used to parallelize existing serial codes. The Star-P system [16] provides the user with a programming environment where most of the parallelism is kept behind the scenes. Hinsen [17] combined Python with the bulk synchronous parallel (BSP) model to enable high-level parallel programming. In addition, quite a number of Python MPI wrappers exist, such as pypar [5], MPYMPI [19], mpi4py [20, 21] and ScientificMPI [22]. Efforts at incorporating parallelism via language extensions of Python can be found in [23–25].

The contribution of the present paper is to show by examples that a function-centric approach using Python may ease the task of parallel scientific programming. This result is primarily due to Python’s flexibility in function handling and function arguments. As a result, generic tasks that arise in connection with parallelization can often be programmed as a collection of simple and widely applicable Python functions, which are ready to be used by non-specialists to parallelize their existing serial codes.

This paper contains three examples with different algorithmic structures. A wide range of problems in science can be attacked by extending and adapting the program code in these examples. Moreover, readers whose problems are not covered by the examples will hopefully understand from these examples how we solve programming problems by identifying the principal, often simplified, underlying algorithmic structure; then creating generic code to reflect the structure; and finally applying the generic code to a specific, detailed case. Our approach is much inspired by the success of mathematics in problem solving, i.e. detecting the problem’s principal structure and devising a generic solution makes complicated problems
tractable. With Python as a tool, we demonstrate how this strategy carries over to parallelization of scientific codes.

The remainder of the paper is organized as follows. We give, in section 2, a simple but motivating example, explaining the principles of splitting a problem into a set of function calls that can easily be parallelized. Generic parallelization of three common types of real scientific applications are then demonstrated in section 3. Afterwards, section 4 reports the computational efficiency of the suggested parallelization approach applied to specific cases in the three classes of scientific problem. Some concluding remarks are given in section 5.

2. A motivating simple example

2.1. Serial version

Suppose we want to carry out a parameter analysis that involves a large number of evaluations of a multi-variable mathematical function \( f(a_1, \ldots, a_q) \). The Python implementation of \( f \) may use \( p \) positional arguments and \( k \) keyword arguments, such that the total \( p + k \) arguments contain at least the variables \( a_1, \ldots, a_q \) (i.e. \( q \leq p + k \)). As a very simple example, consider the parabola \( f(x, a, b, c) = ax^2 + bx + c \) with the following Python implementation (\( q = 4, p = 1, k = 3 \)):

```python
def func(x, a=0, b=0, c=1):
    return a*x**2+b*x+c
```

Suppose that we want to evaluate \( \text{func} \) for a particular set of input parameters chosen from a large search space, where \( x, a, b \) and \( c \) vary in specified intervals. The complete problem can be decomposed into three main steps: (1) initialize a set of arguments to \( \text{func} \); (2) evaluate \( \text{func} \) for each entry in the set of arguments; and (3) process the set of function return values from all the \( \text{func} \) calls.

Let us assume that step (1) calls a user-defined function \( \text{initialize} \) that returns a list of 2-tuples, where each holds one positional argument (as a list of \( x \) values) and three keyword arguments (as a three-item dictionary containing \( a, b \) and \( c \)) for a specific call to \( \text{func} \). Step (2) iterates over the list of 2-tuples from step (1) and feeds the positional and keyword arguments into \( \text{func} \). All the evaluated values of \( \text{func} \) are stored in a result list. Finally, step (3) processes the result list in another user-defined function \( \text{finalize} \).

A generic Python function that implements the three-step parameter analysis can be as follows:

```python
def solve_problem(initialize, func, finalize):
    input_args = initialize()
    output = [func(*args, **kwargs) for args, kwargs in input_args]
    finalize(output)
```

Note that the use of \textit{list comprehension}, i.e. line 3 in the above code, has given a very compact implementation of the for-loop for going through all the evaluations of \( \text{func} \). The \( \text{initialize}, \text{func} \) and \( \text{finalize} \) functions, which are independent of \( \text{solve_problem} \), are passed as input arguments.

As an example, assume that \( x \) is an array of \( n \) uniformly distributed coordinates in \([0, L]\), and we vary \( a \) and \( b \) inside \([-1, 1]\) each with \( m \) values, while \( c \) is fixed at the value 5. For each combination of \( a \) and \( b \), we call \( \text{func} \) with the \( x \) array as a positional argument and the \( a, b, c \) values as keyword arguments, and store the evaluation results of \( \text{func} \) in a list named output. The objective of the computations is to extract the \( a \) and \( b \) values for which \( \text{func} \) gives a negative value for one or several of the coordinates \( x \in [0, L] \). For this very simple example, the concrete implementation of the \( \text{initialize} \) and \( \text{finalize} \) functions can be put inside a class named \( \text{Parabola} \) as follows:

```python
class Parabola:
    def __init__(self, m, n, L):
        self.m, self.n, self.L = m, n, L

    def initialize(self):
        x = numpy.linspace(0, self.L, self.n)
        a_values = numpy.linspace(-1, 1, self.m)
```

b_values = numpy.linspace(-1, 1, self.m)
c = 5

self.input_args = []
for a in a_values:
    for b in b_values:
        func_args = ([x], {'a': a, 'b': b, 'c': c})
        self.input_args.append(func_args)
return self.input_args

def func(self, x, a=0, b=0, c=1):
    return a*x**2+b*x+c

def finalize(self, output_list):
    self.ab = []
    for input, result in zip(self.input_args, output_list):
        if min(result) < 0:
            self.ab.append((input[1]['a'], input[1]['b']))

Now, to find the combinations of \(a\) and \(b\) values that make \(ax^2 + bx + c < 0\), we can write the following two lines of code (assuming \(m = 100\), \(n = 50\) and \(L = 10\):

```python
problem = Parabola(100, 50, 10)
solve_problem(problem.initialize, problem.func, problem.finalize)
```

Note that the desired combinations of \(a\) and \(b\) values will be stored in the list `problem.ab`. Also note that we have placed `func` inside class `Parabola`, to have all the pieces of the problem in one place, but having `func` as a stand-alone function or a class method is a matter of choice.

Despite the mathematical simplicity of this example, the structure of the `solve_problem` function is directly applicable to a wide range of much more advanced problems. Although `initialize` and `finalize` are Python functions with very simple arguments (none and a list, respectively), this is not a limitation of their applicability. For example, the `initialize` step in our simple example needs values for \(m\), \(n\) and \(L\), the \(a\) and \(b\) interval and so on, which cannot be specified in the generic `solve_problem` function. To overcome this limitation, the information on \(m\), \(n\) and \(L\) can be hard-coded (not recommended) or transferred to `initialize` through global variables (not recommended in general) or carried with `initialize` as a state, either as class attributes or as a surrounding scope in a closure. We have chosen the class approach, i.e. class attributes store user-dependent data structures such that the `initialize` and `finalize` methods can have the simple input argument structure demanded by the generic `solve_problem` function. Alternatively, a closure as follows can be used instead of a class (this construct requires some knowledge of Python’s scoping rules):

```python
def initialize_wrapper(m, n, L):
    def initialize(self):
        x = numpy.linspace(0, L, n)
        a_values = numpy.linspace(-1, 1, m)
        ...,
        return input_args
    return initialize
```

Now, the returned `initialize` function will carry with it the values of \(m\), \(n\) and \(L\) in the surrounding scope. The choice between the class approach and the closure approach, or using global variables in a straightforward global `initialize` function, is up to the programmer. The important point here is that `initialize` is required to do a lot, and the input information to `initialize` must be handled by some Python construction. Similar comments apply to `finalize`.

