Using BSP and Python to Simplify Parallel Programming

Konrad Hinsen,*
Hans Petter Langtangen‡
Ola Skavhaug‡
Åsmund Ødegård§

Abstract

Scientific computing is usually associated with compiled languages for maximum efficiency. However, in a typical application program, only a small part of the code is time–critical and requires the efficiency of a compiled language. It is often advantageous to use interpreted high–level languages for the remaining tasks, adopting a mixed–language approach. This will be demonstrated for Python, an interpreted object–oriented high–level language that is well suited for scientific computing. Particular attention is paid to high–level parallel programming using Python and the BSP model. We explain the basics of BSP and how it differs from other parallel programming tools like MPI. Thereafter we present an application of Python and BSP for solving a partial differential equation from computational science, utilizing high–level design of libraries and mixed–language (Python–C or Python–Fortran) programming.

1 Introduction

Scientific computing has some specific requirements that influence the choice of programming tools. The most outstanding property of scientific computing is its explorative nature: although some standard methods are used over and over again, they are used in different combinations every time, and often it is necessary to add custom algorithms and programs to a collection of well–established standard code. Although the literature on scientific computing may leave the impression that all that matters are efficient number crunching and visualization methods, the day–to–day work of a computational scientist involves a lot of interfacing, file format conversion, book–keeping, and similar tasks, often made

*Centre de Biophysique Moléculaire (UPR 4301 CNRS) Rue Charles Sadron, 45071 Orléans Cedex 2, France, hinsen@cnrs-orleans.fr.
‡Simula Research Laboratory (SRL), P.O.Box 134, 1325 Lysaker, Norway, hpl@simula.no.
§SRL, skavhaug@simula.no.
§SRL, asmund@simula.no.
difficult by bad user interface design and lack of documentation. These lengthy and unattractive tasks often discourage scientists to pursue a potentially interesting idea. Good programming tools can thus make a significant contribution to good computational science.

High–level languages can help in several ways. At the simplest level, they can be used to write all tools that are not time critical, such as simple analysis programs, file format converters, etc. As a general rule, high–level languages are better suited for I/O- and text–oriented tasks than the standard low–level programming languages used in scientific computing: Fortran, C, and C++. However, as this paper will show, they can be useful in number–crunching applications as well, making the programs easier to develop and use. The key to these applications is mixed–language programming, i.e., combining a high–level and a low–level language in order to get the best of both worlds.

To avoid misunderstandings, an explanation of the term “high–level” is in order. Most of all, it implies no judgment of quality. High–level languages are by definition those whose constructs and data types are close to natural–language specifications of algorithms, as opposed to low–level languages, whose constructs and data types reflect the hardware level. With high–level languages, the emphasis is on development convenience, whereas low-level languages are designed to facilitate the generation of efficient code by a compiler. Characteristic features of high-level languages are interactivity, dynamic data structures, automatic memory management, clear error messages, convenient file handling, libraries for common data management tasks, support for the rapid development of graphical user interfaces, etc. These features reduce the development and testing time significantly, but also incur a larger runtime overhead leading to longer execution times.

We remark that what is called “high–level” in this paper is often referred to as “very high level”; different authors use different scales. Many people also use the term “scripting languages”.

The high–level language used as an example in this paper is Python [7], a language that is becoming increasingly popular in the scientific community. Although other suitable languages exist and the choice always involves personal preferences, Python has some unique features that make it particularly attractive: a clean syntax, a simple yet powerful object model, a flexible interface to compiled languages, automatic interface generators for C/C++ and Fortran, and a large library of reusable code, both general and scientific. Of particular importance is Numerical Python [6], a library that implements fast array operations and associated numerical operations. Many numerical algorithms can be expressed in terms of array operations and implemented very efficiently using Numerical Python. Moreover, Numerical Python arrays are used at the interface between Python and low–level languages, because their internal data layout is exactly the same as a C array. We will in this paper adopt the widely used term NumPy as a short form of Numerical Python.

The outline of this paper is as follows. Section 2 discusses two different programming styles for utilizing high–level languages for scientific computations. We then turn to the topic of high–level parallel computation, using the BSP
model, in Section 3. This simple model of parallel computing, in combination with Python, enables communication of high-level data types, and eliminates the risk of deadlocks often encountered in other parallel computing tools like MPI and PVM. Section 4 describes how to implement a parallel partial differential equation simulator, using Python and BSP. In this section, we also show how Python can be extended by migrating time-critical functions to tailor-made C and Fortran code. We then run some numerical benchmarks against similar Matlab and C code to quantify the loss of efficiency by using Python and BSP for simplified, high-level parallel computing. The final section summarizes the findings and states some concluding remarks.

2 High-Level Language Programming Styles

There are basically two ways of utilizing a high-level language like Python in scientific computing. Either we equip an external application or library with a Python interface, or we create a new application from scratch in Python and migrate time-critical operations to Fortran or C. These two approaches are described next.

2.1 Python Interfaces to Existing Numerical Codes

A typical situation in computational science is the following: An existing program contains all the relevant methods, but its user interface is cumbersome, I/O facilities not sufficient, and interfacing with other programs could be easier. Another common case is the existence of a library of computational algorithms which is used by relatively simple application programs that are constantly modified. In this case, modification and testing of the applications often take a significant amount of time.

A good solution in both situations is to control the application through a high-level language. This means that one can call functionality in the application directly from Python. Often the existing functionality is mixed with additional functionality provided by Python modules, e.g., for I/O, file handling, report generation, etc. In this way we use Python to glue different software components. (Some refer to this gluing as scripting; others use scripting for programming in high-level languages in general.) For a user, the application and additional functionality are called either from small Python programs or from an interactive Python shell. The interactivity allows for what is called computational steering in the context of scientific computing; the application can be started and stopped interactively, and the user can modify parameters, “rewind” simulations, restart, stop again, visualize data, and so on. This is useful both for explorative scientific computing and for debugging applications. A Python interface to an existing application is easily equipped with a GUI, which may represent another way of enhancing the original user interface of the application.

In case of an existing monolithic application, the first step would be to isolate
the computational parts of the code and turn them into a (highly specialized)
library; much of the user interface and I/O code would be discarded. This
library then has to be provided with a Python interface. In most cases this task
can be accomplished by an automatic interface generator such as the Simplified

An attractive feature of Python interfaces to existing applications is that
one usually needs just some basic Python knowledge to take advantage of the
interface. The Python syntax is close to Matlab’s syntax, i.e., intuitive and easy
to learn. In a sense, the Python interface offers a tailored Matlab-like environ-
ment for running the original application. Although Matlab could be interfaced
with the application, the generation of the interface is easier with Python and
tools like SWIG or F2py. Moreover, the Python language is more powerful than
the Matlab language. In other words, Python interfaces to existing codes ex-
tend the Matlab idea of a user-friendly working environment to new and more
sophisticated problem areas.

We should also mention that an existing application could embed Python,
I.e., call Python to perform certain tasks that are more conveniently performed
in a high–level language than in the application’s original low–level language.

Python interfaces to numerical codes are easy to realize, because existing
code can be used without extensive modifications and the interface can be more
or less automatically generated. However, the benefit of the interfaces is mostly
limited to the users of the applications; numerical code developers work much
like before, as the majority of the code is still written in a low–level language.
The design of the low–level code, especially its data structures, determines the
design of the interface layer and the use of the interface. A completely different
approach to numerical computing is advocated in the next section.

2.2 High–Level Design

The complementary approach to using Python for steering an existing applica-
tion is to design the whole application for the high–level language, switching to
low–level code only for specific time–critical parts. The roles of the two lan-
guages are thus inversed, the low–level code is written specifically to fit into
the high–level design. The developer can profit from all of the advantages of
high–level languages to reduce development and testing time, and – assuming a
good programming style – the code becomes more compact and more readable.
However, this approach makes it less straightforward to integrate existing low–
level code, unless it takes the form of a (generic) library with a well–designed
interface. High–level design also requires a good knowledge of Python and
object–oriented techniques.

It must be emphasized that the result of this approach is quite different
from gluing components of existing applications and providing a high–level in-
terface. In the course of time, a computational scientist can build up a library
of problem–specific code, written by himself or obtained from others, that uses
the same scientific concepts and abstractions as natural language: numbers,
vectors, functions, operators, atoms, molecules, flow fields, differential equation
solvers, graphical representations, etc. In low–level code, everything would have
to be expressed in terms of numbers and arrays plus functions working on these
data.

To give a simple example, suppose you have a large compressed text file
containing some numbers per line and you want to plot a histogram of the
numbers in the second column. In Python this can be written as follows:

```python
from Scientific.IO.TextFile import TextFile
from Scientific.Statistics.Histogram import Histogram
from Gnuplot import plot

data = []
for line in TextFile('data.gz):
data.append(float(line.split()[1]))
plot(Histogram(data, 100))
```

The class `TextFile` presents a simple abstraction of a text file to the user: a
sequence of lines that can be iterated over. Internally it handles many details: it
can deal with standard as well as compressed files, and it can also accept
URLs instead of filenames, in which case it automatically downloads the file
from a remote server, stores it temporarily, and deletes the local copy when it
has been read completely. The user does not need to know how any of this is
accomplished, for him a text file always remains just a sequence of lines. The
`TextFile` module itself is implemented in just a few lines of code, demonstrat-
ing that the high–level nature of Python allows creating quite sophisticated
functionality without much programming.

The class `Histogram` provides a similar abstraction for histograms. The
only required input are the data and the number of bins. Of course the classes
`TextFile` and `Histogram` must be written first, but only once by one person,
they can then be used by anyone anywhere, even interactively, without the need
to know how they work internally.

It is often said that object–oriented low–level languages, such as C++, can
be used in the same way. However, the higher implementation effort is often
discouraging, and impatient scientists frequently settle for the simplest solution
that will do the job at hand, even if that means minor reuse of the code in
the next project. Moreover, the code would have to be recompiled for each
application, whereas the Python code can be used interactively in a calculator–
style. This gives a faster edit–and–run procedure. A problem–specific library in
Python can in this way also be used as a “numerical workbench” for explorative
(interactive) computing. Hence, for a user, high–level design and interfaces to
existing codes give many of the same advantages; the differences are inside the
Python layer and in the numerical code.

According to our experience, code reuse works better in Python than in
low–level languages, whose strict typing rules make it difficult to design suffi-
ciently flexible interfaces. With the exception of libraries designed by expert
programmers with the explicit goal of generality (e.g. LAPACK), scientific code
in low–level languages is almost never directly reusable. In Python, reusability
is easier to achieve, and the weak type compatibility rules, combined with independent name spaces for modules, ensure that even libraries designed completely independently work well together.

3 Parallel Computing Using BSP

As an example of the use of high-level design in a traditional heavy-duty computing field, this section shows how Python can be used to facilitate the development of parallel programs.

Most textbooks on parallel computing focus on algorithmic aspects, mentioning implementation issues only in passing. However, the implementation of parallel algorithms is far from trivial, since a real-life application uses several different algorithms and requires a significant amount of bookkeeping and I/O. Although many computational scientists envisage parallelization at some time, few ever get beyond simple test programs, because development and debugging become too cumbersome.

