Parallel Programming with BSP in Python

Konrad Hinsen
Centre de Biophysique Moléculaire (UPR 4301 CNRS)
Rue Charles Sadron
45071 Orléans Cedex 2, France

1 Introduction

Although a lot of progress has been made in the design of parallel algorithms and in the theory of parallel computation, the number of practically usable parallelized programs is still very small, as is the number of developpers who are working on parallelized code. Non-specialists, e.g. computational scientists, hardly even consider developping their own parallelized code. One reason is that writing parallel code is hard, even when the parallelization of the underlying algorithms is simple.

The BSP module for Python is an attempt to facilitate parallel programming and make it accessible to computational scientists and engineers. Compared to the currently popular parallel programming techniques, there are two major innovations: the use of a high-level language, Python, with high-level data structures, and the use of the Bulk Synchronous Parallel (BSP) model instead of the better known message-passing model.

The advantages of using a high-level programming language are basically the same as for serial code: rapid development, more interactive testing, and high-level data structures. Parallelization can take advantage of the same features. The data that is exchanged between processors can be (almost) arbitrary Python objects. Data exchange is handled by special classes, and users can define their own classes that represent distributed data.

The BSP model is another ingredient for simplfying parallel programming. It divides the execution flow of a parallel program into alternating computation and communication phases. Although this might at first seem very restrictive compared to the popular message-passing model, which allows communication and computation to occur in parallel, it is not found to be a restriction in practice. BSP is, however, much simpler to use, especially since it virtually eliminates the risk of deadlocks.

The use of a high-level language for parallel programming might be surprising to many computational scientists. Parallel machines are mainly used for perfor-

mance, whereas high-level languages are considered to be slow. However, for some applications the overhead is negligible (e.g. if most of the computations can be expressed in terms of efficient array operations), and for most others a translation of the time-critical part of an application (usually a very small part) into a compiled language is sufficient to obtain acceptable performance. And even if ultimately the whole program will have to be rewritten, a high-level language provides a valuable rapid-prototyping platform.

The current state of the Python BSP module must be labelled as "experimental". It was inspired by the BSML library for Objective Caml [1] and developed with applications in computational chemistry in mind, of which only two have been implemented so far. Other applications domains may have other requirements, and the implementation itself can be improved in many ways. No support for C extension modules is provided at the moment. It cannot be excluded at the present time that future releases will be incompatible with this one. Moreover, more and more utility modules will certainly be added to the basic core.

2 Usage

The BSP module is part of the ScientificPython package. Installing this package, including the BSPlib or the MPI interface, is sufficient to use the BSP module.

If you use BSPlib for low-level communications, you must use the `bsppython` executable to run your programs. The BSPlib manual explains all the available options. If you use MPI, use the `mpipython` executable instead, and consult your MPI manual for how to start it (usually via the command `mpirun`).

By far the most convenient way for developing and testing parallel code in Python is the interactive parallel console. It behaves almost exactly like the standard Python interpreter, but you get the output from all processors. It can be used with the Emacs mode for Python just like the standard Python interpreter. This combination is at the moment the best development environment for Python BSP programs.

It is possible to use the standard Python interpreter to execute Python BSP programs on a serial machine, it will then behave like on a parallel machine with a single processor. In many cases, this feature allows to use a single code for both serial and parallel execution, especially since there is almost no overhead for communications; objects "sent" to the same processor are directly returned by the communications calls, i.e. they are not even copied.

3 Concepts

This section gives an overview over the concepts that form the basis of BSP programming in Python. These concepts will be illustrated in the following section

by examples.

The most important concept to keep in mind is that a Python BSP program is a program for a parallel machine made up of N processors and *not* a program for one processor that communicates with $N-1$ others, which is the view taken with message-passing models. A BSP program has two levels, local (one processor) and global (all processors), 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 is a synchronized global operation in which all processors participate.

3.1 Processors

In the BSP model, a parallel machine is made up of N processors that can communicate with each other. Each processor receives a number between 0 and $N-1$. All processors are considered equal. For operations that give a special function to one of the processors, this role is by convention taken by processor number 0. For example, adding up values over all processors leaves the result by default on processor 0.

3.2 Local and global objects

The most important concept for BSP programming in Python is the distinction between local and global objects. Local objects are standard Python objects, they exist on a single processor. Global objects exist on the parallel machine as a whole. They have a local representation on each processor, which may or may not be the same everywhere. Communication is based on global objects as well.