2.2. Parallel version

Let us say that we want to utilize several processors to share the work of all the `func` evaluations, i.e. the `for`-loop in the generic `solve_problem` function. This can clearly be achieved by a task-parallel approach,
where each evaluation of func is an independent task. The main idea of parallelization is to split up the for-loop into a set of shorter for-loops, each assigned to a different processor. In other words, we need to split up the input_args list into a set of sub-lists for the different processors. Note that this partitioning work is generic, independent of both the func function and the actual arguments in the input_args list. Assuming homogeneous processors and that all the function evaluations are equally expensive, we can divide the input_args list into num_procs (number of processors) sub-lists of equal length. In case input_args is not divisible by num_procs, we adjust the length of some sub-lists by 1:

```python
def simple_partitioning(length, num_procs):
    sublengths = [length/num_procs]*num_procs
    for i in range(length % num_procs):  # treatment of remainder
        sublengths[i] += 1
    return sublengths

def get_subproblem_input_args(input_args, my_rank, num_procs):
    sub_ns = simple_partitioning(len(input_args), num_procs)
    my_offset = sum(sub_ns[:my_rank])
    my_input_args = input_args[my_offset:my_offset+sub_ns[my_rank]]
    return my_input_args
```

Using the above generic get_subproblem_input_args function, each processor gets its portion of the global input_args list, and a shorter for-loop can be executed there. Note that the syntax of Python lists and numpy arrays has made the function very compact.

The next step of parallelization is to collect the function evaluation results from all the processors into a single global output list. Finally, we let finalize(output) run only on the master processor (assuming that this work does not require parallelization). For the purpose of collecting outputs from all the processors, the following generic Python function can be used:

```python
def collect_subproblem_output_args(my_output_args, my_rank, num_procs, send_func, recv_func):
    if my_rank == 0:  # master process?
        output_args = my_output_args
        for i in range(1, num_procs):
            output_args += recv_func(i)
        return output_args
    else:
        send_func(my_output_args, 0)
        return None
```

The last two input arguments to the above collect_subproblem_output_args function deserve some attention. Both send_func and recv_func are functions themselves. In the simplest case of using the pypar wrapper of MPI commands, we can simply pass pypar.send as the send_func input argument and pypar.receive as recv_func. Moreover, switching to another MPI module is transparent with regard to the generic collect_subproblem_output_args function. (An example of using the mpi4py wrapper will be given at the end of this section.) It should also be noted that most Python MPI modules are considerably more user-friendly than the original MPI commands in C/Fortran. This is because (1) the use of keyword arguments greatly simplifies the syntax and (2) any picklable (marshalable) Python data type can be communicated directly.

Now that we have implemented the generic functions get_subproblem_input_args and collect_subproblem_output_args, we can write a minimalistic parallel solver as follows:

```python
def parallel_solve_problem(initialize, func, finalize, my_rank, num_procs, send, recv):
    input_args = initialize()
    my_input_args = get_subproblem_input_args(input_args, my_rank, num_procs)
    my_output = [func(*args, **kwargs) for args, kwargs in my_input_args]
output = collect_subproblem_output_args(my_output, my_rank, num_procs, send, recv)

if my_rank == 0:
    finalize(output)

We remark that the above parallel_solve_problem function is generic in the sense that it is independent of the actual implementation of initialize, func and finalize, as well as the Python MPI module being used. All problems that can be composed from independent function calls can (at least in principle) be parallelized by the small pieces of Python code shown.

As a specific example of using this parallel solver, we may address the problem of evaluating the parabolic function (func and class Parabola) for a large number of parameters. Using the pypar MPI module and having the problem-dependent code in a module named Parabola and the general function-centric tools in a module named function_centric, the program becomes the following:

```python
from Parabola import Parabola
from function_centric import parallel_solve_problem
import pypar

problem = Parabola(m=100, n=50, L=10)
my_rank = pypar.rank()
num_procs = pypar.size()
parallel_solve_problem(problem.initialize, problem.func, problem.finalize,
                      my_rank, num_procs, pypar.send, pypar.receiv)
pypar.finalize()
```

To the reader, it should now be clear how to parallelize other independent function calls by the described function-centric approach. To conclude this section, let us show another version of the parallel code, this time with mpi4py as the MPI module (which has a more comprehensive functionality than pypar):

```python
from Parabola import Parabola
from function_centric import parallel_solve_problem
from mpi4py import MPI

problem = Parabola(m=100, n=50, L=10)
comm = MPI.COMM_WORLD
my_rank = comm.Get_rank()
num_procs = comm.Get_size()

def mpi4py_send(data, to_rank):
    comm.send(data, dest=to_rank)

def mpi4py_recv(from_rank):
    return comm.recv(source=from_rank)

parallel_solve_problem(problem.initialize, problem.func, problem.finalize,
                        my_rank, num_procs, mpi4py_send, mpi4py_recv)
MPI.Finalize()
```

By comparison with the pypar-enabled parallel version, the above program has to include two help functions for communication: mpi4py_send and mpi4py_recv. The reason for not directly using mpi4py’s own send and recv functions is because the destination and source processor ranks are required in the mpi4py module as keyword arguments, not as positional arguments in the pypar module. However, the reader should see that the straightforward use of the generic parallel_solve_problem function remains unchanged.

3. Function-centric parallelization

We have shown how to parallelize a serial program that is decomposable into three parts: initialize, calls to func (i.e. a set of independent tasks) and finalize. In this section, we describe how the
function-centric parallelization is helpful for three important classes of scientific applications: Markov chain Monte Carlo (MCMC) simulations, dynamic population Monte Carlo simulations and solution of partial differential equations (PDEs). Python can be used to program a set of simple and generic parallelization functions.

3.1. Parallel Markov chain Monte Carlo simulations

The standard MCMC algorithms are embarrassingly parallel and have exactly the same algorithmic structure as the example of parameter analysis in section 2. This means that the functions initialize, func and finalize can easily be adapted to Monte Carlo problems. More specifically, the initialize function prepares the set of random samples and other input parameters. Some parametric model is computed by the func function, whereas finalize collects the data returned from all the func calls and prepares for further statistical analysis.

Function-centric parallelization of MCMC applications closely follows the example in section 2. We can reuse the three generic functions named get_subproblem_input_args, collect_subproblem_output_args and parallel_solve_problem, assuming that all the func evaluations are equally costly and all the processors are equally powerful so there is no need for more sophisticated load balancing.

In section 4.2, we will look at a real-life Markov chain problem from political science (appendix A gives its mathematical description).

3.2. Population Monte Carlo with dynamic load balancing

A more advanced branch of Monte Carlo algorithms is population Monte Carlo, see [26]. Here, a group of walkers, also called the population, is used to represent a high-dimensional vector and the computation is carried out by a random walk in the state space. During the computation some of these walkers may be duplicated or deleted according to some acceptance/rejection criteria, i.e. the population is dynamic in time. Population Monte Carlo algorithms have been proven useful in a number of fields, spanning from polymer science to statistical sciences, statistical physics and quantum physics.

Unlike the examples given so far, where the computational tasks were totally independent and of static size, population Monte Carlo algorithms may be viewed as an iteration in time where we repeatedly do some work on a dynamic population, including moving the walkers of the population and adjusting the population size, which in a parallel context calls for dynamic load balancing.

3.2.1. Serial implementation. A serial implementation of the time integration function can be as follows:

```python
def time_integration(initialize, do_timestep, finalize):
    walkers, timesteps = initialize()
    output = []
    for step in range(timesteps):
        output.append(do_timestep(walkers))
        walkers.finalize_timestep(output)
    finalize(output)
```

The input arguments to the generic time_integration function are three functions: initialize, do_timestep and finalize. This resembles the three-step structure discussed in section 2. The do_timestep function can have a unified implementation for all the variants of population Monte Carlo algorithms. The other two input functions are typically programmed as methods of a class that implements a particular algorithm (such as diffusion Monte Carlo (DMC) in section 4.3). Here, the initialize method sets up a population object walkers (to be explained below) and determines the number of time steps the walkers are to be propagated. The finalize method can, e.g., store the output for later analysis.

The purpose of the do_timestep function is to implement the work for one time step, including propagating the walkers and adjusting the population. An implementation that is applicable for all population
Monte Carlo algorithms may have the following form:

```python
def do_timestep(walkers):
    walkers.move()
    for walker in range(len(walkers)):
        if walkers.get_marker(walker) == 0:
            walkers.delete(walker)
        elif walkers.get_marker(walker) > 1:
            walkers.append(walker, walkers.get_marker(walker)-1)
    return walkers.sample_observables()
```

The above implementation of time integration and do_timestep assumes that walkers is an object of a class, say with the name Walkers, that has a certain number of methods. Of course, the flexibility of Python allows that the concrete implementation of class Walkers be made afterwards, unlike C++ and Java that require class Walkers be written before implementing time integration and do_timestep. Here, we expect class Walkers to provide a generic implementation of a group of walkers, with supporting methods for manipulating the population. The most important methods of class Walkers are the following:

- **move()** carries out the work of moving each walker of the population randomly according to some rule or distribution function;
- **get_marker(walker)** returns the number of copies belonging to a walker with index walker, where 0 means the walker should be deleted and 2 or more means that clones should be created;
- **append(walker, nchilds)** and **delete(walker)** carry out the actual cloning and removal of a walker with index walker;
- **sample_observables()** returns the observables at a given time step, e.g. an estimate of the system energy;
- **finalize_timestep(old_size, new_size)** does some internal bookkeeping at the end of each time step, such as adjusting some internal variables. It takes as input the total number of walkers before and after the walker population has been adjusted by the do_timestep function;
- **len** is one of Python’s special class methods and is in our case meant to return the number of walkers. A call len(walkers) yields the same result as walkers.__len__().

For a real application, such as the DMC algorithm (see section 4.3 and appendix B), the concrete implementation of the methods should reflect the desired numerical algorithm. For example, the move method of DMC uses diffusion and branching as the rule to randomly move each walker, and the finalize_timestep method adjusts the branching ratio.

### 3.2.2. Parallelization.

Parallelism in population Monte Carlo algorithms arises naturally from dividing the walkers among the processors. Therefore, a parallel version of the time integration function may be as follows:

```python
def parallel_time_integration(initialize, do_timestep, finalize,
                             my_rank, num_procs, send, recv, all_gather):
    my_walkers, timesteps = initialize(my_rank, num_procs)
    old_walkers_len = sum(all_gather(numpy.array([len(my_walkers)])))
    my_output = []
    for step in range(timesteps):
        # do what is required at this time step and measure CPU time
        t_start = time.time()
        results = do_timestep(my_walkers)
        my_output.append(results)
        task_time = time.time() - t_start
```
# redistribute walkers and get walker size per process
num_walkers_per_proc = dynamic_load_balancing(
    my_walkers, task_time, my_rank, num_procs,
    send, recv, all_gather)

# finalize task for this time step
new_walkers_len = sum(num_walkers_per_proc)
my_walkers.finalize_timestep(old_walkers_len, new_walkers_len)
old_walkers_len = new_walkers_len
my_output = collect_subproblem_output_args(my_output, my_rank,
    num_procs, send, recv)
if my_rank == 0:
    finalize(my_output)

In comparison with its serial counterpart, the parallel time integration function has a few noticeable changes. Firstly, the input arguments have been extended with five new arguments. The two integers my_rank and num_procs are, as before, meant for identifying the individual processors and finding the total number of processors. The other three new input arguments are MPI communication wrapper functions: send, recv and all_gather, which can be provided by any of the Python wrapper modules of MPI. The only exception is that pypar does not directly provide the all_gather function, but we can easily program it as follows:

def all_gather (input_array):
    array_gathered_tmp = pypar.gather (input_array, 0)
    array_gathered = pypar.broadcast (array_gathered_tmp, 0)
    return array_gathered

Secondly, we note that the initialize function is slightly different from the serial case, now accepting my_rank and num_procs as input. This is because initial division of the walkers is assumed to be carried out here, giving rise to my_walkers on each processor. Thirdly, a new function dynamic_load_balancing is called during each time step. This function will be explained below in detail. Fourthly, unlike the case where the serial counterpart could simply pass the size of its walkers to finalize_timestep, the parallel implementation needs to collect the global population size before calling finalize_timestep. We remark that each local population knows its own size, but not the global population size. For this purpose, the dynamic_load_balancing function returns the individual local population sizes as a numpy array. Last, the collect_subproblem_output_args function from section 2.2 is used to assemble all the individual results onto the master processor before calling the finalize function.

As mentioned before, parallelization of population Monte Carlo algorithms has to take into account that the total number of walkers changes with time. Dynamic re-distribution of the walkers is therefore needed to avoid work load imbalance. The generic dynamic_load_balancing function is designed for this purpose, where we evaluate the amount of work for each processor and, if the work distribution is too skew, we move the excess walkers from a busy processor to a less busy one. The function first checks the distribution of local population sizes. If the difference between the smallest number of walkers and the largest number of walkers exceeds some predefined threshold, dynamic_load_balancing finds a better population distribution and redistributes the walkers:

def dynamic_load_balancing(walkers, task_time, my_rank, num_procs, 
    send, recv, all_gather):
    walkers_per_proc = all_gather(numpy.array([len(walkers)]))
    if imbalance_rate(walkers_per_proc) > walkers.threshold_factor:
        timing_list = all_gather(numpy.array([task_time]))
        rebalanced_work = find_optimal_workload(timing_list,
            walkers_per_proc)
        walkers = redistribute_work(walkers,
            walkers_per_proc,
            rebalanced_work,
            my_rank, num_procs, send, recv)
    return walkers_per_proc
Two helper functions `find_optimal_workload` and `redistribute_work` are used in the above implementation. Here, `find_optimal_workload` finds the optimal distribution of work, based on how much time each local population has used. The `redistribute_work` function carries out the re-shuffling of walkers. A straightforward (but not optimal) implementation of these functions is as follows:

```python
def find_optimal_workload(timing_list, current_work_per_proc):
    total_work = sum(current_work_per_proc)
    C = total_work/sum(1./timing_list)
    tmp_rebalanced_work = C/timing_list
    rebalanced_work = numpy.array(tmp_rebalanced_work.tolist(),'i')
    remainders = tmp_rebalanced_work-rebalacned_work
    while sum(rebalanced_work) < total_work:
        maxarg = numpy.argmax(remainders)
        rebalanced_work[maxarg] += 1
        remainders[maxarg] = 0
    return rebalanced_work