A major reason for the difficulty of parallel programming is the low-level nature of the most popular parallel communications library, the Message Passing Interface (MPI). MPI has a large number of functions that permit the optimization of many communication patterns, but it lacks easy-to-use high-level abstractions. Most importantly, it places the responsibility for synchronization fully on the programmer, who spends a lot of time analyzing deadlocks. Moreover, MPI does not offer much support for transferring complex data structures.

A simpler and more convenient parallel programming model is the Bulk Synchronous Parallel (BSP) model [3], [11]. In this model, computation and communication steps alternate, and each communication step involves a synchronization of all processes, effectively making deadlocks impossible. Another advantage of bundling communication in a special step is the possibility for an underlying communications library to optimize data exchange for a given machine, e.g. by combining messages sent to the same process. The analysis of algorithms is also facilitated, making it possible to predict the performance of a given algorithm on a given parallel machine based on only three empirical parameters. The Python implementation of BSP (which is part of the Scientific Python package [9]) adds the possibility of exchanging almost arbitrary Python objects between processes, thus providing a true high-level approach to parallelization.

Although the BSP model is a convenient parallel programming model for many problems, the more advanced features of MPI are missing. Several extensions of the original BSP model have been developed that add many of the missing features. The Python BSP module is built on the BSPlib implementation, and thus, the recent extensions to BSP do not apply in this context. When using BSPlib, grouping of processes, overlapping communication, and computation are not possible. Using one of the many primitives of MPI one can perform optimizations of an application that are out of the question using BSPlib.
3.1 Overview

An important difference for readers familiar with MPI programming is that a Python BSP program should be read as a program for a parallel machine made up of \( N \) processes and not as a program for one process that communicates with \( N - 1 \) others. A Python BSP program has two levels, local (any one process) and global (all processes), whereas a typical message-passing program uses only the local level. In message-passing programs, communication is specified in terms of local send and receive operations. In a BSP program, communication operations are synchronized and global, i.e., all processes participate.

The two levels are reflected by the existence of two kinds of objects: local and global objects. Local objects are standard Python objects; they exist in a single process. Global objects exist on the parallel machine as a whole. They have a local value on each process, which may or may not be the same everywhere. For example, a global object “process id” would have a local value equal to the respective process number. Global objects also often represent data sets of which each process stores a part, the local value is then the part of the data that one process is responsible for.

The same distinction applies to functions. Standard Python functions are local functions: their arguments are local objects, and their return values are local objects as well. Global functions take global objects as arguments and return global objects. A global function is defined by one or more local functions that act on the local values of the global objects. In most cases, the local function is the same on all processes, but it is also common to have a different function on one process, usually number 0, e.g. for I/O operations. It is possible to transfer global objects as arguments to local functions. In this case, it is not possible to access the local values of the global objects directly. However, calls to the methods of the global objects are possible.

Finally, classes can be local or global as well. Standard Python classes are local classes, their instances are local objects, and their methods act like local functions. Global classes define global objects, and their methods act like global functions. A global class is defined in terms of a local class that describes its local values.

Communication operations are defined as methods on global objects. An immediate consequence is that no communication is possible within local functions or methods of local classes, in accordance with the basic principle of the BSP model: local computation and communication occur in alternating phases. It is, however, possible to implement global classes that are not simply global versions of some local class, and that can use communication operations within their methods. They are typically used to implement distributed data structures with non-trivial communication requirements. The design and implementation of such classes requires more care, but they allow the complete encapsulation of both the calculation and the communication steps, making them very easy to use. An example within the Python BSP package is a class that represents distributed netCDF files, and which ensures automatically that each process handles a roughly equal share of the total data. From a user’s point of view,
this class has a programming interface almost identical to that of the standard Python netCDF module. Another example is given in Section 4, where a PDE simulator is developed.

3.2 Standard Global Classes

The simplest and most frequent global objects are those which simply mirror the functionality of their local values and add communication operations. They are represented by the classes ParConstant, ParData, and ParSequence, all of which are subclasses of ParValue. The three classes differ in how their local representations are specified.

ParConstant defines a constant, i.e. its local representation is the same on all processes. Example:

```python
zero = ParConstant(0)
```

has a local representation of 0 on all processes.

ParData defines the local representation as a function of the process number and the total number of processes. Example:

```python
def getPid(pid, nprocs):
    return pid
pid = ParData(getPid)
```

has an integer (the process number) as local representation.

ParSequence distributes its argument, which must be a Python sequence (list, tuple, dict, etc.), over the processes as evenly as possible. For example,

```python
integers = ParSequence(range(10))
```

divides the ten integers among the processes. With two processes, number 0 receives the local representation [0, 1, 2, 3, 4] and number 1 receives [5, 6, 7, 8, 9]. With three processes, number 0 receives [0, 1, 2, 3], number 1 receives [4, 5, 6, 7], and number 2 receives [8, 9]. The reason for this particular choice of the possible divisions among the processes, is that the smallest integer larger than the average is used as chunk-size. Then all processes but the last get this number of elements, the last process gets the remaining elements.

All these classes define the standard arithmetic operations, which are thus automatically parallelized. They also support indexing and attribute extraction transparently. For example, adding two NumPy vectors in parallel can be coded like this:

```python
x = Numeric.arange(0, 1.0, 0.1, Numeric.Float)
v1 = Numeric.sin(x); v2 = Numeric.cos(x)
vlp = ParSequence(v1); v2p = ParSequence(v2)
sum = vlp + v2p
```

Global functions are created using the class ParFunction when the local representation is the same local function on all processes (the most common case). Another frequent case is to have a different function on process 0, e.g. for I/O operations. This is arranged by the class ParRootFunction.
Numerical array computing in Python should always be based on NumPy arrays, which are more effective than plain Python arrays or lists. NumPy arrays are created and manipulated by functions in the Numeric module. For efficiency, the programming style must be vector-oriented, much like that in Matlab. That is, explicit loops over NumPy arrays must be avoided in Python code. Instead one needs to express the computational algorithm as a set of vector operations, or if this is inconvenient, the loop over NumPy arrays should be implemented in C. We explain the latter strategy in Sections 4.2 and 4.5.

3.3 A Simple Example

The first example illustrates how to deal with the simplest common case: some computation has to be repeated on different input values, and all the computations are independent. The input values are thus distributed among the processes, each process calculates its share, and in the end all the results are communicated to one process that takes care of output. In the following example, the input values are the first 100 integers, and the computation consists of squaring them.

```python
from Scientific.BSP import ParSequence, ParFunction, \n    ParRootFunction
import operator

# The local computation function:
def square(numbers):
    s = []
    for n in numbers:
        s.append(n*n)
    return s

# The global computation function:
global_square = ParFunction(square)

# The local output function:
def output(result):
    print result

# The global output function - active on process 0 only:
global_output = ParRootFunction(output)

# A list of numbers distributed over the processes:
items = ParSequence(range(100))

# Computation:
results = global_square(items)

# Collect results on process 0:
all_results = results.reduce(operator.add, [])

# Output from process 0:
global_output(all_results)
```

The local computation function is a straightforward Python function: it takes a list of numbers, and returns a list of results. The call to ParFunction then generates a corresponding global function. ParSequence takes care of distributing the input items over the processes, and the call to global_square does all the computation. Before process 0 can output the results, it has to collect them
from all other processes. This is handled by the method `reduce`, which works much like the Python function of the same name, except that it performs the reduction over all processes instead of over the elements of a sequence, analogous to the reduce method of MPI. The arguments to `reduce` are the reduction operation (addition in this case) and the initial value, which is an empty list here because we are adding up lists.

This program works correctly independently of the number of processes it is run with, which can even be higher than the number of input values. However, the program is not necessarily efficient for any number of processes, and the result is not necessarily the same, as the order in which the local result lists are added up by the reduction operation is not specified. If an identical order is required, the processes have to send their process ID along with the data, and the receiving process must sort the incoming data according to process ID before performing the reduction.

One possibly critical aspect of this program is that each process needs to store all the input data for the whole computation. When working with big data objects, it might not be feasible to have each process store more than the data for one iteration at the same time. This case can be handled with synchronized iterations, as will be shown in Section 3.5.

### 3.4 Systolic Algorithms

The next example presents another frequent situation in parallel programming, a systolic algorithm. It is used when some computation has to be done between all possible pairs of data items distributed over the processes. In the example, a list of items (letters) is distributed over the processes, and the computational task is to find all pairs of letters (in a real application, a more complex computation would of course be required).

The principle of a systolic algorithm is simple: each data chunk is passed from one process to the next, until after \( N - 1 \) iterations each process has seen all data, \( N \) is the number of processes. The new features that are illustrated by this example are general communication and accumulation of data in a loop.

```python
from Scientific.BSP import ParData, ParSequence, 
ParAccumulator, ParFunction, ParRootFunction, 
numberOfProcessors

import operator

# Local and global computation functions:
def makepairs(sequence1, sequence2):
    pairs = []
    for item1 in sequence1:
        for item2 in sequence2:
            pairs.append((item1, item2))
    return pairs
global_makepairs = ParFunction(makepairs)

# Local and global output functions:
def output(result):
    print result
global_output = ParRootFunction(output)
```
# A list of data items distributed over the processes:
my_items = ParSequence('abcdef')

# The number of the neighbor to the right (circular):
neighbor_pid = ParData(lambda pid, nprocs: [(pid+1)%nprocs])

# Loop to construct all pairs:
pairs = ParAccumulator(operator.add, [])
pairs.addValue(global_makepairs(my_items, my_items))
other_items = my_items
for i in range(numberOfProcessors-1):
    other_items = other_items.put(neighbor_pid)[0]
    pairs.addValue(global_makepairs(my_items, other_items))

# Collect results on process 0:
all_pairs = pairs.calculateTotal()

# Output results from process 0:
global_output(all_pairs)

The essential communication step is in the line

    other_items = other_items.put(neighbor_pid)[0]

The method put is the most basic communication operation. It takes a list of destination processes (a global object) as its argument; in this example, that list contains exactly one element, the process number of the successor. Each process sends its local representation to all the destination processes.

In the example, each process receives exactly one data object, which is extracted from the list by a standard indexing operation. The result of the line quoted above thus is the replacement of the local value of other_items by the local value that was stored in the preceding process. After repeating this $N - 1$ times, each process has seen all the data.

It is instructive to analyze how the systolic loop would be implemented using the popular MPI library in a low-level language. First, either the “items” have to be represented by arrays, or appropriate MPI data types need to be defined, whereas Python allows almost any object to be transmitted in a message. Then each process must send its own data and receive the data from its neighbor. If standard send and receive calls are used, the programmer must take care not to use the same order (send/receive or receive/send) on all processes, as this creates a risk of deadlock. MPI provides a special combined send+receive operation for such cases, whereas BSP eliminates deadlocks altogether. With MPI, the programmer must also allocate a sufficiently large receive buffer, which implies knowing the size of the incoming data. All this bookkeeping overhead would easily exceed the code size of the whole Python program, the programmer would have to take care that it doesn’t cause run–time overhead when executing on a single process, and even more effort would be required to make the program work serially without MPI being present. In contrast, Python BSP programs automatically work serially with a standard Python installation, and in most cases with negligible performance overhead.
3.5 Back to Trivial Parallelism

We come back to the situation that was treated in the first example: many independent calculations whose results are combined in the end. The solution that was presented above is not always sufficient. The input data that is distributed over the processes might come from a file or from previous computations, and might thus need to be distributed explicitly by one process to the others. Also, it might not be possible to do all the computations locally and collect the results in the end, e.g. due to memory restrictions if the results have a large number of data items. If the results must be written to some file immediately, and if only process 0 has access to I/O, then the original solution cannot easily be adapted, as no communication is possible during the local computations.

