PMI - Parallel Method Invocation

Olaf Lenz (lenzo@mpip-mainz.mpg.de) – *Max Planck Institute for Polymer Research, Postfach 3148, D-55021 Mainz* Germany

The Python module "pmi" (Parallel Method Invocation) is presented. It allows users to write simple, non-parallel Python scripts that use functions and classes that are executed in parallel.
The module is well suited to be employed by other modules and packages that want to provide functions that are executed in parallel. The user of such a module does not have to write a parallel script, but can still profit from parallel execution.

Introduction

All modern CPUs provide more than one core, so that they can execute several tasks in parallel. Still, most software, and in particular self-written scripts, do not exploit this capability. The reason for this is mainly, that parallel programming is significantly harder than writing serials programs.

In particular in a scientific environment, one often has to use computationally intensive functions that could greatly benefit from parallelism. Still, scientific packages (like SciPy) usually provide only serial implementations of these functions, as providing a parallel function would mean that the user would have to call it from a parallel program.

When parallelizing a program, one usually has the choice between two fundamental models: the shared-memory thread model and the distributed-memory message-passing model [Bar07] .

In the shared-memory thread model, the different parallel threads have access to shared variables that they can use to exchange information. Programming in the world of threads is relatively simple, however, it is also relatively simple to produce problems like deadlocks and race conditions, or to create highly inefficient code, as all communication between the threads happens implicitly via the shared variables. Furthermore, the model can only be employed efficiently on machines that provide shared memory that can be accessed by all threads.

In the distributed-memory message-passing model, the parallel tasks (or processes) all have an independent data space. To exchange information, the tasks have to use explicit message-passing functions. Writing message-passing programs is more tedious than writing threaded programs, as all communication has to be made explicit. On the other hand, message-passing programs are less error-prone, as each task has its own, independent data space, so that certain kinds of deadlocks and race conditions can not happen. Another advantage of using message-passing parallelization is that it can be used on distributed-memory machines, but that it can also be easily and efficiently mapped to shared-memory platforms, while it is not possible to map threaded programs onto distributed-memory machines.

Therefore, when using a shared-memory machine, the choice between the models is mostly a choice of the programming style, but it does not influence how an algorithm is parallelized. Many algorithms, in particular in scientific computing where large datasets are processed, are algorithms that can profit from data parallelism. In data-parallel programs, the different tasks do not execute completely independent operations, instead each task does the same operation, but on a different piece of data. In particular, most control structures and the program logic is the same in all tasks. This can greatly simplify writing parallel programs [Boy08].

Data-parallel programming can be used in both parallelization models. In the world of C/C++ and FORTRAN, data-parallel programming in the shared-memory thread model is supported by the fork-join programming model of OpenMP [OpenMP] [CDK01] . In this model, the programmer marks regions of the program to be executed in parallel, while the rest of the program is running in a serial fashion. At the beginning of this region, the flow of control is forked, and the code of the region is executed on all threads in parallel. If not marked otherwise, all variables are shared, so that all threads can access the same data, and can work on their piece of data. At the end of the parallel region, all threads are joined back, and it returns to a single flow of control. Using OpenMP is relatively simple and has a few nice properties. In particular, it is easy to synchronize the threads and to implement control structures and the program logic, as these can be implemented in serial fashion, so that only the time-consuming parts have to be parallelized. On the other hand, in the parallel regions, the programmer is exposed to all dangers of threaded programming, like race conditions, deadlocks and inefficient access to the shared memory.

Data-parallel programming in the message-passing model is supported by the standardized library MPI (Message Passing Interface) [MPI] [MPIF09]. Each parallel task runs completely independent, however MPI provides advanced communication operations that allows the tasks to explicitly communicate. Writing data-parallel algorithms in this framework is somewhat more tedious, as it requires a lot of explicit communication, in particular when it comes to implementing the program logic and control structures. On the other hand, it is not so easy to fall into the traps of parallel programming, like producing inefficient code or race conditions.

When writing data-parallel programs, it would be good if one could combine at least some of the advantageous features of both programming models. To this end, it helps to understand that the fork-join model of OpenMP makes it simple to implement control structures and the program logic is independent of the underlying parallel programming model. The notion of parallel regions could be used in both the shared-memory thread model, where the threads have access to shared variables, as well as in the distributed-memory message-passing model, where the different tasks can exchange messages. However, to the author's best knowledge, combining the fork-join model with message-passing for data-parallel programming has not been done so far.