def redistribute_work(my_walkers, work_per_proc, rebalanced_work, my_rank, num_procs, send, recv):
    difference = work_per_proc-rebalanched_work
    diff_sort = numpy.argsort(difference)
    prev_rank_min = diff_sort[0]
    while sum(abs(difference)) != 0:
        diff_sort = numpy.argsort(difference)
        rank_max = diff_sort[-1]
        rank_min = diff_sort[0]
        if rank_min == prev_rank_min and rank_max != diff_sort[1]:
            rank_min = diff_sort[1]
        if my_rank==rank_max:
            send(my_walkers.cut_slice(rebalaced_work[my_rank]),
                int(rank_min))
        elif my_rank==rank_min:
            my_walkers.paste_slice(recv(int(rank_max)))
        difference[rank_min] += difference[rank_max]
        difference[rank_max] = 0
        prev_rank_min = rank_min
    return my_walkers
```

Careful readers will notice that two particular methods, `my_walkers.cut_slice` and `my_walkers.paste_slices`, provide the capability of migrating the work load between processors in the `redistribute_work` function. These two methods have to be programmed in class `Walkers`, like the other required methods described earlier: `move`, `get_marker`, `append`, `delete` and so on. The `cut_slice` method takes away excess work from a local population and the `paste_slice` method inserts additional work into a local population. Note that the input argument to the `cut_slice` method is an index threshold meaning that local walkers with indices larger than that are to be taken away. The returned slice from `cut_slice` is a picklable Python object that can be sent and received through MPI calls.

The generic `redistribute_work` function deserves a few more words. Among its input arguments is the ideal work distribution, `rebalanced_work`, which is calculated by `find_optimal_workload`. The `redistribute_work` function first calculates the difference between the current distribution, `work_per_proc`, and the ideal distribution. It then iteratively moves walkers from the processor with the most work to the processor with the least work until the difference is evened out.

This load balancing scheme is, in fact, independent of population Monte Carlo algorithms. As long as you have an algorithm repeatedly doing a task over time and where the amount of work in the task varies over time, this scheme can be reused. The only requirement is that an application-specific implementation of class `Walkers`, in terms of method names and functionality, should match with `dynamic_load_balancing` and `redistribute_work`. It should be noted that the given implementation of the latter function is not optimal.
The algorithm of DMC, described in appendix B, is a typical example of a population Monte Carlo algorithm. The implementation is described in section 4.3 and appendix B.

3.3. Parallel additive Schwarz iterations

From the perspective of communication between processors, parallelization of the Monte Carlo algorithms is relatively easy. Parallel MCMC algorithms only require communication at the very beginning and end, whereas parallel population Monte Carlo algorithms require communication at the end of each time step. Actually, our function-centric approach to parallelization can allow more frequent communication. To show the versatility of function-centric parallelization, we apply it to an implicit method for solving PDEs where communication is frequent between processors.

More specifically, many PDEs can be solved by an iterative process called domain decomposition. The idea is to divide the global domain, in which the PDEs are to be solved, into $n$ overlapping subdomains. The PDEs can then be solved in parallel on the $n$ subdomains. However, the correct boundary condition at the internal subdomain boundaries is not known, thus leading to an iterative approach where one applies boundary conditions from the last iteration, solves for the $n$ subdomain problems again and repeats the process until convergence of the subdomain solutions (see e.g. [27, 28]). This algorithm is commonly called additive Schwarz iteration and can successfully be applied to many important classes of PDEs [29–31]. The great advantage of the algorithm, especially from a software point of view, is that the PDE solver for the global problem can be reused for each subdomain problem. Some additional code is needed for communicating the solutions at the internal boundaries between the subdomains. This code can be implemented in a generic fashion in Python, as we explain later.

Let us first explain the additive Schwarz algorithm for solving PDEs in more detail. We consider some stationary PDEs defined on a global domain $\Omega$:

$$
\mathcal{L}(u) = f, \quad x \in \Omega,
$$

subject to some boundary condition involving $u$ and/or its derivatives. Dividing $\Omega$ into a set of overlapping subdomains $\{\Omega_s\}_{s=1}^{P}$, we have the restriction of (1) onto $\Omega_s$, for all $s$, as

$$
\mathcal{L}(u) = f, \quad x \in \Omega_s.
$$

The additive Schwarz method finds the global solution $u$ by an iterative process that generates a series of approximations $u_0, u_1, u_2$ and so on. During iteration $k$, each subdomain computes an improved local solution $u_{s,k}$ by locally solving (2) for $u = u_{s,k}$ with $u_{s,k} = u_{k-1}$ as (an artificial) boundary condition on $\Omega_s$’s non-physical internal boundary that borders with neighboring subdomains. All the subdomains can concurrently carry out the local solution of (2) within iteration $k$, thus giving rise to parallelism. At the end of iteration $k$, neighboring subdomains exchange the latest local solutions in the overlapping regions to (logically) form the global field $u_k$. The subdomain problems (2) are of the same type as the global problem (1), which implies the possibility of reusing an existing serial code that was originally implemented for (1). The additional code for exchange of local solutions among neighbors can be implemented by generic communication operations, independently of specific PDEs.

A generic implementation of the parallel additive Schwarz iteration algorithm can be realized as the following Python function:

```python
def additive_Schwarz_iterations(subdomain_solve, communicate, set_BC, max_iter, threshold, solution, convergence_test=simple_convergence_test):
    iter = 0; not_converged = True
    while not_converged and iter < max_iter:
        iter += 1
        solution_prev = solution.copy()
        set_BC(solution)
```

12
solution = subdomain_solve()
communicate(solution)
not_converged = not convergence_test(solution, solution_prev, threshold)

In the above function, max_iter represents the maximum number of additive Schwarz iterations allowed, and subdomain_solve is a function that solves the subdomain problem of form (2) and returns an object solution, which is typically a numpy array containing the latest subdomain solution \( u_{s,k} \) on a processor (subdomain). However, solution may very well be a more complex object, say holding a collection of scalar fields over computational grids, provided that (i) the object has a copy method, (ii) convergence_test and communicate can work with this object type and (iii) subdomain_solve returns such an object. This flexibility in choosing solution reflects the major dynamic power of Python and provides yet another illustration of the generality of the examples in this paper.

Given an existing serial code, for example in a language like Fortran or C/C++, the subdomain_solve function is easily defined by wrapping up an appropriate piece of the serial code as a Python class (since subdomain_solve does not take any arguments, the function needs a state with data structures, conveniently implemented as class attributes as explained in section 2.1).

The communicate argument is a function for exchanging the latest local solutions among the subdomains. After the call, the solution object is updated with recently computed values from the neighboring subdomains, and the contents of solution have been sent to the neighbors. The communicate function is problem independent and can be provided by some library. In our implementation, the implementation is entirely in Python to take advantage of easy programming of parallel communication in Python. The set_BC argument is a function for setting boundary conditions on a subdomain’s internal boundary. This function depends on the actual serial code and is naturally implemented as part of the class that provides the subdomain_solve function.

The convergence_test function is assumed to perform an appropriate convergence test. The default generic implementation can test

\[
\max_{1 \leq s \leq P} \frac{\|u_{s,k} - u_{s,k-1}\|^2}{\|u_{s,k}\|^2}
\]

against a prescribed threshold value. An implementation reads