The following example works in a different way. First, process 0 reads the input numbers from a file, and distributes them explicitly to the other processes. Second, the local computation (still just squaring, for simplicity) works on one number at a time. A parallelized loop is set up which at each iteration processes one number per process and then sends the result back to process 0 for output. (This can be necessary if the local computation generates a lot of data, which makes global storage of all the results not feasible.)

```python
from Scientific.BSP import ParFunction, ParRootFunction, ParMessages, 
                      ParConstant, ParIterator, numberOfProcessors

import operator, string

# The local and global input functions.
def input():
    data = open('numbers').readlines()
    numbers = map(string.atol, map(string.strip, data))
    chunk_size = (len(numbers)+numberOfProcessors-1)/numberOfProcessors
    chunks = []
    for i in range(numberOfProcessors):
        chunks.append((i, numbers[i*chunk_size:(i+1)*chunk_size]))
    return chunks

def empty():
    return []

global_input = ParRootFunction(input, empty)

# The local and global computation functions.
def square(x):
    return x**2

global_square = ParFunction(square)

# The local and global output functions.
def output(results):
    file = open('results', 'a')
    for value in results:
        file.write('value' + '\n')
    file.close()

global_output = ParRootFunction(output)

# Read input data.
data = global_input()

# Distribute input data.
items = ParMessages(data).exchange().extractData()

# Computation and output loop.
for item in ParIterator(items):
    result = global_square(item)
```
collected_results = result.put(ParConstant([0])).data()
global_output(collected_results)

The first new feature is that ParRootFunction is called with two arguments, the second one being the local function that is called on processes other than number 0. The default, which has been used until now, is a function that returns None. The input function reads a list of numbers from a file and splits them into chunks for each process. The result of the global input function is a list of (pid, numbers) pairs on process 0 and an empty list elsewhere.

The computation function is different here because it expects a single number instead of a list. The output function is also different in that it appends results to a file instead of printing them to standard output.

The distribution step is particularly interesting. The communication operations we have seen so far do not permit to send different data to each process in a single operation. This can be achieved by explicitly constructing a ParMessages object, whose local values are lists of (pid, data) pairs, which are sent using the exchange method.

Finally, we see a parallelized loop using a ParIterator. Inside the loop, the global object item contains one number from items at each iteration. After the computation, the results are immediately sent to process 0 and written to the output file.

In general, the number of data items handled by each process will not be exactly equal. Some processes might thus have nothing to do during the last iteration of the loop. This is handled internally by flagging the local value of item as invalid. Invalid values are ignored by all subsequent operations, which all yield invalid values as well. In this way, the application programmer does not have to take special precautions for this case.

The last loop can also be written in a slightly different way:

```python
for index in ParIndexIterator(items):
    result = global_square(items[index])
    collected_results = result.put(ParConstant([0])).data()
    global_output(collected_results)
```

The ParIndexIterator returns a parallel index object instead of returning the sequence elements directly. It is mainly used for looping over several distributed sequences (of equal length) at the same time.

Like in the previous example, the order of the results of this program depends on the number of processes. If this presents a problem in practice, it must be solved by using a special storage format.

4 Partial Differential Equations

Solving partial differential equations (PDEs) is considered a task where low-level compiled languages are required in order to obtain sufficient efficiency. However, PDE simulators consist of more than number crunching. System initialization, file handling and visualization, among others, are tasks that may
be cumbersome in low–level languages. Thus, the concept of high–level design, explained in Section 2.2, can be interesting to explore for applications involving PDEs. Because speed is important in PDE applications, it is crucial that the time–critical parts of the computation take place in pre–compiled low–level libraries. Having BSP as well, we can utilize the power of a parallel architecture to speed up the simulation even further. For the remaining tasks, we make use of the high–level nature of Python.

As a model problem, we consider a PDE that comes from mathematical finance; the Black–Scholes equation. A simple one–dimensional version of this equation is addressed to keep the mathematical and numerical details at a low yet relevant level and enable focus on the principal implementational aspects. The building blocks we develop for this particular problem are relevant and reusable in many other PDE problems.

The process of solving PDEs numerically consists of two steps. First, the equations must be discretized using a suitable strategy, e.g. finite difference, volume, or element methods. Second, the resulting, often linear, algebraic problem needs to be solved. Either iterative or direct methods must be used for this task, and the focus here will be on iterative methods.

This section is outlined as follows. We start by discretizing the model problem, using a finite difference method. This leads to an algebraic problem that requires a certain data structure. The implementation of this data structure is the topic of the following section. Since our model problem is one–dimensional, we need distributed tri–diagonal matrix and vector classes. The time–critical parts of the operations needed are carried out in tailor–made C or Fortran 77 code in order to enhance efficiency.

Describing in detail how to build a general, parallel, state–of–the–art linear algebra library accessible from Python and utilizing BSP is clearly beyond the scope of this text. However, by showing how this can be done in a simple case, we provide a starting point for writing extensions needed to solve other, more realistic problems.

Having the tri–diagonal data–structure at hand, we describe how to implement a simple iterative method for solving a linear system of algebraic equations. It is well known that there exist faster solvers, like cyclic reduction and tri–diagonal Gaussian elimination, for tri–diagonal linear systems of equations. These methods are not trivial to parallelize and may be hard to generalize to higher–dimensional problems. Instead, we implement the iterative Jacobi method. This method involves matrix–vector multiplication and vector operations only, and thus mimics the basic properties of more advanced solution strategies like Krylov subspace solvers. Also, the Jacobi method, or flavors of this method, is often used as smoother in multilevel methods. Although the Gauss–Seidel or Successive Over Relaxation (SOR) methods usually perform better than Jacobi iteration and are equally simple to implement, we prefer Jacobi iteration because it is easier to parallelize.

Finally, we apply the framework to the discretized model problem. We compare the run time of the simulator to similar Matlab and C codes, and investigate the speed-up gained by running the simulator on multiple processors.
using a shared memory parallel machine.

4.1 Model Problem

Our model problem involves the Black–Scholes equation, which was derived by Black and Scholes in a celebrated work from 1973 [4] (later, this work was rewarded the Nobel price in economics). This equation, subject to specific final and boundary conditions, models the risk–free price of options (see [12] for an introduction to mathematical models for option pricing). In the simplest case, the so-called European vanilla options, a backward parabolic PDE problem of the following form has to be solved.

\[ \frac{\partial P}{\partial t} + \frac{1}{2} \sigma^2 S^2 \frac{\partial^2 P}{\partial S^2} + rS \frac{\partial P}{\partial S} - rP = 0, \quad (1) \]

\[ P(S, T) = \max(E - S, 0), \]

\[ P(0, t) = E e^{-r(T-t)}, \]

\[ P(S, t) = 0 \quad \text{as} \quad S \to \infty. \]

This models the price \( P(S, t) \) of a European Put option, which gives the holder of the contract the right to sell the asset of value \( S \) for the price \( E \) at the future time \( T \). An asset may be some stock or commodity and \( S \) is then the asset’s market price. The asset may also be an index, in which case \( S \) is simply the value of the index. The other parameters in the model are \( \sigma \), the volatility of the asset and \( r \), the risk free interest rate. The point is to compute the price \( P \) backwards in time, and in particular, compute the present value \( P(S, 0) \).

We discretize this problem using a finite difference method in both time and space\(^1\). We approximate the time derivative at time \( t_k \) by the backward difference

\[ \frac{\partial}{\partial t} P(S_i, t_k) \approx \frac{P^{k+1}_i - P^k_i}{\Delta t}, \]

where \( \Delta t = \frac{T}{N+1} \) is the time step size, giving the total number of \( N \) inner time steps. The first-order spatial derivative is approximated by an upwind scheme,

\[ \frac{\partial}{\partial S} P(S_i, t_k) \approx \frac{P^{k+1}_{i+1} - P^k_i}{\Delta S}, \]

where the step size is \( \Delta S = \frac{s_{\infty}}{M+1} \). Here \( s_{\infty} \) is the length of the truncated solution domain, and \( M \) is the number of interior mesh points. A centered scheme is used for the the second–order term,

\[ \frac{\partial^2}{\partial S^2} P(S_i, t_k) \approx \frac{P^{k+1}_{i+1} - 2P^k_i + P^{k-1}_i}{(\Delta S)^2}. \]

We remark that “upwind” and “backward” in this backward PDE problem correspond to “downwind” and “forward” in a forward PDE problem.

\(^1\)We use the term “space” for the asset dimension.
The complete scheme takes the form
\[ \frac{P_{k+1}^i - P_k^i}{\Delta t} + \frac{1}{2} \sigma^2 S_i \frac{P_{k+1}^i - 2P_k^i + P_{k-1}^i}{(\Delta S)^2} + rS_i \frac{P_{k+1}^i - P_k^i}{\Delta S} - rP_k^i = 0, \]
i = 1, \ldots, M. Since we solve this problem backwards in time, this is an implicit scheme with unknowns \( P_k^i \), \( i = 1, \ldots, M \). The system is linear, and can be written as
\[ Au^k = b^k, \quad k = N, N - 1, \ldots, 0, \]
where \( A \) is a tri-diagonal matrix in \( \mathcal{R}^{(M+2) \times (M+2)} \) and \( u, b \) are vectors of length \( M + 2 \). We have Dirichlet boundary conditions at both \( S = 0 \) and at \( S_\infty \). These are enforced into the system by setting,
\[ A_{0,0} = 1, \quad A_{0,1} = 0, \]
\[ b_0^k = E e^{-r(T-k\Delta t)}, \]
for the left boundary and,
\[ A_{M+1,M+1} = 1, \quad A_{M+1,M} = 0, \]
\[ b_{M+1}^k = 0, \]
for the right boundary. At the interior nodes, \( b_j = u_j^{n+1} \). The matrix \( A \) has the entries,
\[ A_{j,j-1} = l_{j-1} = -\alpha_j, \]
\[ A_{j,j} = d_j = 1 + r\Delta t + 2\alpha_j + \beta_j, \]
\[ A_{j,j+1} = u_{j+1} = -\alpha_j - \beta_j, \]
for \( j = 1, \ldots, M \), where,
\[ \alpha_j = \frac{1}{2} \frac{\Delta t}{(\Delta S)^2} S_j^2 \sigma^2, \]
\[ \beta_j = r \frac{\Delta t}{\Delta S} S_j. \]

This model problem is quite simple. However, if more underlying assets are involved in the option contract, each asset increases the dimension of the problem by one. When the number of assets is large, Monte Carlo simulations are usually carried out in order to determine the option price numerically. When the number of assets is low, i.e. up to three, finite difference, element and volume methods are used as well.