Global objects can be used in different ways. In the simplest form, a global object represents a constant that is available on all processors. Another common situation is that the local representation is a function of the processor number and perhaps the total number of processors. For example, a global object "processor id" would have a local representation equal to the processor number. Finally, global objects often represent data sets of which each processor stores a part, the local representation is then the part of the data that one processor is responsible for.

3.3 Local and global 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 representations of the global objects. In most cases, the local function is the same on all processors, but it is also common to have a different function on one processor, usually number 0.

3.4 Local and global classes

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 representations.

3.5 Communication

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.

An important property of all communication operations is that they are *synchronizing*. This means that when a processor reaches the communication operation, it returns only after all other processors have reached and executed this operation as well. Again this is a consequence of the BSP model. Although it may seem restrictive compared to the message-passing model, it turns out to be an advantage in practice, as it eliminates deadlocks.

A particular advantage of Python in this context is that almost any kind of object can be transmitted to other processors. The only limitation is that the Python module `cPickle` must be able to serialize the object type. This applies to most of the built-in types as well as to any object defined by a Python class. See the Python Library Reference Manual for details.

Efficiency-conscious users should also note that Numerical Python arrays are transmitted without pickling and thus with practically no overhead compared to direct MPI communication. It is therefore advisable to store large data objects in the form of arrays where possible.

3.6 Input and output

I/O is always a problematic issue in parallel computing as it is not standardized. On a cluster of workstations, for example, each processor typically has its own I/O system, which may or may not have access to the same files. On dedicated parallel machines, anything is possible. The example programs in this tutorial restrict file I/O to processor 0. This seems like a minimal requirements approach that should work everywhere.

4 Standard global classes

The classes `ParConstant`, `ParData`, and `ParSequence`, all of which are subclasses of `ParValue`, are the core of the BSP module. They are global classes that represent global data objects which can be used in communication and in parallelized

computations. 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 processors. Example:

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

has a local representation of 0 on all processors.

`ParData` defines the local representation as a function of the processor number and the total number of processors. Example:

```
pid = ParData(lambda pid, nprocs: pid)
```

has an integer (the processor number) as local representation

`ParSequence` distributes its argument (which must be a Python sequence) over the processors as evenly as possible. Example:

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

divides the ten integers among the processors. With two processors, number 0 receives the local representation `[0, 1, 2, 3, 4]` and number 1 receives `[5, 6, 7, 8, 9]`. With three processors, number 0 receives `[0, 1, 2, 3]`, number 1 receives `[4, 5, 6, 7]`, and number 2 receives `[8, 9]`.

All these classes define the standard arithmetic operations, which are thus automatically parallelized.

5 A first 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 processors, each processor calculates its share, and in the end all the results are communicated to one processor that takes care of output. In the following example, the input values are the first 100 integers, and the computation consists of squaring them.

```
from Scientific.BSP import ParSequence, ParFunction, ParRootFunction
import operator

# The local computation function.
def square(numbers):
    return [x*x for x in numbers]

# The global computation function.
```

```
global_square = ParFunction(square)

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

# The global output function - active on processor 0 only.
global_output = ParRootFunction(output)

# A list of numbers distributed over the processors.
items = ParSequence(range(100))

# Computation.
results = global_square(items)

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

# Output from processor 0.
global_output(all_results)
```

The local computation function is a straightforward Python functions: it takes a list of numbers, and returns a list of results. The call to `ParFunction` then generates a global function whose local representation is `square` on all processors. For the output function the requirements are different: we want one processor (number 0) to handle the output, while the others do nothing. This is arranged by the class `ParRootFunction`.

Once the functions are defined, the program proceeds in a straightforward way. `ParSequence` takes care of distributing the input items over the processors, and the call to `global_square` does all the computation. Before processor 0 can output the results, it has to collect them from all other processors. 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 processors instead of over the elements of a sequence. 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 has the useful property of working correctly independently of the number of processors it is run on. The number of processors can even be higher than the number of input values. However, the program is not necessarily efficient for any number of processors, 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 processor ID along with the data, and the receiving processor must sort the

incoming data according to processor ID before performing the reduction.

Another useful property of the program is that there is no significant overhead compared to serial code when running with only one processor, as in that case communication operations become simply copies in memory.