```python
def simple_convergence_test(solution, solution_prev, threshold=1E-3):
    diff = solution - solution_prev
    loc_rel_change = vdot(diff, diff) / vdot(solution, solution)
    glob_rel_change = all_reduce(loc_rel_change, MAX)
    return glob_rel_change < threshold
```

We remark that all_reduce should be a wrapper function of the MPI MPI_Allreduce command and vdot computes the inner product of two numpy arrays.

Unlike the three-component structure described in sections 3.1 and 3.2, the main ingredients for parallel additive Schwarz iterations are the functions of subdomain_solve, communicate, set_BC and convergence_test. In other words, it is not natural to divide the work of solving a PDE into initialize, func and finalize. Nevertheless, function-centric parallelization is also here user-friendly and gives a straightforward implementation of additive_Schwarz_iterations as above. The convergence_test function shown above is clearly generic, and so is the communicate function in the sense that it does not depend on the PDE. Both functions can be reused for different PDEs. The other two functions are PDE dependent; however, subdomain_solve normally wraps an existing serial code, while the implementation of set_BC is typically very simple.

4. Applications and numerical experiments

In this section, we will address three real research projects involving the three classes of algorithms covered in section 3. The projects have utilized our function-centric approach to parallelizing existing codes. That is,
we had some software in Fortran, C++ and R performing the basic computations needed in the projects. The serial software was wrapped in Python, adapted to components such as initialize, func, do_timestep, finalize, subdomain_solve, communicate and set_BC. Parallelization was then carried out as explained in previous sections. An important issue to be reported is the parallel efficiency obtained by performing the parallelization in a Python layer that is separate from the underlying serial scientific codes.

The Python enabled parallel codes have been tested on a Linux cluster of 3.4 GHz Itanium2 processors, which are interconnected through 1 Gbits ethernet. The purpose is to show that the function-centric parallelization approach is easy to use and that satisfactory parallel performance is achievable.

4.1. Parallel performance metrics

To help novice readers in the subject of parallel computing, let us quickly define some important metrics for measuring the parallel performance, which will be used in sections 4.2–4.4.

Let $T(p, n)$ denote the time used by a parallel program to solve a problem of size $n$ using $p$ processors. Speedup is defined as

$$s(p, n) = \frac{T(1, n)}{T(p, n)},$$

and the associated parallel efficiency is defined as

$$\eta(p, n) = \frac{s(p, n)}{p}.$$

It should be noted that due to the existence of non-parallelizable code segments and/or parallelization-induced overhead, such as communication cost and partitioning cost, the speedup $s(p, n)$ actually obtained is normally smaller than $p$. The parallel efficiency $\eta(p, n)$ has consequently a normalized value between 0 and 1.

In the above two definitions, the problem size $n$ is fixed. Another metric of performance is scaled speedup, which is defined as

$$s_s(p, n) = \frac{kT(1, n)}{T(p, kn)}.$$

This metric considers the possibility of solving larger problems with more processors. When measured in scaled speedup, good parallel performance is also indicated by a value close to $p$. We refer novice readers to [32].

4.2. Parallel Markov chain Monte Carlo simulations

The first case is from political science and concerns estimating legislators’ ideal points using the MCMC method. For a detailed description of the mathematical problem and the numerical method, we refer the reader to appendix A. This application fits into the setup in section 3.1. The statistical engine is provided by the PSCL library [33] in R [34], for which there exists a Python wrapper.

To use the function-centric parallelization described in section 3.1, we have written a Python class named PIPE. In addition to the constructor of the class (i.e. the __init__ method), there are the following three methods:

- `initialize` sets up the functionality of the PSCL library through the Python wrapper of R (named rpy), and prepares the input argument list needed for `func`;
- `func` carries out the computation of each task by invoking appropriate functions available through rpy (in short, `func` is a Python wrapper to the R function `ideal` from the PSCL library);
- `finalize` summarizes the output and generates an array in R format.
Table 1. Speedup results associated with voting analysis.

<table>
<thead>
<tr>
<th>Legislature</th>
<th>Votes</th>
<th>Members</th>
<th>1 CPU</th>
<th>32 CPUs</th>
<th>Efficiency (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1979–1984</td>
<td>810</td>
<td>548</td>
<td>287 m 32.560 s</td>
<td>10 m 13.318 s</td>
<td>87.91</td>
</tr>
<tr>
<td>1984–1989</td>
<td>1853</td>
<td>637</td>
<td>783 m 59.059 s</td>
<td>26 m 58.702 s</td>
<td>91.06</td>
</tr>
<tr>
<td>1989–1994</td>
<td>2475</td>
<td>597</td>
<td>1006 m 59.258 s</td>
<td>33 m 26.140 s</td>
<td>94.11</td>
</tr>
<tr>
<td>1994–1999</td>
<td>3603</td>
<td>721</td>
<td>1905 m 0.930 s</td>
<td>66 m 0.068 s</td>
<td>90.20</td>
</tr>
<tr>
<td>1999–2004</td>
<td>5639</td>
<td>696</td>
<td>2898 m 45.224 s</td>
<td>102 m 7.786 s</td>
<td>88.70</td>
</tr>
</tbody>
</table>

The resulting parallel Python program is now as short as

```python
from function_centric import parallel_solve_problem
import pypar
my_rank = pypar.rank()
num_procs = pypar.size()

from pypipe import PIPE
problem = PIPE("EP1.RData", "rcvs", "NULL", "NULL")
parallel_solve_problem(problem.initialize, problem.func, problem.finalize,
                      my_rank, num_procs, pypar.send, pypar.receive)
pypar.finalize()
```

The practical importance of a parallel MCMC code is that large and computationally intensive simulations are now easily doable. More specifically, data from the European Parliament between 1979 and 2004 [35] are used for simulation. During the five-year legislative terms, the European Parliament expanded the size of the membership as well as the number of votes taken. (This trend has continued since 2004.) It is hence increasingly computationally intensive to estimate the ideal point model without reducing the length of the Markov chain. We examined the parallel performance by comparing the computing time for each of the five legislatures, running the parallelized code on 32 CPUs. The results are reported in Table 1. When comparing the results, the reader should note that we have not made any attempts to optimize the ideal code (called by our func function) for the purpose of parallelization. This makes it straightforward to switch to new versions of the ideal function. We ran 100 000 MCMC iterations. The parallel efficiency was about 90%.

4.3. Parallel diffusion Monte Carlo simulations

As an example of population Monte Carlo methods, we will now look at parallel DMC computations (see appendix B for a detailed numerical description), which is used here to simulate Bose–Einstein condensation. We recall from section 3.2 that dynamic load balancing is needed in connection with the parallelization, and can be provided by the generic dynamic_load_balancing function. To utilize the parallel time integration function parallel_time_integration from section 3.2, we need to program a parallel version of the initialize function. The do_timestep function from section 3.2 can be used as is.