From an implementation point of view, the main difference between a multi-dimensional and a one-dimensional problem is that the coefficient matrix in the linear system of equations is no longer tri-diagonal, but sparse with a few
nonzero bands. The other components of the simulator stay more or less unchanged.

Figure 1 displays the computed scaled present value $P(S, 0)$ of an option, given the future conditions at $t = T$ and the backward evolution equation (1). The interpretation of the graph is the present risk-neutral option price, with short time to expiry. The value is lesser than the pay–off at expiry for small values of $S$. This is expected for a European put option, since we cannot exercise the option before the predetermined expiry date. Hence, the current option price represents the discounted future pay–off (if any), see [12].

![Solution of the Black-Scholes equation (European Put)](image)

Figure 1: The solution at time $t = 0$. The module Gnuplot is used to generate this plot and save it to file.

### 4.2 Tri–diagonal Matrix Computations

When using Python to solve PDEs, the main goal is to develop data types that can be used in a high–level fashion. For instance, a matrix–vector product in parallel should, after initialization, be as simple as $A \cdot b$, where $A$ is the matrix and $b$ is the vector. Below we will develop data types offering this functionality.

We need a distributed vector, DistVector, and a distributed tri–diagonal matrix, DistTriDiagMatrix to store the linear system of equations at each time step in the model problem above. Both classes are equipped with methods for addition and multiplication as well as other useful methods of various kinds.

Consider first the vector class. The standard way of implementing a new distributed class using scientific.BSP is to sub–class ParBase. This class implement basic methods for concurrent computing, and sub–classing this ensure that new distributed classes share a least common set of functionality for
concurrency. Therefore, `DistVector` is also a sub-class of this class:

```python
class DistVector(ParBase):
    # Global vector init
    def __parinit__(self, pid, nprocs, vec):
        chunk_size = (vec.shape[0]+nprocs-1) / nprocs
        self.cs = chunk_size # cs is used later, avoid recalculation.
        self.pid = pid
        self.nprocs=nprocs
        self.v = vec[pid*chunk_size:(pid+1)*chunk_size] # Local data
```

When a global class is instantiated, the special method `__parinit__` is used for object initialization (i.e. constructor in the parallel case). The process ID, `pid`, and the total number of processes used during the execution of the program, `nprocs`, are always passed to the method before the rest of the argument list. In this case, the “rest” is only the full vector to be distributed among the available processes. Additional ways to initialize the distributed vector are possible, in order to avoid creating the complete vector before it is distributed to the different processes. A similar problem is treated in Section 3.5, where one process distributes local values to the other processes. Because of the general way arguments are passed on to functions in Python, other sets of parameters could be passed to the `__parinit__` method, e.g. as keyword arguments.

Different parts of the global vector must be stored locally on each process. The `__parinit__` method above computes the length of the local storage, `self.cs`, and then extracts the (global) values into the local NumPy array, called `self.v`.

When used as a local object, the usual `__init__` method (constructor) is called when the instance is created, and the whole vector provided as argument is attached.

```python
# Local vector init
def __init__(self, vec, pid=0, nprocs=0, cs=0):
    self.v = vec
    self.pid = pid
    self.nprocs=nprocs
    self.cs=cs
```

Some of the methods of the `DistVector` class return new vectors. In this case, the local class is used to create the new local vector, and the usual `__init__` method is used for initialization. If the object returning the new vector is global, several new local objects are created, and they collectively define a new global object.

We need the plus, `+`, operator for adding vectors. This is done by overloading the special method `__add__`.

```python
# Vector-vector addition
def __add__(self, other):
    if len(self.v) == len(other.v):
        return DistVector(self.v+other.v, self.pid, self.nprocs, self.cs)
    else:
        raise ValueError, "Incompatible lengths"
```
When two global vector objects are added together, this method computes the local contribution to the global sum and returns the result as a distributed vector. Thus, $c = a + b$, where $a$ and $b$ are distributed vectors, creates the new distributed vector $c$.

The subtraction operator is implemented similarly:

```python
# Vector-vector subtraction
def __sub__(self, other):
    if len(self.v) == len(other.v):
        return DistVector(self.v-other.v, self.pid, self.nprocs, self.cs)
    else:
        raise ValueError, "Incompatible lengths"
```

We want the `__mul__` method between two vectors to return the vector resulting from component-wise multiplication. By doing so, we can use the vector to store a diagonal matrix, eliminating the need for additional data types in applications involving diagonal matrices.

```python
# Vector-vector multiplication
def __mul__(self, other):
    if len(self.v) == len(other.v):
        return DistVector(self.v*other.v, self.pid, self.nprocs, self.cs)
    else:
        raise ValueError, "Incompatible lengths"
```

Note that the above methods return new `DistVectors`. Creating new objects takes time, and we try to avoid doing so whenever possible. By implementing the incremental addition, subtraction and multiplication through the operators `+=`, `-=`, and `*=` we can manipulate vectors “in-place”. This is important since we avoid creating large temporary NumPy arrays. Here is an example of incremental add (`+=`), where we just operate directly on the object’s vector (`self.v`):

```python
def __iadd__(self, other):
    if len(self.v) == len(other.v):
        self.v = other.v
    else:
        raise ValueError, "Incompatible lengths"
```

When adding, subtracting, and multiplying distributed vectors, communication between the local parts of the global vector is not needed. However, there are several relevant operations on vectors that require communication between processes, e.g., norms and inner products. The `norm` method computes the local contribution to the global norm, sends this contribution to all processes and adds the contributions to form the global norm (discrete $l_2$ norm).

```python
def norm(self):
    if self.nprocs==0: # One process only
        return sqrt(dot(self.v, self.v)) # Fast computation of global norm
    loc = dot(self.v, self.v) # Compute squared norm of local vector
    messages = []
    for i in range(self.nprocs):
        messages.append((i, loc)) # Prepare to send squared local norm
    data = self.exchangeMessages(messages) # Send/receive squared local norms
    sum = reduce(add, data, 0) # Add squared local norm contributions
    return sqrt(sum) # Return the global norm
```
The call `self.exchangeMessages(messages)` is a convenient interface to the
low–level general communication functions. The data `messages` is a list of
`(pid, data)`, and all data are sent to the destination processes. The return
value is a list of incoming data.

We now consider the `DistTriDiagMatrix` class for distributed tri–diagonal
matrices. This class stores the sub, super, and main diagonal of the matrix as
vectors.

```python
class DistTriDiagMatrix(ParBase):
    # Global matrix init
    def __parinit__(self, pid, nprocs, 1, d, u):
        self.num_entries = d.shape[0]  # dimension
        self.type = "DistTriDiagMatrix"
        self.pid = pid
        self.nprocs = nprocs
        chunk_size = (d.shape[0]+self.nprocs-1)/self.nprocs
        self.cs=chunk_size
        self.l = array([0])
        self.d = d[pid*chunk_size:(pid+1)*chunk_size]  # Lower diagonal
        self.u = u[chunk_size*pid:(pid+1)*chunk_size]  # Diagonal
        if pid == nprocs-1:
            self.l = concatenate((self.l, z), 1)
        if pid == 0:
            self.u = concatenate((z, u[0:chunk_size-1]), 1)  # Upper diagonal
        else:
            self.u = u[chunk_size*pid-1:(pid+1)*chunk_size-1]
        self._prod = zeros(len(self.d)+2, Float)  # matrix–vector prod
```

The `__parinit__` method is somewhat more complicated in this case. In
addition to the mandatory arguments `pid` and `nprocs` described above, the
argument list consists of the sub diagonal, `l`, main diagonal, `d`, and super diagonal,
`u`. This method defines the type of the object, stores the local process number,
and extracts the local contribution to the global object. The local storages
of the matrix are chunks of columns. The number of columns in each chunk is
determined by the formula

\[
cs = \frac{n + p - 1}{p},
\]

where `cs` is the chunk size, `n` is the number of columns in the global matrix,
and `p` is the number of processes. The matrix class' `__mul__` method implements
the matrix–vector product, which is the most time–critical component of the
library. (Note that the matrix resulting from the corresponding matrix–matrix
multiplication is not tri–diagonal, and thus not supported by our library.)

First we list the code, and thereafter we explain what is going on in detail.

```python
# Matrix–vector multiplication
def __mul__(self, other):
    if len(other.v) != len(self.d):  # Wrong use.
        raise ValueError, "incompatible lengths"
    pid = self.pid; nprocs=self.nprocs; prod = self._prod  # For efficiency
    # Compute the local matrix vector product
    # tridiagprod is a tailored function implemented in C
    tridiagprod(self.l, self.d, self.u, other.v, prod)

    # The size of prod is len(self.d)+2. The additional entries must be
```
# communicated to the neighboring procs in order to compute the product correctly

messages = []
if pid > 0:  # Send to pid-1.
    messages.append((pid-1, (pid, prod[0])))
if pid < nprocs-1:  # Send to pid+1
    messages.append((pid+1, (pid, prod[-1])))

data = self.exchangeMessages(messages)

messages={}  # We need to know that pid sent the message, and use a hash for this.
if nprocs > 1:
    messages[data[0][0]]=data[0][1]
if (pid >0) & (pid < nprocs-1):
    messages[data[1][0]]=data[1][1]
if pid > 0:
    prod[1] += messages[pid-1]
if pid < nprocs-1:
    prod[-2] +=messages[pid+1]

return DistVector(prod[1:-1], pid, nprocs, self.cs)

The BSP model requires the programmer to divide the parallel application into so-called super–steps [8]. Each super–step is divided into three parts: local computation, communication, and an immediate barrier synchronization of all the processes. All communication within a super–step is guaranteed to occur by the end of the super–step, such that the communicated information is available on the receiver at the beginning of the next super–step. In the __mul__ method above, this structure is easily recognized. First, we compute the local contribution and prepare to send the data that the other processes need. Then the communication and barrier synchronization take place in the method self.exchangeMessages(message). Finally, the local values are updated and the method returns the result.

Before discussing the actual implementation of the matrix–vector product, the numerical method itself needs some words of explanation. Assume that l, d and u are the sub, main and super–diagonal of the matrix A, respectively. We want to compute \( v = Ax \). We can extend A with two zero rows, one at the top and one at the bottom. This gives a modified system \( \tilde{v} = \tilde{A}x \):

\[
\begin{bmatrix}
v_0 \\
v_0 \\
\vdots \\
v_{n+1} \\
v_{n+1}
\end{bmatrix}
= 
\begin{bmatrix}
0 & 0 & 0 & \cdots \\
d_0 & u_1 & 0 & \cdots \\
l_0 & d_1 & u_2 & \cdots \\
0 & l_1 & d_2 & \cdots \\
& & & & l_{n-1} & d_n & u_{n+1} \\
& & & & l_n & d_{n+1} & 0 & 0
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
\vdots \\
x_{n+1}
\end{bmatrix}
\]

(2)

The modification results in two extra entries in the product \( \tilde{v} \), \( v_0 \) and \( v_{n+1} \).
The matrix-vector product of the original system is given as a slice of \( \tilde{v} \), \( (v_0, \ldots, v_{n+1}) \).
This extension of the system is of course pointless in the scalar case, but proves useful in a parallel algorithm. Instead of computing the matrix-vector product the usual way,
\[ v_i = l_{i-1} x_{i-1} + d_i x_i + u_{i+1} x_{i+1}, \]
in a loop over the elements of the product vector \( \mathbf{v} \), we compute,
\[
\begin{align*}
v_{i-1} & = u_i x_i, \\
v_i & = d_i x_i, \\
v_{i+1} & = l_i x_i,
\end{align*}
\]
that is a loop over the elements of the vector \( \mathbf{x} \).