6 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 processors. In the example, a list of items (letters) is distributed over the processors, 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 processor to the next, until after $N - 1$ iterations each processor has seen all data. The new features that are illustrated by this example are general communication and accumulation of data in a loop.

```
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 (here letters) distributed over the processors.
my_items = ParSequence('abcdef')

# The number of the neighbour to the right (circular).
neighbour_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(neighbour_pid)[0]
    pairs.addValue(global_makepairs(my_items, other_items))

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

# Output results from processor 0.
global_output(all_pairs)
```

The essential communication step is in the line

```
other_items = other_items.put(neighbour_pid)[0]
```

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

The return value of `put` is another global object that stores all the data that a processor has received in the form of a list of values. It is important to keep in mind that the order in which the values appear in the list is completely unspecified. If it is important to know which value comes from which processor, the processor ID has to be communicated along with the data. In practice this is rarely necessary.

In the example, each processor 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 processor. After repeating this $N-1$ times, each processor has seen all the data.

The other new feature in this example is the accumulation of data in a `ParAccumulator` object. A `ParAccumulator` is a variation of a `ParData` object. Its initial value is the second argument (here an empty list). The method `addValue` adds some argument value to the current value. Finally, the method `calculateTotal` performs a reduction operation, just like `reduce` used in the previous example.

Like the previous example, this one works with any number of processors, but the order of the pairs in the result is not always the same. It should also be noted that the input sequence (here a string of characters) is completely arbitrary and can contain arbitrarily complex Python objects.

7 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 processors might come from a file or from previous computations, and might thus need to be distributed explicitly by one processor 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 are big. If the results must be written to some file immediately, and if only processor 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, processor 0 reads the input numbers from a file, and distributes them explicitly to the other processors. 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 processor and then sends the result back to processor 0 for output.

```
from Scientific.BSP import ParFunction, ParRootFunction, ParMessages, \
                           ParConstant, ParIterator, ParIndexIterator, \
                           numberOfProcessors
import operator, string

# The local and global input functions.
def input():
    data = open('numbers').readlines()
    numbers = map(string.atoi, 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*x
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()[0]

# Computation and output loop.
for item in ParIterator(items):
    result = global_square(item)
    collected_results = result.put(ParConstant([0]))
    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 processors 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 processor. The result of the global input function is a list of (pid, numbers) pairs on processor 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 processor in a single operation. This can be achieved by explicitly constructing a `ParMessage` object, whose local values are lists of (pid, data) pairs, which are sent using the `exchange` method. The result of this operation is a global object storing the incoming values, as in the case of a `put` operation.

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 processor 0 and written to the output file.

In general, the number of data items handled by each processor will not be exactly equal. Some processors 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:

```
for index in ParIndexIterator(items):
    result = global_square(items[index])
    collected_results = result.put(ParConstant([0]))
    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 processors. If this presents a problem in practice, it must be solved by using a special storage format.

8 Distributed data classes

Although `ParValue` and its subclasses are very flexible (not all of its features have been demonstrated here), there are situations in which it is more convenient to write special-purpose distributed data classes that can handle communications within their methods. This is illustrated in the following example, which treats a special case of parallelized matrix multiplication. A special-purpose distributed data class implements the distributed matrices.

The special case that is considered here allows matrices to be distributed along the row or the column index. However, addition is implemented only between matrices of equal distribution mode, and multiplication is implemented only between a column-distributed and a row-distributed matrix. The practical situation for which this arrangement was optimized is matrix-vector multiplication in which the first matrix is a square matrix and the second one a column vector.

The basic rules for distributed classes are the following: the class that implements the local representation must inherit from `ParBase`, and the corresponding global class is generated by `ParClass`. When an instance of the global class is created, the local representations are created and initialized *by calling the special method* `__parinit__` instead of the ususal `__init__` method. The latter is used when a local representation is created from inside a method of a local class. The global class provides all the attributes and methods of the local one, suitably translated into global objects and global methods.