```python
def initialize(my_rank, num_procs):
    nwalkers = 1000
    nspacedim = 3
    stepsize = 0.1
    timesteps = 2000
    walkers_per_proc = simple_partitioning(nwalkers, num_procs)
    my_walkers = Walkers(walkers_per_proc[my_rank], nspacedim, stepsize)
    my_walkers.threshold_factor = 1.1
    return my_walkers, timesteps
```

This initialize function is quite similar to its serial counterpart. As noted in section 3.2, it takes as input my_rank and num_procs. The simple_partitioning function is called to partition the walker population. A my_walkers object is assigned to each processor, and a threshold factor is prescribed to determine when load balancing is needed.
Table 2. Timing results of the parallel DMC simulations where each processor is constantly assigned with 200 walkers, all moved in 5000 time steps.

<table>
<thead>
<tr>
<th>CPUs</th>
<th>Time</th>
<th>Efficiency (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>37 m 10.389 s</td>
<td>N/A</td>
</tr>
<tr>
<td>5</td>
<td>42 m 32.359 s</td>
<td>87.39</td>
</tr>
<tr>
<td>10</td>
<td>42 m 00.734 s</td>
<td>88.48</td>
</tr>
<tr>
<td>20</td>
<td>42 m 29.945 s</td>
<td>87.47</td>
</tr>
<tr>
<td>30</td>
<td>42 m 33.895 s</td>
<td>87.33</td>
</tr>
<tr>
<td>40</td>
<td>43 m 30.092 s</td>
<td>85.45</td>
</tr>
<tr>
<td>50</td>
<td>43 m 39.159 s</td>
<td>85.16</td>
</tr>
</tbody>
</table>

Together with the parallel_time_integration function from section 3.2, the above initialize function is the minimum programming effort needed to parallelize a serial population Monte Carlo code. For the particular case of our parallel DMC implementation, we also need to know the global number of walkers in every time step to be able to estimate its observables globally. Moreover, the load balancing scheme needs the time usage of each processor during each time step.

A class with the name Walkers needs to be implemented to match with the implementations of parallel_time_integration, dynamic_load_balancing and the above initialize function. The essential work is to provide a set of methods with names already decided (see section 3.2), such as move, append, delete, finalize_timestep, cut_slice and paste_slice. A concrete example of the Walkers class is described with more details in appendix B.

We report in table 2 the timing results of a number of parallel DMC computations. The total number of walkers was increased when more processors were used, such that the number of walkers assigned to each processor remained at 200. Such a use of parallel computers for DMC simulations mimics the everlasting wish of quantum physicists to do larger computations as soon as more computing resource becomes available. Note that in this scaled scalability test, good parallel performance is indicated by an almost constant time usage independent of the number of processors. The parallel efficiency associated with scaled speedup was about 85%.

4.4. Parallel Boussinesq simulations

Simulating the propagation of ocean waves is the target of our third and final concrete case. We refer the reader to appendix C for the mathematical model and the numerical method. The equations involved can be solved in parallel by the additive Schwarz algorithm of section 3.3.

Our starting point for parallelization is a 25 years old legacy Fortran 77 code consisting of a set of subroutines. The most important subroutines of the legacy code are KONTIT and BERIT, which target the two semi-discretized equations (C.3) and (C.4) of the mathematical model (see appendix C). These two Fortran 77 subroutines contain intricate algorithms with nested layers of do-loops, which are difficult to parallelize by directly inserting MPI calls in the Fortran code. Performing the parallelization outside the legacy code is therefore highly desirable. We thus adopted the approach of using parallel additive Schwarz iterations.

The main computational work of a subdomain solver is done by calling KONTIT or BERIT. In addition, a subdomain solver needs to set artificial boundary conditions at non-physical boundaries. When programming the two actual subdomain_solve functions, which are needed for the two equations (C.3) and (C.4), we chose to first implement two light-weight wrapper subroutines in Fortran: subd_continuity and subd_beroulli. This choice was for the sake of computing speed. Since the generic additive_Schwarz_iterations function from section 3.3 requires a callable subdomain_solve function in Python, we then extended the two new Fortran subroutines with Python interface. F2PY software [4] is a very convenient tool for this purpose. As an example, let us look at the syntax of subd_continuity, which is the subdomain-solver wrapper of KONTIT:
% lower_x_neigh, upper_x_neigh, lower_y_neigh, upper_y_neigh
integer ns, ms
real*8 F(0:ns,0:ms), Y(0:ns,0:ms), YW(0:ns,0:ms), H(0:ns,0:ms)
real*8 QY(8,0:*), WRK(*)
real*8 dx, dy, dt, gg, alpha, eps
integer kit, ik, nbit
integer lower_x_neigh, upper_x_neigh, lower_y_neigh, upper_y_neigh
Cf2py intent(in, out) Y, nbit

All the input arguments (except the four last arguments) to subd_continuity are needed by the legacy KONTIT subroutine, which is called inside subd_continuity. The computed results from KONTIT are passed back in Y and nbit. The above code line that starts with Cf2py will be ignored by any standard Fortran compiler, but is important to F2PY software. It specifies Y and nbit as both input and output arguments when extending subd_continuity with a Python interface. More specifically, suppose that the source code of subd_continuity and subd_bernoulli is contained in the file new_subroutines.f. The following F2PY command will build a Python extension module with the name f77.so:

f2py -m f77 -c new_subroutines.f legacy.o

We remark that legacy.o is a pre-compiled object file containing the legacy KONTIT and BERIT subroutines. The extension module f77.so allows the new subd_continuity and subd_bernoulli subroutines to be called in a Python program as follows:

import f77
f77.subd_continuity(...)  
f77.subd_bernoulli(...)  

Finally, in order to create two Python subdomain_solve functions that can be used inside the generic additive_Schwarz_iterations function, we created a class where many Fortran input variables are stored as class attributes:

import f77 # extension module for the new Fortran subroutines

class SubdomainSolver:
    def __init__(self, ...):
        # set input arguments to the Fortran subroutines as class attributes
        # (nbit,F,Y,YW,H,QY,WRK,dx,dy,dt,kit,ik,gg,alpha,eps,
        # lower_x_neigh,upper_x_neigh,lower_y_neigh,upper_y_neigh)

        def continuity(self):
            self.Y, self.nbit = f77.subd_continuity(
                self.dx, self.dy, self.dt, self.kit, self.ik,
                self.gg, self.alpha, self.eps, self.nbit,
                self.lower_x_neigh, self.upper_x_neigh,
                self.lower_y_neigh, self.upper_y_neigh)

        def bernoulli(self):
            # similar to the continuity method

    Note that the two methods continuity and bernoulli can be called without any arguments, which match the design of additive_Schwarz_iterations.

    The main computation of the resulting parallel program is done in the following while-loop:

    sd = SubdomainSolver(...)  
    
    t = 0
    while t < t_stop:
        t = t+dt
Table 3. The speedup results of the Python enabled parallel Boussinesq simulations.

<table>
<thead>
<tr>
<th>CPUs</th>
<th>Time (s)</th>
<th>Speedup</th>
<th>Efficiency (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>166.66</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>83.61</td>
<td>1.99</td>
<td>99.67</td>
</tr>
<tr>
<td>4</td>
<td>44.45</td>
<td>3.75</td>
<td>93.73</td>
</tr>
<tr>
<td>8</td>
<td>20.16</td>
<td>8.27</td>
<td>103.33</td>
</tr>
<tr>
<td>16</td>
<td>11.43</td>
<td>14.58</td>
<td>91.13</td>
</tr>
</tbody>
</table>

The `additive_Schwarz_iterations` function from section 3.3 can be placed in a reusable module. The `communicate` function is borrowed from a Python library for mesh partitioning and inter-subdomain communication. The `set_BC` function actually does nothing for this particular application.

Speedup results are reported in table 3, for which the global solution mesh was fixed at $1000 \times 1000$, and the number of time steps was 40. The results show that we can handle quite a complicated mathematical problem by a legacy Fortran code (trivially extended) and obtain reasonably good speedup.

5. Conclusion

We have shown how serial scientific codes written in various common languages, including Fortran, C, C++, and Python, can be parallelized in a separate, small software unit written in Python. The advantage of such an approach is twofold. Firstly, the existing, often complicated, scientific high-performance code remains (almost) unchanged. Secondly, the parallel algorithm and its inter-processor communication are conveniently implemented in high-level Python code.

This approach to parallelization is function centric, where the programmer needs to implement a few Python functions for carrying out the key steps in the solution approach. For example, our first application involves doing a set of independent tasks in parallel, where a small Python framework deals with the parallelism and demands the user to supply three functions: `initialize` for preparing input data to the mathematical model, `func` for calling up the serial scientific code and `finalize` for processing the computational results. Some more functions must be supplied in more complicated problems where the algorithm evolves in time, with a need for dynamic load balancing and more parallel communication.