Consider the parallel case, and for simplicity, with two processes available. The values of \( \mathbf{v} \) and \( \mathbf{x} \) are distributed evenly among the two processes of the form \( \mathbf{v}_0 = \{ v_i : i = 0, 1, \ldots, n/2 \} \), \( \mathbf{v}_1 = \{ v_i : i = n/2 + 1, \ldots, n + 1 \} \). The matrix \( \mathbf{A} \) is extended and split vertically as shown below:

\[
\begin{bmatrix}
v_0 & v_0 \\
v_{n/2} & v_{n/2}\star \\
v_{n/2+1} & v_{n/2+1}\star \\
v_{n+1} & v_{n+1}\star \\
\end{bmatrix}
= \begin{bmatrix}
0 & \cdots & u_1 \\
d_0 & d_1 & \cdots \\
l_0 & l_1 & \cdots & u_{n/2} \\
0 & l_1 & \cdots & d_{n/2} \\
0 & l_1 & \cdots & l_{n/2+1} \\
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
x_{n/2} \\
x_{n/2+1} \\
x_n \\
x_{n+1} \\
\end{bmatrix}
\]

Due to the extension of the original matrix, the boundaries of the original matrix do not need special treatment during the local computations. We simply multiply the left part of the matrix with the first half of the vector \( \mathbf{x} \) on the first process, and multiply the right part of the matrix with the second half of the vector on the second process. During this, local, part of the computation, there is no need for communication between the processes. However, the products on both processes consist of two additional entries in both ends of the local product vector; the extension described earlier and a contribution to the local product stored on the other process, \( \mathbf{v}_0 = (v_0, v_0, v_{n/2}\star) \) and \( \mathbf{v}_1 = (v_{n/2+1}, v_1, v_{n+1}\star) \). We exchange \( v_{n/2}\star \) and \( v_{n/2+1}\star \) and compute
\[
\begin{align*}
v_{n/2} & = v_{n/2+1}\star, \\
v_{n/2+1} & = v_{n/2}\star,
\end{align*}
\]
which completes the computation. The matrix vector product is then given by discarding the first and last entries of the local product vectors \( \mathbf{v}_i, i = 0, 1 \).
Let us have a closer look at the implementation of the matrix–vector product. First the local contribution to the matrix–vector multiplication is carried out, using only the local data structure of the matrix and the vector. In order to enhance efficiency, the local computation of the matrix–vector multiplication takes place in a tailored function tridiagprod implemented in an external C library lautils (see below). To avoid creating a new vector, we have equipped the matrix class with a vector prod, which contains the result after the multiplication has been carried out.

The next step in the parallel algorithm regards the need to communicate the boundary values of the product to the neighboring processes. This is done by loading the boundary values in prod into the list messages in the form (topid,(frompid,value)). We need to send frompid in addition to the value, since the order of the entries in the returned list is not deterministic; we need frompid to identify where the value belongs. The exchange and barrier synchronization takes place in the method exchangeMessages(message), which returns the received data as a list data of two tuples:

\[
[(\text{frompid1},\text{value1}), (\text{frompid2},\text{value2})]
\]

The frompid1 and frompid2 variables correspond to the process numbers of the two neighboring processes (i.e., left and right process in a one–dimensional problem). The variables value1 and value2 hold the corresponding exchanged values. We translate the nested tuple in data into a dictionary, as this is more convenient; we can then access the communicated value from the left process as message[pid-1] and the value from the right process as message[pid+1].

Having a high–level data–structure that allows matrix–vector products and common vector operations, we can easily develop iterative solvers for systems of linear equations, without the need to know the details of how the data–structure is implemented.

### 4.3 An Iterative Solver

We have already argued why Jacobi iteration is conceptually a relevant solution method for the linear system, at least if the goal is to outline ideas of how Python and BSP can be used in typical operations encountered in PDE solvers. We now write the linear system as \( \mathbf{A} \mathbf{x} = \mathbf{b} \). The Jacobi method is an iteration where in step \( m \) we compute

\[
\mathbf{r}^m = \mathbf{A} \mathbf{x}^m - \mathbf{b} \\
\mathbf{x}^{m+1} = \mathbf{x}^m - \mathbf{D}^{-1} \mathbf{r}^m.
\]

where \( \mathbf{D} = \text{diag}\{\mathbf{A}\} \). The above iteration is carried out until the norm of the residual \( \mathbf{r}^m \), relative to the norm of the initial guess for \( \mathbf{x} \), meets a predefined convergence criterion. If zero is used as initial guess, an absolute convergence criterion is applied instead. A unified code for serial and parallel Jacobi iteration is quite simple to implement when we utilize the previously introduced data structures.
The arguments to the Jacobi function are the linear system in the form $A\cdot x$ (initial guess), and $b$, the inverse of the diagonal of $A$, named $Dinv$, and the convergence criterion $e$. Sending vectors as DistVector instances and $Dinv$ and $A$ as DistTriDiagMatrix instances “turns on” the parallel version of this function, simply because the matrix–vector multiplication operator (*) and the vector operations are available in a parallel version for DistVector and DistTriDiagMatrix instances.

The Jacobi method above could be implemented in a more human readable form. For example, the two statements $r=A\cdot b$ followed by $r=b$ could be written as $r=A\cdot x-b$ instead. We choose the first formulation for efficiency reasons, since here only one new vector is created when computing the residual $r$. The other way of computing $r$ would require the creation of an extra temporary vector.

### 4.4 The Simulator

We can now write a simulator for the problem described in Section 4.1, using the data structure and iterative solver described in Sections 4.2 and 4.3. The parallel vector and matrix classes are collected in a Python module we have called ParLALib, while the Jacobi function has been placed in a module IterMethods. The application code is quite simple:

```python
from Numeric import zeros, Float, arrayrange, maximum; from math import exp
from Scientific.BSP import ParConstant, ParClass, ParRootFunction
from ParLALib import DistTriDiagMatrix, DistVector
from operator import add, from time import time; from sys import argv
from IterMethods import Jacobi

# Model parameters
T = 0.1 # Time to expiry
E = 1.0 # Exercise price at expiry
s = 0.2 # Volatility
r = 0.1 # Interest rate

# Numerical parameters
L = 3.0 # Length of truncated domain (S(infinity)=L)
M = int(argv[-1]) # Asset resolution (interior nodes)
N = 1 # Time resolution
_e = 1.e-200 # Conv. crit. in Jacobi iteration
maxIter = 500 # Max. iter. in Jacobi's method
h = 1.0*L/(M+1) # Cell size in "space" (S direction)
dt = 1.0*T/(N+1) # Time step

# Compute matrix entries and final condition
alpha = arrayrange(0, M+2, 1, Float) # Use alpha as index first for speed
S = b+alpha # Asset dimension
beta = r*dt+alpha # beta according to num. scheme
```
alpha = 0.5*dt*alpha*alpha*s*s # alpha according to num. scheme

_x = maximum(E*S, 0) # Final condition
_l = -alpha[1:]; _l[-1] = 0; # Fill lower diagonal (incl. BC)
_d = 1+2*r*dt+2*alpha*beta; # Fill main diagonal
_d[0] = _d[-1] = 1 # Include BCs
_u = -alpha[1:-1]-beta[:-1]; # Fill upper diagonal (BC not needed)

#alpha = beta = 0 # Free some memory
_Dinv = 1/_d # For the Jacobi method

# Create global classes
DistTriDiagMatrix = ParClass(DistTriDiagMatrix)
DistVector = ParClass(DistVector)

# Create global objects, using global classes
A = gDistTriDiagMatrix(_l, _d, _u) # Fill global matrix
x = gDistVector(_x) # and vec (using final condition)
_Dinv = gDistVector(_Dinv); e = ParConstant(_e)

_l = _d = _u = _x = _Dinv = 0 # Free memory after initialization

# Main program, solve problem
for n in xrange(1, N+2):
    t = time() # Start timing of the time-loop
    b = x.copy() # Update right hand side
    b[0] = E*exp(-r*n*dt); b[-1] = .0; # Boundary conditions
    x = Jacobi(A, x, b, _Dinv, e) # Solve at this time step
    t = time()-t # wall clock time of the time-loop
    print x
    print "M = %d: t = %f" %(M, t)

Besides the data structure initialization and the linear algebra computations, our simulator is also equipped with useful features for scientific investigations, including plotting of the solution (as in Figure 1) via the Gnuplot module and writing of reports with key results. Such data post processing and analysis are particularly convenient in a high-level language like Python.

4.5 Extending Python with Fortran and C Modules

The idea of high-level design, presented in Section 2.2, is to design an application or library for the high-level language. In order to ensure efficiency, the time-critical parts of the code are implemented using an efficient low-level language, normally Fortran 77, C, or C++.

In general there are two different ways to interface low-level codes. Existing codes should be interfaced with some automatic tool, like SWIG [2] or F2py [10]. This approach is preferable also when interfacing several functions migrated to the low-level language. For smaller tasks, an interface can be written manually, using the C API of Python. All of these approaches will be briefly illustrated in the following.

Our goal is to implement the tridiagonalprod function, called from the multiplication operator function (mul) in class DistTriDiagMatrix. The purpose of tridiagonalprod is to perform the local matrix–vector product (without any communication) in a process or the complete product in the serial case.
The simplest and most convenient way of implementing the matrix–vector product in a compiled language is probably to use Fortran 77 as programming language and the F2py or Pyfort tools for generating wrapper code such that the Fortran function can be called from Python. The F2py and Pyfort tools handle NumPy arrays and basic variables in a fully transparent way. For example, the calling Python code sends NumPy array object as arguments and the Fortran code sees plain arrays. The Fortran routine may look like this:

```fortran
subroutine tridiagprod (l, d, u, prod, n)
  integer n, i
  real*8 l(0:n+1), d(0:n+1), u(0:n+1), x(0:n+1), prod(0:n+1)
  do i = 1, n
    prod(i) = 0.0
  end do
  do i = 1, n
    prod(i-1) = prod(i-1) + u(i)*x(i)
    prod(i) = prod(i) + d(i)*x(i)
    prod(i+1) = prod(i+1) + l(i)*x(i)
  end do
end subroutine tridiagprod
```

The call to this routine from Python (in DistTriDiagMatrix._mul_) reads

```python
tridiagprod(self.l, self.d, self.u, other.v, prod)
```

That is, five NumPy arrays are sent to the routine. The size of the arrays (the n parameter in the Fortran argument list) is optional in Python; the wrapper code adds the size parameter when calling Fortran (since the size is available from the NumPy array objects).