To simplify the following example, no proper output routines are provided for the results. A simple `print` statement outputs the local value (with an indication of class and processor number). This assumes that the standard output of all processors is accessible to the user, which is often the case, and almost a requirement for convenient development and debugging.

```
from Scientific.BSP import ParClass, ParBase, numberOfProcessors
import Numeric, operator

class DistributedMatrix(ParBase):

    def __parinit__(self, pid, nprocs, matrix, distribution_mode):
        self.full_shape = matrix.shape
        self.mode = distribution_mode
        if distribution_mode == 'row':
            chunk_size = (matrix.shape[0]+numberOfProcessors-1) \
                         / numberOfProcessors
            self.submatrix = matrix[pid*chunk_size:(pid+1)*chunk_size, :]
        elif distribution_mode == 'column':
            chunk_size = (matrix.shape[1]+numberOfProcessors-1) \
                         / numberOfProcessors
            self.submatrix = matrix[:, pid*chunk_size:(pid+1)*chunk_size]
        else:
            raise ValueError, "undefined mode " + distribution_mode

    def __init__(self, local_submatrix, full_shape, distribution_mode):
        self.submatrix = local_submatrix
        self.full_shape = full_shape
        self.mode = distribution_mode

    def __repr__(self):
        return "\n" + repr(self.submatrix)

    def __add__(self, other):
        if self.full_shape == other.full_shape and self.mode == other.mode:
            return DistributedMatrix(self.submatrix+other.submatrix,
                                     self.full_shape, self.mode)
        else:
            raise ValueError, "incompatible matrices"

    def __mul__(self, other):
        if self.full_shape[1] != other.full_shape[0]:
            raise ValueError, "incompatible matrix shapes"
        if self.mode == 'row' or other.mode == 'column':
            raise ValueError, "not implemented"
        product = Numeric.dot(self.submatrix, other.submatrix)
        full_shape = product.shape
        chunk_size = (full_shape[0]+numberOfProcessors-1)/numberOfProcessors
        messages = []
```

```
        for i in range(numberOfProcessors):
            messages.append((i, product[i*chunk_size:(i+1)*chunk_size, :]))
        data = self.exchangeMessages(messages)
        sum = reduce(operator.add, data, 0)
        return DistributedMatrix(sum, full_shape, 'row')

gDistributedMatrix = ParClass(DistributedMatrix)

m = Numeric.resize(Numeric.arange(10), (8, 8))
v = Numeric.ones((8, 1))
matrix = gDistributedMatrix(m, 'column')
vector = gDistributedMatrix(v, 'row')
product = matrix*vector
print product
```

9 Distributed file access

A much more elaborate example of distributed data classes is given by the classes `ParNetCDFFile` and `ParNetCDFVariable` in the module `Scientific.BSP.IO`. These classes implement distributed access to netCDF files, using an interface that is almost identical to the one in `Scientific.IO.NetCDF`. When a file is created or opened, one dimension is specified for distribution. Every processor then becomes responsible for a specific slice of all variables that use this dimension.

A practical example is given below. It reads a netCDF file containing a function $F(q,t)$ (the intermediate scattering function used with neutron scattering experiments) and performs a Fast Fourier Transform from time to frequency, resulting in the dynamic structure factor $S(q,\omega)$ which is written to a newly created netCDF file. Note that the program is very similar to a serial version, with the exception of the `ParConstant` conversions, the only difference is in the iteration loop over the variable q.

```
from Scientific.BSP import ParConstant, ParFunction, ParIndexIterator
from Scientific.BSP.IO import ParNetCDFFile
from nMoldyn.calc import GaussianWindow, fft
import Numeric

fqt_file = ParNetCDFFile('fqt.nc', 'LENGTH', local_access=1)
sqw_file = ParNetCDFFile('sqw.nc', 'LENGTH', 'w')
alpha = 0.

t = fqt_file.variables['time']
q = fqt_file.variables['qlength']
```

```
f = fqt_file.variables['SF']

dt = t[1]-t[0]
domega = ParConstant(Numeric.pi)/(t.shape[0]*dt)

sqw_file.createDimension('LENGTH', q.shape[0])
sqw_file.createDimension('OMEGA', t.shape[0])
sqw_file.createVariable('omega', Numeric.Float, ('OMEGA',))
sqw_file.createVariable('qlength', Numeric.Float, ('LENGTH',))
sqw_file.createVariable('DSF', Numeric.Float, ('LENGTH', 'OMEGA'))

sqw_file.variables['omega'][:] = domega*ParFunction(Numeric.arange)(t.shape[0])
sqw_file.variables['qlength'][:] = q[:]

s = sqw_file.variables['DSF']

def do_fft(fqt):
    return fft(GaussianWindow(fqt, alpha)).real[:len(fqt)]
do_fft = ParFunction(do_fft)

for i in ParIndexIterator(f):
    s[i] = do_fft(f[i])

fqt_file.close()
sqw_file.close()
```

References

[1] G. Hains and F. Loulergue, "Functional Bulk Synchronous Parallel Programming using the BSMLlib Library", CMPP'2000 International Workshop on Constructive Methods for Parallel Programming, S. Gorlatch ed., Ponte de Lima, Portugal, Fakultät für Mathematik und Informatik, Univ. Passau, Germany, Technical Report, July, 2000