Our simple parallelization approach outlined in this paper is applicable to many different scientific areas, and we have described some common classes of problem: parameter investigation of a mathematical model, standard Monte Carlo simulation, Monte Carlo simulation with a need for dynamic load balancing and numerical solution of PDEs. In each of these cases, we have outlined fairly detailed Python code such that most technical details of the parallel implementations are documented. This may ease the migration of the ideas to new classes of problem beyond the scope of this paper.

In particular, our function-centric approach has been used to parallelize three real scientific problems taken from our research. The problems concern MCMC models for voting behavior in political science, DMC methods for Bose–Einstein condensation in quantum mechanics and simulating ocean waves by a system of PDEs. The results of our investigations of the parallel efficiency are very encouraging: in all these real science problems, almost optimal speedup results are obtained, showing that there arises no significant loss due to using Python and performing the parallelization ‘outside’ the serial codes.

As a conclusion, we believe that the ideas and code samples from this paper can simplify parallelization of serial codes greatly, without significant loss of computational efficiency. This is good news for scientists who are non-experts in parallel programming but want to parallelize their serial codes with as little effort as possible.
Appendix A. Voting in legislatures

In the spatial model of politics, both actors’ preferences over policies (ideal points) and policy alternatives are arranged geometrically in a low-dimensional Euclidean space. An actor receives the highest possible utility if a policy is located at her ideal point; she gains or loses utility as the policy moves towards or away from her ideal point [36]. We adopt the Bayesian approach proposed by Clinton et al [37]. Assume that there are \( n \) legislators who vote on \( m \) proposals. On each vote \( j = 1, \ldots, m \), legislator \( i = 1, \ldots, n \) chooses between a ‘Yea’ position \( \xi_j \) and a ‘Nay’ position \( \psi_j \) located in the policy space \( \mathbb{R}^d \), where \( d \) is the number of dimensions. Then, we have \( y_{ij} = 1 \) if legislator \( i \) votes ‘Yea’ on roll call \( j \) and \( y_{ij} = 0 \) if she votes ‘Nay’. The model assumes quadratic utility functions. The ideal point of legislator \( i \) is \( x_i \in \mathbb{R} \), while \( \eta_{ij} \) and \( \psi_{ij} \) are stochastic elements whose distribution is jointly normal. The variance of the stochastic elements is \( (\eta_{ij} - \psi_{ij}) = \sigma_j^2 \).

Denote the Euclidean norm by \( \| \cdot \| \); utility maximizing implies that legislator \( i \) votes ‘Yea’ on vote \( j \) if

\[
U_i(\xi_j) = -\|x_i - \xi_j\|^2 + \eta_{ij} > U_i(\psi_j) = -\|x_i - \psi_j\|^2 + \psi_{ij}
\]

and ‘Nay’ otherwise. Clinton et al [37] show that the model can be understood as a hierarchical probit model:

\[
P(y_{ij} = 1) = \Phi(\beta_j^* x_i - \alpha_j)
\]

where \( \beta_j = 2(\xi_j - \psi_j)/\sigma_j, \alpha_j = (\xi_j^* - \psi_j^*)/\sigma_j, \Phi(\cdot) \) is the standard normal function, \( \beta_j \) is the midpoint between the ‘Yea’ and ‘Nay’ positions on proposal \( j \) and \( x_i \) is the legislator’s ideal point. The direction of \( \alpha_j \) indicates the location of the status quo relative to the proposal. If \( \alpha_j \) is positive, the new proposal is located higher on the dimension than the status quo. If \( \alpha_j \) is negative, the new proposal is located lower on the dimension than the status quo.

A.1. Markov chain Monte Carlo algorithm

In the MCMC algorithm for the statistical analysis of voting behavior [37], the difference between utilities of the alternatives on the \( j \)th vote for the \( i \)th legislator is given by \( y_{ij}^* = \beta_j x_i - \alpha_j + \epsilon_{ij} \), where \( \beta_j \) and \( \alpha_j \) are model parameters, \( x_i \) is a vector of regression coefficients and \( \epsilon_{ij} \) are standard normal errors. If we know \( \beta_j \) and \( \alpha_j \), \( x_i \) can be imputed from the regression of \( y_{ij}^* + \alpha_j \) on \( \beta_j \) using the \( m \) votes of legislator \( i \) and vice versa. If we know \( x_i \), we can use the votes of the \( n \) legislators on roll call \( j \) to find \( \beta_j \) and \( \alpha_j \). Given \( x_i, \beta_j \) and \( \alpha_j \) (either from priors or from the previous iteration), we can find \( y_{ij}^* \) by drawing \( \epsilon_{ij} \) randomly from a normal distribution subject to the constraints implied by the actual votes, i.e. if \( y_{ij} = 0, y_{ij}^* < 0 \) and if \( y_{ij} = 1, y_{ij}^* > 0 \).

The goal is to compute the joint posterior density for all model parameters \( \beta_j, \) and \( \alpha_j, j = 1, \ldots, m \) and all coefficient vectors \( x_i, i = 1, \ldots, n \). The MCMC algorithm forms a Markov chain to explore as much as possible of this joint density, i.e. letting \( t \) index an MCMC iteration,

(i) find \( y_{ij}^{t, *}, \beta_j^{t-1}, \) and \( \alpha_j^{t-1} \),
(ii) sample \( \beta_j^t \) and \( \alpha_j^t \) using \( x_i^{t-1} \) and \( y_{ij}^{t, *}, \)
(iii) find \( \xi_j^t \) from \( \beta_j^t, \alpha_j^t \) and \( y_{ij}^{t, *} \).

This process must then be repeated until convergence, i.e. that the samples have moved away from the priors to the neighborhood of the posterior mode before samples are drawn.

Clinton et al [37, p 369] find that the computation time is increasing in \( nmT \), where \( n \) is the number of legislators, \( m \) is the number of votes and \( T \) is the number of MCMC iterations. Although they argue that very long runs are normally not necessary, they nevertheless advise long runs to ensure that the MCMC algorithm has converged. It is increasingly time-consuming to estimate the model on a standard desktop computer as the size of the legislature and the number of votes increase.
Appendix B. Bose–Einstein condensation

The famous experiment of Anderson et al [38] was about cooling $4 \times 10^6 ^{87}\text{Rb}$ down to temperatures of the order of 100 nK for observing Bose–Einstein condensation in the dilute gas. To model this fascinating experiment in the framework of quantum Monte Carlo, so that numerical simulations can be extended beyond the physical experiments, we may use the governing Schrödinger equation

$$i\hbar \frac{\partial}{\partial t} \Psi(R, t) = H \Psi(R, t). \quad \text{(B.1)}$$

The most important parts of the mathematical model are a Hamiltonian $H$ and a wave function $\Psi$, see [39]. The Hamiltonian for $N$ trapped interacting atoms is given by

$$H = -\frac{\hbar^2}{2m} \sum_{i=1}^{N} \nabla_i^2 + \sum_{i=1}^{N} V_{\text{ext}}(r_i) + \sum_{i<j}^{N} V_{\text{int}}(|r_i - r_j|). \quad \text{(B.2)}$$

The external potential $V_{\text{ext}}$ corresponds to the trap used to confine the $^{87}\text{Rb}$ atoms, and was in the experiment of the order of $r^2$. The two-body interaction $V_{\text{int}}(|r_i - r_j|)$ can be easily described by a hard-core potential of radius $a$ in a dilute gas. We have, however, for the sake of simplicity, neglected these interactions in our example implementation of class Walkers.

B.1. Diffusion Monte Carlo method

In the DMC method [40], the Schrödinger equation is solved in imaginary time,

$$-\frac{\partial}{\partial t} \psi(R, t) = [H - E] \psi(R, t). \quad \text{(B.3)}$$

The formal solution of (B.3) is

$$\psi(R, t) = e^{-\int[H - E]dt} \psi(R, 0), \quad \text{(B.4)}$$

where $e^{-\int[H - E]dt}$ is called Green’s function, and $E$ is a convenient energy shift.

The wave function $\psi(R, t)$ in DMC is represented by a set of random vectors $\{R_1, R_2, \ldots, R_M\}$, in such a form that the time evolution of the wave function is actually represented by the evolution of a set of walkers. This feature gives rise to task parallelism. The wave function is positive definite everywhere, as it happens with the ground state of a bosonic system, so it may be considered as a probability distribution function.

The DMC method involves Monte Carlo integration of Green’s function by every walker. The time evolution is done in small time steps $\tau$, using the following approximate form of Green’s function:

$$e^{-\int[H - E]\tau} = \prod_{i=1}^{n} e^{-\int[H - E]\tau}, \quad \text{(B.5)}$$

where $\tau = t/n$. Assuming that an arbitrary starting state can be expanded in the basis of stationary,

$$\psi(R, 0) = \sum_{\nu} C_{\nu} \phi_{\nu}(R), \quad \text{(B.6)}$$

we have

$$\psi(R, t) = \sum_{\nu} e^{-\int[E_{\nu} - E]\tau} C_{\nu} \phi_{\nu}(R), \quad \text{(B.7)}$$

in such a way that the lowest energy components will have the largest amplitudes after a long elapsed time, and in the limit of $t \to \infty$ the most important amplitude will correspond to the ground state (if $C_0 \neq 0$).\(^6\)

\(^6\) This can be easily seen by replacing $E$ with the ground state energy $E_0$ in (B.7). As $E_0$ is the lowest energy, we will obtain

$$\lim_{t \to \infty} \sum_{\nu} \exp[-(E_{\nu} - E_0)t] \phi_{\nu} = C_0 \phi_0.$$
Green’s function is approximated by splitting it up into a diffusional part,

\[ G_D = (4\pi D \tau)^{-3N/2} \exp\{-\frac{(R' - R)^2}{4D\tau}\}, \]  

which has the form of a Gaussian, and a branching part,

\[ G_B = \exp\{-\frac{((V(R) + V(R'))/2 - E_T)\tau}{2}\}. \]  

While diffusion is taken care of by a Gaussian random distribution, the branching is simulated by creation and destruction of walkers with a probability \( G_B \). The idea of DMC computation is quite simple; once we have found an appropriate approximation of the short-time Green’s function and determined a starting state, the computation consists of representing the starting state by a collection of walkers and letting them independently evolve in time. That is, we keep updating the walker population, until a large enough time when all the states other than the ground state are negligible.

**Algorithm 1.** Diffusion Monte Carlo