The generation of wrapper code is particularly easy with F2py. Suppose the Fortran code resides in a file `tridiagprod.f` and we want to create a module `lautils` out of this. Running

```bash
f2py tridiagprod.f -m lautils -h lautils.pyf
```

creates a Fortran 90 module specification `lautils.pyf` of the interface to the Fortran code. One can edit `lautils.pyf` and run `f2py lautils.pyf` to adjust details of the interface, if desired, but this is not necessary in the present simple example. The above `f2py` command also creates utilities (makefile or `setup.py` script) such that compiling the Fortran code and the wrapper code (in C) and linking these to a shared library module `lautils` can be done by just typing one operating system command. To summarize, we can write a standard array-manipulating function in plain Fortran 77 and via two commands get a module that allows Python code to call the function. Python and Fortran 77 obviously constitute a very convenient and easy-to-use pair of programming languages.

A similar function could equally well be written in C:

```c
void tridiagprod (double *l, double *d, double *u,
                 double *v, double *prod, int n)
{
  
  
}
```

26
Generation of wrapper code can be accomplished by SWIG. However, the mapping between NumPy arrays and plain C arrays must be implemented via special SWIG typemaps. The work with specifying the SWIG input and compiling/linking the module is more comprehensive than when using Fortran and F2py.

There is one disadvantage with the Python–Fortran or Python–C approaches described above. The array entry type in Python and Fortran/C must be compatible, i.e., NumPy arrays with `Numeric.Float` entries must be declared as `real*8` or `double` precision in the Fortran code and as `double` pointers in the C code. If the entry types do not match, the computations in the C code will be erroneous. The Fortran wrappers, on the other hand, will generate code that copies the NumPy data to a new NumPy array compatible with the declarations in the Fortran code. Such hidden copies guarantee correct calculations but degrade performance significantly.

We can write a hand–made C function that calculates the matrix–vector product correctly regardless of the type of entries encountered in the NumPy array arguments, using the C API of NumPy. Showing the details of such a C function also gives a glimpse of the very different nature of Python and C and how these languages are integrated at the C level. It is natural to divide the presentation of this C function into two parts: parsing of the function arguments and the numerical computations.

The parsing part is required because data are stored differently in Python and C, simply because Python is a dynamically typed language (a variable can change its type) while C applies static typing (a variable has one and only one type). This means that Python has a different framework for storing, e.g., an integer than the simple `int` type used in C. The arguments sent from Python to a C function will be passed as a Python tuple object, where each argument in the call corresponds to an item in the tuple. In the C function, this tuple appears as a `PyObject*` pointer. The `PyObject` data type acts as a base class of all Python types, when viewed from C. We therefore need to pass the `PyObject*` pointer to a function `PyArg_ParseTuple` in the Python C library for parsing the tuple. Along with this pointer we must provide information about the type of arguments expected in the call and send the C variables that are supposed to hold these arguments in the C function. This should be clear from the following example.

In our case, we expect five NumPy arrays (three diagonals in the matrix, the vector to multiply with, and storage of the product). No information about the size of the arrays are needed since such data are stored in the NumPy objects. Technically, we shall extract five pointers to NumPy array C `structs`, as these objects are seen from C:

```c
void triadigprod (PyObject *self, PyObject *args)
{
    /* parse the arguments sent to this func via the args tuple */
    PyArg_ParseTuple (args, "O!O!O!O!O!", &PyArray_Type, &l,
                     &PyArray_Type, &d, &PyArray_Type, &u,
                     &PyArray_Type, &v, &PyArray_Type, &prod);
```
The `self` parameter is of no interest as long as we do not use user–defined Python classes in the calling code. The character 0 in the second argument to the `PyArg_ParseTuple` function signifies that the argument should be a pointer, and the exclamation sign indicates that an exception should be raised if the argument is not of the specified type, here `PyArray_Type`. Since we expect five such pointer arguments, we repeat 0! five times. Thereafter we provide five pairs of the pointer type (`PyArray_Type`) and the pointer variable (`PyArrayObject*`). Both `PyArray_Type` and `PyArrayObject*` are variable types defined in the `NumPy` C library. (For detailed information on the API for programming with `NumPy` arrays in C code, see the NumPy [1] documentation. For information on the general C API of Python, look up the Python/C API Reference at the Python website [7].)

We are now ready for the core of the C version of the local matrix–vector product function. The implementation of this function looks a bit complicated. The reason is that we operate directly on the data structures of the `PyArrayObjects` in accordance with the `NumPy` C API. The computation is carried out using pointer arithmetics on the data structures. For efficiency reasons, we store temporarily the strides and the data pointers, to avoid lookups of `C struct` members inside the computational loop. With some compilers this may be critical for achieving optimal performance.

```c
/* Get the size of the vectors, use just the main diag. */
size_t n = d->dimensions[0];

/* Store strides (size of datatype) and pointers to data to make the
 * loop more efficient */
p_stride = prod->strides[0];    p_data = prod->data;    
q_stride = d->strides[0];      q_data = d->data;      

for (i=0; i<n+2; i++)
    vval = *(double *)(v_data + i*p_stride); /* value of vector */
    vval = *(double *)(v_data + (i-1)*v_stride); /* upper */
    vval = *(double *)(v_data + (i+1)*v_stride); /* diagonal */
    vval = *(double *)(v_data + (i-1)*d_stride); /* lower */

return PyInt_FromLong(0);
```

## 4.6 Efficiency

Several topics are of interest when it comes to the efficiency of our simulator for solving the model PDE. We start by comparing the performance of the
simulator with that of another code, written in a comparable environment. The similarity between NumPy and Matlab is obvious, although Matlab is not commonly used for parallel computations. However, we can develop a quite similar, scalar Matlab simulator for our problem, making the comparison as fair as possible in the scalar case. The Matlab program must be based on vector operations to avoid explicit (slow) loops and use the same algorithms such that the total work for solving our problem is the same for the Python and Matlab codes. The code for the Matlab program is included in Appendix A.1.

We measure two different time parameters. The first is the wall clock (elapsed) time of the entire simulation. For real–time applications, this is perhaps the only relevant timing parameter, since it determines how long a computational scientist has to wait for the result of the simulator. The second time measurement is the wall clock time of the time loop only. This measurement reflects the computational (linear algebra) work; all the data structures are allocated and initialized prior to the time loop. With these two timings we are able to compare the efficiency of the data structure initialization and the linear algebra operations, separately, in both Python and Matlab.

During the experiments, we choose \( T = 1 \) and \( \Delta t = 0.5 \), resulting in two time steps only. For each spatial problem, we run 500 Jacobi iterations regardless of the norm of the residual. Thus, we control the number of arithmetic operations during the experiments. Of course, this number of Jacobi iterations does not, in general, suffice to solve a large problem to a given accuracy. However, in a modern solution–algorithm like multigrid, the smoother at each grid level is often a simple iterative method like the Jacobi method used here. In such a method, the number of iterations in the smoother will be fixed. All the simulations are run on a dual CPU shared memory Linux PC, and the Python simulator uses both CPUs in parallel.

The Python simulator could handle quite large problems, with more than 16 million unknowns. We were not able to run the Matlab code with that many unknowns. This indicates that the memory overhead caused by Python is smaller than that of Matlab.

It may be informative to see how inefficient a pure Python implementation of the matrix–vector product is. By pure Python, we mean Python and the Numeric module, i.e. we do not migrate the time-critical matrix-vector product to a tailored C or Fortran code. However, due to the fact that the Numeric module is really an interface to optimized C-code, the efficiency of this pure Python approach is dramatically better than a corresponding implementation using native Python list traversal. Since the matrix–vector NumPy operations generate temporary vectors, we were not able to run the largest problem with this pure Python version.

Table 1 shows the total time of the simulations and Table 2 the time of the time loop (note that “Python” refers to the implementation with the matrix–vector product migrated to C). We see that the speed of the Python simulator can compete with the Matlab code, at least when the problem size increases. When running the Python BSP simulator on two CPUs, we get timing results directly comparable with Matlab, and even better as the problem size increases.
<table>
<thead>
<tr>
<th>Problem size</th>
<th>$2^2 \cdot 10^3$</th>
<th>$2^6 \cdot 10^3$</th>
<th>$2^{10} \cdot 10^3$</th>
<th>$2^{12} \cdot 10^3$</th>
<th>$2^{14} \cdot 10^3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0.09</td>
<td>9.90</td>
<td>148.9</td>
<td>586.1</td>
<td>2394.6</td>
</tr>
<tr>
<td>Fortran</td>
<td>0.09</td>
<td>9.31</td>
<td>148.8</td>
<td>542.6</td>
<td>N/A</td>
</tr>
<tr>
<td>Matlab</td>
<td>0.7689</td>
<td>25.25</td>
<td>338.34</td>
<td>1347.2</td>
<td></td>
</tr>
<tr>
<td>Pure Python, one CPU</td>
<td>13.66</td>
<td>68.78</td>
<td>850.38</td>
<td>3240.4</td>
<td></td>
</tr>
<tr>
<td>Pure Python, two CPUs</td>
<td>16.14</td>
<td>50.28</td>
<td>545.0</td>
<td>2095.0</td>
<td></td>
</tr>
<tr>
<td>Python, one CPU</td>
<td>12.96</td>
<td>29.71</td>
<td>273.5</td>
<td>1072.4</td>
<td>4439.45</td>
</tr>
<tr>
<td>Python, two CPUs</td>
<td>12.13</td>
<td>24.71</td>
<td>183.90</td>
<td>686.36</td>
<td>3442.31</td>
</tr>
</tbody>
</table>

Table 1: Total run time for the simulations on a Linux PC.

<table>
<thead>
<tr>
<th>Problem size</th>
<th>$2^2 \cdot 10^3$</th>
<th>$2^6 \cdot 10^3$</th>
<th>$2^{10} \cdot 10^3$</th>
<th>$2^{12} \cdot 10^3$</th>
<th>$2^{14} \cdot 10^3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0.080</td>
<td>9.80</td>
<td>148.3</td>
<td>584.8</td>
<td>2386.0</td>
</tr>
<tr>
<td>Fortran</td>
<td>0.09</td>
<td>9.29</td>
<td>148.7</td>
<td>541.9</td>
<td>N/A</td>
</tr>
<tr>
<td>Matlab</td>
<td>0.7597</td>
<td>25.05</td>
<td>335.26</td>
<td>1334.9</td>
<td></td>
</tr>
<tr>
<td>Pure Python, one CPU</td>
<td>2.685</td>
<td>57.89</td>
<td>837.97</td>
<td>3224.0</td>
<td></td>
</tr>
<tr>
<td>Pure Python, two CPUs</td>
<td>1.725</td>
<td>37.24</td>
<td>529.29</td>
<td>2075.5</td>
<td></td>
</tr>
<tr>
<td>Python, one CPU</td>
<td>1.214</td>
<td>17.38</td>
<td>260.23</td>
<td>1054.5</td>
<td>4296.0</td>
</tr>
<tr>
<td>Python, two CPUs</td>
<td>0.8711</td>
<td>13.42</td>
<td>171.41</td>
<td>668.93</td>
<td>3073.5</td>
</tr>
<tr>
<td>Speedup (Python)</td>
<td>1.39</td>
<td>1.30</td>
<td>1.52</td>
<td>1.58</td>
<td>1.40</td>
</tr>
</tbody>
</table>

Table 2: Timing of the time loop in the simulations on a Linux PC.

<table>
<thead>
<tr>
<th>Problem size</th>
<th>$2^2 \cdot 10^3$</th>
<th>$2^6 \cdot 10^3$</th>
<th>$2^{10} \cdot 10^3$</th>
<th>$2^{12} \cdot 10^3$</th>
<th>$2^{14} \cdot 10^3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran Average</td>
<td>0.07</td>
<td>2.55</td>
<td>80.93</td>
<td>451.79</td>
<td>1970.39</td>
</tr>
<tr>
<td>Fortran Standard deviation</td>
<td>0.05</td>
<td>0.48</td>
<td>2.40</td>
<td>10.02</td>
<td>32.12</td>
</tr>
<tr>
<td>C Average</td>
<td>0.06</td>
<td>2.8</td>
<td>71.69</td>
<td>411.78</td>
<td>1750.13</td>
</tr>
<tr>
<td>C Standard deviation</td>
<td>0.20</td>
<td>0.57</td>
<td>5.64</td>
<td>20.96</td>
<td>87.14</td>
</tr>
</tbody>
</table>

Table 3: Total runtime, average and standard deviation for simulations on an IBM Regatta (AIX).
The speed of the pure Python simulator, with plain NumPy implementation of the matrix-vector product, shows that for tasks requiring heavy numerical work the savings from migrating critical code to C are considerable.

The other timing issue is the speedup gained from running the code on several processors. As the problem size increases, the time spent on communication should play a smaller role, and the speedup is expected to increase. As seen from Table 2, there is still room for improvement, since the speedup does not seem to approach two, the theoretical upper limit.

We have also implemented optimized, pure C and Fortran simulators, that solve the Black-Scholes problem using the same numerical method as the Python and Matlab programs. See Appendix A.1, A.2, and A.3 for an exact documentation of the constructs used in the Matlab, C and Fortran codes. All C and Fortran codes were compiled by the GNU gcc and g77 compilers on the Linux PC. As expected, the Fortran and C simulators are the most efficient. However, the results are comparable to both the Python and the Matlab codes, indicating the these tools are well suited not only for test problems with a few numbers of unknowns, but for real–world applications as well.

As a last test, we have run the the C and Fortran simulation on a IBM Regatta system with 1.3Ghz Power4 CPUs. Due to the lack of control regarding the load on the system, we have run a series of tests on this machine, and report the average timing of the simulations and the standard deviations, see Table 3. For these tests, we used the xlc and xlf compilers with the optimization options given Table 4. We do not have access to Matlab and Python/BSP environments on the Regatta, hence there is no direct comparision with these simulators.

5 Conclusions

Simulation programs encountered in computational science usually have two main levels: a program management level and a computing–intensive level. Traditional numerical software development has been very concerned with the computing–intensive level, often at the cost of decreased flexibility of the program management part. The result may be rigid, monolithic codes, which are hard to maintain and even harder to continuously adapt to the changing needs of scientists. In this paper, we have presented a high–level approach to scientific computing, using Python and its recent module extensions as programming language. Python has a particular strength for program management tasks and may provide a very flexible and highly configurable interface to computational tasks, file and database management, data analysis, visualization, and automatic report generation. In other words, the increasingly popular Python
language has a potential for significantly increased human productivity also in scientific computing. How much computational efficiency that must be sacrificed is an open question, but recent extensions of Python, mainly offered by the modules Numeric, Scientific and SciPy, dramatically decrease the shortcomings of Python as a programming environment in scientific computing.

Scientific computations, especially loops over standard Python array (list) structures, are known to be very slow, often a factor 10-100 slower than a corresponding plain C implementation. Hence, array processing should be performed in C, C++, or Fortran code. There are basically three ways of gaining the required efficiency in Python codes: (i) using NumPy arrays and associated optimized array operations, (ii) interfacing existing C, C++, or Fortran libraries, or (iii) designing the application in Python and migrating the bottlenecks to Fortran or C. We have advocated the latter approach, as this makes possible a high–level design of numerical codes, probably resulting in increased flexibility and better tools for explorative research. The time–critical parts to be migrated to Fortran or C can either utilize hand–coded routines written by the application developer or utilize parts of existing libraries.

Many scientific computing applications require parallel computers. A particular feature of the present paper is to show that the previously mentioned advantages of high–level design also carry over to the concurrent computing case. This is enabled by combining Python with the BSP parallel programming paradigm. The combination represents a more high–level, and hence easier, implementation of parallel algorithms than the common MPI standard.

The Python BSP model for parallel programming differs conceptually from the MPI model. In the MPI model, the application is made up of N programs communicating with N – 1 others, whereas the Python BSP program is a parallel application running on a N processor machine. To facilitate this idea, a Python BSP program has two different scopes: local and global. Also, as Python BSP is a high–level parallel programming model, almost arbitrary Python objects can be communicated between processes. Several examples have been presented to illustrate these concepts.

Our most advanced example on utilizing Python in parallel computing concerned numerical solution of partial differential equations. To illustrate the application of Python and BSP in this application domain, we addressed a simple one–dimensional problem from computational finance. The numerical methods were chosen such that we encountered typical operations that occur in more advanced higher–dimensional problems. Simulators were developed in Python and in Matlab, as well as native C. The most time–critical operation in the Python code was a matrix–vector product, which we showed could be migrated to C and Fortran functions, applicable to both scalar and parallel computations. The timings showed that Matlab ran about twice as fast as Python in this particular application, and, as expected, C was faster than Matlab. However, the Python code was very easily parallelized, and the parallel version ran at the speed of Matlab on a standard desktop PC with two processors. Further improvements in speed are enabled by moving the code to a computer with more processors. On such a computer, Python and BSPlib must of course be installed (BSP can
operate both on shared memory and distributed memory machines).

When discussing speed, we think focusing on computational speed only is too limiting. The critical time parameter for researchers is the time it takes from getting an idea to the moment the idea is explored by computations. This “speed of research” can benefit greatly from working in a convenient and human-efficient programming environment, such as Matlab and Python. The widespread tendency of scientists to move from Fortran and C to Matlab illustrates this point.

Looking more closely at the Matlab and Python codes, there are some striking similarities: intuitive and easy-to-read syntax, few statements (in comparison with Fortran and C), as well as gluing of computations, data analysis, and visualization, perhaps through graphical interfaces. There are also some significant differences: Python is a more powerful programming language, parallel programming is not only possible with Python but also very convenient, Python has a more flexible and easier-to-use interfacing to Fortran, C, and C++ codes, with easier shuffling of data between different tools, and Python has rich functionality for file and directory management, text processing (via regular expressions, for instance), Internet programming/applications, to mention some areas of increasing importance also in computational science.

Of course, this paper only gives a glimpse of some promising aspects of high-level Python programming for numerical applications. It remains to develop Python software on a larger scale and measure the advantages and shortcomings of the result over traditional programming styles before firm conclusions about the feasibility of Python in a computational science context can be drawn.

Acknowledgements. The authors want to thank Dr. Xing Cai for his many useful comments on this manuscript and for assistance with running efficiency tests.

A Implementational Details

Timing results of computer codes are sensitive to the details of the implementation. There are many ways of performing a computational task in Matlab, C, and Python NumPy, some of which may be inefficient. We therefore list the Matlab, C, and Python NumPy codes to enable the reader to judge the relevance of the implementations and the corresponding timings.

A.1 The Matlab Code

```matlab
function [t, x, h]=BS(M, L, N, maxIter)
    % [v,S,h]=BS(M,Sinf,N)
    % M number of inner nodes
    % L "infinity"
    % N number of time steps
    % maxIt max. number of iterations in Jacobi.
    % x The solution
    % S "x-vector"
    % h Step size.

    T = 0.1; E = 1.0; s = 0.2;
```

33
r = 0.1; e = 1.e-200;

h = L*1.0/(M+1);
dt = T/(N+1);

alpha = [0:M+1]; % Asset dimension
beta = r*dt*alpha; % beta in numerical scheme
alpha = 0.5*dt*s*s*alpha.*alpha; % alpha in numerical scheme

x = max(0, E-S)'; % Final condition
l = -alpha(2:M+2); l(1) = 0; % Fill lower diagonal (inc. BC)
d = 1 + r*dt + 2*alpha + beta; % Fill diagonal
d(1) = 1; d(M+2) = 1; % Include BCs
u = -alpha(1:M+1) - beta(1:M+1); % Fill upper diagonal
% (BC not needed, see scheme)
alpha = 0; beta = 0; % Free some memory

% Create matrices and fill with values according to FDM scheme
A = sparse(1:M+2, 1:M+2, d, M+2, M+2) +
    sparse(2:M+2, 1:M+1, l, M+2, M+2) +
    sparse(1:M+1, 2:M+2, u, M+2, M+2);
Dinv = sparse(1:M+2, 1:M+2, 1./d, M+2, M+2); % For the Jacobi method

l = 0; d = 0; u = 0; % Free memory after initialization

t1 = clock; % Start timing of the time-loop
for n=1:N+1,
    b = x; % Update right hand side
    b(1) = E*exp(-r*n*dt); % Boundary conditions
    b(M+2) = .0;
    x = jacobi(A, x, b, Dinv, e, maxIter); % Solve at this time step
end;
t = etime(clock, t1); % Compute the wall clock time of the time-loop

function x = jacobi(A, x, b, Dinv, e, maxIter)
r = A*x - b; x_norm = norm(x); i = 0;
if x_norm < e,
    tolerance = e;
else
    tolerance = e*x_norm;
end;
while norm(r) > tolerance & i < maxIter,
i = i+1;
x = x - Dinv*r;
r = A*x - b;
end

A.2 The C Code

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Matrix-vector product, in the form A=tridiag{l,d,u}, p=A*x
void mvp(double* l, double* d, double* u, double* x, double* p, int n);
// Incremental matrix-vector product, in the form A=diag(b), a=A*a
void dmvp(double* a, double* b, int n);
void isub(double* a, double* b, int n); // Incremental subtraction, a-=b

double alpha(int i, double r, double dt, double s); // Discretization
double beta(int i, double r, double dt); // Discretization
double mnorm(double* a, int n); // Discrete l_2 norm
int main(int argc, char *argv[])
{
    /* Model and numeric parameters */
    const int N=atoi(argv[1]); // Number of inner nodes
    const int Nt=atoi(argv[2]); // Number of time steps
    const double ir=.1; // Interest rate
    const double vol=2.; // Volatility of underlying asset
    const double E=1.; // Exercise price of European put option
    const double T=.1; // Expiry date of contract
    const double L=3.; // Length of truncated domain (L=S_infinity)
    const double dt=T/(N+1); // Time step size
    const double h = L/(N+1); // Spatial resolution (regular grid)
    const double e=1.e-200; // Conv. crit. in Jacobi iteration
double tolerance; // Adjusted conv. crit in Jacobi iteration
double x_norm;