```python
for step in range(0, timesteps):
    for i in range(0, Nwalkers):
        Diffusion;
        propose move \( R' = R + \xi \)
        Branching;
        calculate replication factor \( n \):
        \[ n = \text{int}(\exp\{-((V(R) + V(R'))/2 - E_T)\tau\}) \]
        if \( n = 0 \):
            mark walker as dying
        if \( n > 0 \):
            mark walker to make \( n - 1 \) clones
        Remove dead walkers and make new clones;
        Update walker population \( N_{walkers} \) and adjust trial energy;
        Sample contributions to observable.
```

**B.2. Implementation**

In algorithm 1, we summarize the DMC algorithm corresponding to (B.8) and (B.9). In the algorithm \( \xi \) is a Gaussian with zero mean and a variance of \( 2D\tau \) corresponding to (B.8). The deleting and cloning of walkers are, as mentioned in section 3.2, performed by the do_timestep function, repeated here for clarity:

```python
def do_timestep(walkers):
    walkers.move()
    for walker in range(len(walkers)):
        if walkers.get_marker(walker) == 0:
            walkers.delete(walker)
        elif walkers.get_marker(walker) > 1:
            walkers.append(walker, walkers.get_marker(walker)-1)
    return walkers.sample_observables()
```

The main computational work of the DMC algorithm at each time step is implemented in the move function inside class Walkers, together with a helper function branching:

```python
class Walkers:
    ...

    def branching(self, new_positions):
        old_potential = potential(self.positions)
        new_potential = potential(new_positions)
```
branch = numpy.exp(-(0.5 * (old_potential + new_potential) - self.adjust_branching) * self.stepsize)
self.markers = numpy.array(branch +
numpy.random.uniform(0,1,branch.shape), 'i')

def move(self):
displacements = numpy.random.normal(0, 2*self.stepsize,
self.positions.shape)
new_positions = self.positions + displacements
self.branching(new_positions)
self.positions = new_positions

The move function first generates a set of Gaussian (normal) distributed random numbers, corresponding to (B.8). Next, it calls the branching function, which calculates a potential $V(r) = r^2$ for the old and the new positions. These potentials are used to calculate $G_B$ following (B.9) and create an integer array self.markers with its average value equal to $G_B$ (stored in the branch variable). This array is of the same length as the number of walkers (stored in self.positions) and marks the walkers as dying or clonable.

It is worth noting that if the new potential of a walker is much higher than that in the previous time step (i.e. the walker is far from the center of the trap), the value of branch will be close to 0 and the walker will be deleted. However, if the new potential is much lower (i.e. closer to the center of the trap), branch will be greater than 1 and the walker will be cloned. As long as the two-body interaction is ignored, the walkers will only be encouraged to move towards the center of the trap, thus yielding a lower energy than seen in real experiments.

**Appendix C. Ocean wave propagation**

The following two PDEs, normally termed as the Boussinesq water wave equations [41], can be used to model wave propagation:

\[
\frac{\partial \eta}{\partial t} + \nabla \cdot \left( (H + \alpha \eta) \nabla \phi + \epsilon H \left( \frac{1}{6} \frac{\partial \eta}{\partial t} - \frac{1}{3} \nabla H \cdot \nabla \phi \right) \right) \nabla H = 0, \tag{C.1}
\]

\[
\frac{\partial \phi}{\partial t} + \frac{\alpha}{2} \nabla \phi \cdot \nabla \phi + \eta - \frac{\epsilon}{2} H \nabla \cdot \left( H \nabla \frac{\partial \phi}{\partial t} \right) + \frac{\epsilon}{6} H^2 \nabla^2 \frac{\partial \phi}{\partial t} = 0. \tag{C.2}
\]

The primary unknowns of (C.1) and (C.2) are the water surface elevation $\eta(x, y, t)$ and the depth-averaged velocity potential $\phi(x, y, t)$. The symbol $H$ denotes the water depth as a function of $(x, y)$. The advantage of the above Boussinesq wave model, in comparison with the standard shallow water equations, is its capability of modeling waves that are weakly dispersive ($\epsilon > 0$) and/or weakly nonlinear ($\alpha > 0$), see [42]. Therefore, the Boussinesq water wave equations are particularly adequate for simulating ocean wave propagation over long distances and large water depths.

Discretization of the Boussinesq water wave equations (C.1) and (C.2) normally starts with a temporal discretization as follows:

\[
\frac{\eta^\ell - \eta^{\ell-1}}{\Delta t} + \nabla \cdot \left( \left( H + \alpha \frac{\eta^{\ell-1} + \eta^\ell}{2} \right) \nabla \phi^{\ell-1} + \epsilon H \left( \frac{1}{6} \frac{\eta^\ell - \eta^{\ell-1}}{\Delta t} - \frac{1}{3} \nabla H \cdot \nabla \phi^{\ell-1} \right) \right) \nabla H = 0, \tag{C.3}
\]

\[
\frac{\phi^\ell - \phi^{\ell-1}}{\Delta t} + \frac{\alpha}{2} \nabla \phi^{\ell-1} \cdot \nabla \phi^{\ell-1} + \eta^\ell - \frac{\epsilon}{2} H \nabla \cdot \left( H \nabla \left( \frac{\phi^\ell - \phi^{\ell-1}}{\Delta t} \right) \right) + \frac{\epsilon}{6} H^2 \nabla^2 \left( \frac{\phi^\ell - \phi^{\ell-1}}{\Delta t} \right) = 0. \tag{C.4}
\]

In a more optimized implementation, the old potential would have been stored from the previous move and not calculated every time.
where we use $\ell$ to denote the time level, and $\Delta t$ denotes the time step size. The basic idea of computation at each time step is to first compute $\eta^\ell$ based on $\eta^{\ell-1}$ and $\phi^{\ell-1}$ from the previous time step and then compute $\phi^\ell$ using the new $\eta^\ell$ and the old $\phi^{\ell-1}$. To carry out the actual numerical computation, we need a spatial discretization of (C.3) and (C.4), using, e.g. finite differences or finite elements, so we end up with two systems of linear equations that need to be solved during each time step.

References

[18] Jones E et al 2008 pyMPI: Putting the py in MPI pypm.sourceforge.net/
[33] Jackman S 2006 PSCL: classes and methods for R developed in the Political Science Computational Laboratory, Stanford University Technical Report Department of Political Science, Standford University