    /* Allocation and Initialization */
    const int stride = sizeof(double); // sizeof(double) = 8 here
    const int strided = stride*(N+2); // Amount of memory to allocate
    double* x = (double *)malloc(strided);
double* b = (double *)malloc(strided);
double* l = (double *)malloc(strided);
double* d = (double *)malloc(strided);
double* u = (double *)malloc(strided);
double* dxv = (double *)malloc(strided);
double* p = (double *)malloc(strided);
double* r = (double *)malloc(strided);

    int i,j,k,n; double tmp; clock_t t1,t2; // Help variables
double min,max;

    for(i=0;i<=N+1;i++)
    { // Pointer arithmetics, initialize data structure
        *(x+i) = E - h*i; if(x[i]<.0) x[i] = .0; // Final condition
        *(l+i) = - alpha(i+1, ir, dt, s); // Matrix according to disc. meth.
        *(d+i) = 1. + ir*dt + 2.*alpha(i , ir, dt, s) + beta(i , ir, dt);
        *(u+i) = - alpha(i-1, ir, dt, s) - beta(i-1, ir, dt);
    }

    d[0]=d[N+1]=1.; l[N]=0; // Boundary conditions
    for(i=0; i<=N+1; i++) b[i]=1./d[i]; // inv(diag(A)), for Jacobi's method

    /* Solve the numerical problem */
    t1=clock(); // Start time of time-loop
    for(n=1; n<N+1; n++)
    { // Time loop
        memcpy(b, x, strided); // Update right hand side
        b[0]=b[0]+exp(-ir*n*dt); b[0]=0.0; // Enforce BCs
        // Jacobi's method
        mvp(l, d, u, x, M); // r = A*x
        isub(r, b, M); // r = r-b = A*x-b
        if (x_norm < e) tolerance = e; else tolerance = e*x_norm;
        while ((mynorm(r,M) > tolerance) && (++i<maxIter))
        {
            dmvp(r, d, v, M); // r = dinv*r
            isub(r, b, M); // r = A*x
            mvp(l, d, u, x, M); // r = A*x
            isub(r, b, M); // r = r-b = A*x-b
        }
        // End of Jacobi's method
    }
    t2=clock(); // Stop time of time-loop
    printf("%lf\n",(double)(t2-t1)/CLOCKS_PER_SEC);
    // for(i=0; i<=N+1; i++) printf("%f\n",i,x[i]);
    min = 2.0; max = -2.0;
for(i=0;i<=M+1;i++) {
    printf("x[%d] = %f
",i,x[i]);
    if ( x[i] < min )
        min = x[i];
    if ( x[i] > max )
        max = x[i];
}
printf("max: %f, min: %f"

return 0;
}
void mvp(double* l, double* d, double* u, double* x, double* p, int n){
    double x_val; int i;
    *(p) = *(d)*(*(x)); *(p+1) = *(l)*(*(x));
    for(i=1;i<=n;i++) { // Compute the matrix vector product as loop over x.
        x_val=*(x+i);
        *(p+i-1) += *(u+i)*x_val;
        *(p+i) += *(d+i)*x_val;
        *(p+i+1) = *(l+i)*x_val;
    }
    *(p+n+1) = *(d+n+1)*(*(x+n+1));
}
void dmvp(double* a, double* b, int n){
    int i;
    for(i=0; i<=n; i++)
        *(a+i) *= *(b+i);
}
void isub(double* a, double* b, int n){
    int i;
    for(i=0; i<=n; i++)
        *(a+i) -= *(b+i);
}
double mynorm(double* a, int n){
    int i; double nrm=0;
    for(i=0; i<=n; i++)
        nrm += *(a+i)*(*(a+i));
    return sqrt(nrm);
}
double alpha(int i, double r, double dt, double s){ return .5*dt*i*i*s*s; }
double beta (int i, double r, double dt){ return r*dt*i; }

A.3 The Fortran Code

program bs
  implicit logical (a-z)
  integer M, nmax, maxIter
  real*8 ir, s, E, T, xL, dt, h, tolerance, x_norm, tmp, tol
  real*8 dtime, t1, t2, alpha, beta, mynorm
  external dtime
C number of inner nodes (hardcoded)
  parameter (M=16384000)
  real*8 x(0:M+1), b(0:M+1), l(0:M+1), d(0:M+1), u(0:M+1), > p(0:M+1), r(0:M+1), dinv(0:M+1)
  integer i, j, iter, n
C Model and numeric parameters
C Number of inner time steps:
    nmax=1
C Max. iter. in Jacobi’s method:
maxIter=500
C Interest rate:
ir=.1
C Volatility of underlying asset:
s=.2
C Exercise price of European put option:
E=1.
C Expiry date of contract:
T=.1
C Length of truncated domain (xL=S_infinity):
xL=3.
C Time step size:
dt=T1./float(nmax+1)
C Spatial resolution (regular grid):
h = xL/float(M+1)
C Tolerance in Jacobi method:
tol = 1.0e-200
do i = 0, M+1
   x(i) = E - h*i
   if (x(i) .lt. 0.0) then
      C Final condition:
x(i) = .0
   end if
C Matrix according to disc. meth.:
l(i) = - alpha(i+1, ir, dt, s)
d(i) = 1. + ir*dt + 2.*alpha(i, ir, dt, s) + beta(i, ir, dt)
u(i) = - alpha(i-1, ir, dt, s) - beta(i-1, ir, dt);
end do
C boundary conditions:
d(0) = 1.0
d(M+1) = 1.0
l(0) = 0.0
do i = 1, M+1
   dinv(i) = 1.0/d(i)
end do
C solve the numerical problem:
t1 = 0.0
t1 = dtme(t1)
do n = 1, nmax+1
do i = 0, M+1
   b(i) = x(i)
   end do
   b(0)=E*exp(-ir*n*dt)
   b(M+1)=0.0
C Jacobi's method
C r = A*x:
call mvp(l, d, u, x, r, M)
C r = r-b = A*x-b:
call isub(r, b, M)
   x_norm = mynorm(x,M)
   if (x_norm .lt. tol) then
      tolerance = tol
   else
      tolerance = tol*x_norm
   end if
   i=0
write("",'n='n
   do while ((mynorm(r,M) .gt. tolerance) .and. (i .le. maxIter))
i = i + 1
r = dinv*r:
call dmvp(r, dinv, M)
C x = x-r = x-dinv*r:
call isub(x, r, M)
C r = Ax:
call mvp(l, d, u, x, r, M)
C r = r-b = A*x-b:
call isub(r, b, M)
end do
t2=dtime(t1)
write(*,*) 'tot.iter=',i-1,' M=',M,' time-loop (secs) =', t2-t1
write(*,*) 'r-norm=',mynorm(r,M)
end subroutine mvp(l, d, u, x, p, n)
integer n
real*8 l(0:n+1), d(0:n+1), u(0:n+1), x(0:n+1), p(0:n+1)
real*8 x_val
integer i
p(0) = d(0)*x(0)
p(1) = l(0)*x(0)
do i = 1, n
x_val = x(i)
p(i-1) = p(i-1) + u(i)*x_val
p(i) = p(i) + d(i)*x_val
p(i+1) = p(i+1) + l(i)*x_val
end do
p(n+1) = p(n+1) + d(n+1)*x(n+1)
return
end subroutine dmvp(a, b, n)
integer n, i
real*8 a(0:n+1), b(0:n+1)
do i = 0, n+1
a(i) = a(i)*b(i)
end do
return
end subroutine isub(a, b, n)
integer n, i
real*8 a(0:n+1), b(0:n+1)
do i = 0, n+1
a(i) = a(i) - b(i)
end do
return
end
real*8 function mynorm(a, n)
integer n, i
real*8 a(0:n+1), nrm
nrm = 0.0
do i = 0, n+1
nrm = nrm + a(i)*a(i)
end do
mynorm = sqrt(nrm)
return
end
real*8 function alpha(i, r, dt, s)
integer i
real*8 r, dt, s
alpha = .5*dt*i*i*s*s
A.4 The Python NumPy Code

This is a pure Python implementation of the PDE simulator, utilizing NumPy and Scientific Python functionality. The difference between this code and the one developed earlier is that the tridiagonal matrix–vector product is now carried out by operations on NumPy arrays inside the Python script.

class DistTriDiagMatrix(ParBase):
    # Global matrix init
    def __parinit__(self, pid, nprocs, l, d, u):
        self.num_entries = d.shape[0]  # dimension
        self.type = "DistTriDiagMatrix"
        self.pid = pid
        self.nprocs = nprocs
        chunk_size = (d.shape[0]+self.nprocs-1)/self.nprocs
        self.cs = chunk_size
        self.l = l[pid*chunk_size:(pid+1)*chunk_size]  # Lower diagonal
        self.d = d[pid*chunk_size:(pid+1)*chunk_size]  # Diagonal
        if pid == nprocs-1:
            self.l = concatenate((self.l, z), 1)
        if pid == 0:
            self.u = concatenate((z, u[0:chunk_size-1]), 1)  # Upper diagonal
        else:
            self.u = u[chunk_size*pid-1:(pid+1)*chunk_size-1]
        self._prod = zeros(len(self.d)+2, Float)  # matrix-vector prod

    # Local matrix init
    def _init__ (self, l, d, u, pid=0, nprocs=0):
        self.l = concatenate((l, array([0])), 1)
        self.d = d
        self.u = concatenate((array([0]), u), 1)
        self._prod = zeros(len(self.d)+2, Float)
        self.type = "DistTriDiagMatrix"
        self.pid = pid
        self.nprocs = nprocs

    def __repr__ (self):
        return "\n" + repr(self.l)+repr(self.d)+repr(self.u)
# Matrix-matrix addition
def __add__ (self, other):
    if self.type == other.type and self.num_entries == other.num_entries:
        return DistTriDiagMatrix(self.l+other.l, self.d+other.d, self.u+other.u, self.pid, self.nprocs)
    else:
        raise ValueError, "Incompatible dimensions or matrix types"

# Matrix-vector multiplication
def __mul__ (self, other):
    if len(other.v) != len(self.d):
        raise ValueError, "incompatible lengths"
    pid = self.pid
    nprocs = self.nprocs
    prod = self._prod
    # For efficiency
    # Compute local matrix vector product
    prod[0] = prod[-1] = 0.0
    prod[1:-1] = self.d*other.v
    prod[2:] += self.l*other.v
    prod[:-2] += self.u*other.v
    # The size of prod is len(self.d)+2. The additional entries must be # communicated to the neighboring procs in order to compute the product # correctly
    messages = []
    if pid > 0:  # Send to pid-1.
        messages.append((pid-1, (pid, prod[0])))
    if pid < nprocs-1:  # Send to pid+1
        messages.append((pid+1, (pid, prod[-1])))
    data = self.exchangeMessages(messages)
    messages = []
    # We need to know that pid sent the # message, and use a hash for this.
    if nprocs > 1:
        messages[data[0][0]]=data[0][1]
        if (pid >0) & (pid < nprocs-1):
            messages[data[1][0]]=data[1][1]
        if pid > 0:  # prod[1] += messages[pid-1]
            prod[1] += messages[pid-1]
        if pid < nprocs-1:  # prod[-2] +=messages[pid+1]
            prod[-2] += messages[pid+1]
    return DistVector(prod[1:-1], pid, nprocs, self.cs)

References