The PMI module

In the course of the ESPResSo++ [ESPResSo] project, the module `pmi` (Parallel Method Invocation) [PMI] has been developed. It is a pure Python module that tries to combine the fork-join model with message-passing. Like this, it allows the user to write the program logic and control structures in the fashion of OpenMP, while it still uses the less error-prone MPI for the actual parallelization. The only requirement of the module is a working MPI module (for example mpi4py [mpi4py] or boostmpi [boostmpi].

PMI does not go so far to allow a programmer to simply mark certain regions of the program to be run in parallel. Instead, it allows users to call - from a serial script - arbitrary Python functions to be executed in parallel (e.g. on multicore CPUs or on large parallel machines). Once called, the different invocations of the function can communicate via MPI. When the function returns, the flow of control is returned to the serial script. Furthermore, PMI allows to create parallel object instances, that have a corresponding instance in all parallel tasks, and to call arbitrary methods in these objects.

PMI has two possible areas of use: on the one hand, it allows modules or packages to provide parallel functions and classes. A user can call these from a simple, apparently serial Python script, that in fact runs parallel code, without the user having to care about parallelization issues. On the other hand, PMI could be used within a GUI that is used to control parallel code.

Other than modules that base on thread parallelization, scripts using PMI and MPI can be used on multicore machines as well as on convenience clusters with fast interconnects and big high-performance parallel machines.

When comparing PMI to the standard multithreading or multiprocessing modules, it must be stressed that PMI has all the advantages that message-passing parallelization has over thread-parallelization: it can work on both shared-memory as well as on distributed memory-machines, and it provides the less error-prone parallelization approach.

When comparing PMI to using the pure MPI modules or other message-passing solutions (like PyPar [PyPar]), it has the advantage that it doesn't require the programmer to write a whole parallel script to use a parallel function. Instead, only those functions that actually can use parallelism have to be parallelized. PMI allows a user to hide the parallelism in those functions that need it.

To the best knowledge of the author, the only other solution that provides functionality comparable to PMI are the parallel computing facilities of IPython [IPython]. Using these, it would be possible to write parallel functions that can be called from a serial script. Note, however, that IPython is a significantly larger package and the parallel facilities have a number of strong dependencies. These dependencies make it hard to run it on some more exotic high-performance platforms like the IBM Blue Gene, and prohibit its use within simple libraries.

Parallel function calls

Within PMI, the task executing the main script is called the *controller*. On the controller, the pmi commands `call()`, `invoke()` or `reduce()` can be called, which will execute the given function on all workers (including the task running the controller itself). The three commands differ only in the way they handle return values of the called parallel functions. Furthermore, the command `exec_()` allows to execute arbitrary Python code on all workers.

In the following example, we provide the outline of the module `mandelbrot_pmi` that contains a function to compute the Mandelbrot fractal in parallel:

```
import pmi
# import the module on all workers
pmi.exec_('import mandelbrot_pmi')

# This is the parallel function that is
# called from mandelbrot()
def mandelbrot_parallel((x1, y1), (x2, y2),
                        (w, h), maxit):
   '''Compute the local slice of the
      mandelbrot fractal in parallel.'''
   # Here we can use any MPI function.
     .
     .
     .

# This is the serial function that can be
# called from a (serial) user script
def mandelbrot(c1, c2, size, maxit):
   return pmi.call(
      'mandelbrot_pmi.mandelbrot_parallel',
      c1, c2, size, maxit)
```

A user can now easily write a serial script that calls the parallelized function `mandelbrot`:

```
import pmi, mandelbrot_pmi
# Setup pmi
pmi.setup()
# Call the parallel function
M = mandelbrot_pmi.mandelbrot(
  (-2.0, -1.0), (1.0, 1.0),
  (300, 200), 127)
```

Parallel class instances

`pmi.create()` will create and return a parallel instance of a class. The methods of the class can be invoked via `call()`, `invoke()` or `reduce()`, and when the parallel instance on the controller is used as an argument to one of these calls, it is automatically translated into the corresponding instance on the worker. Taking the following definition of the class `Hello` in the module `hello`:

```
from mpi4py import MPI
class Hello(object):
  def __init__(self, name):
    self.name = name
    # get the number of the parallel task
    self.rank = MPI.COMM_WORLD.rank
  def printmsg(self):
    print("Hello %s, I'm task %d!" %
          (self.name, self.rank))
```

Now, one could write the following script that creates a parallel instance of the class and call its method:

```
import pmi
pmi.setup()
pmi.exec_('import hello')
hw = pmi.create('hello.Hello', 'Olaf')
pmi.call(hw, 'printmsg')
```

This in itself is not very useful, but it demonstrates how parallel instances can be created and used.

Parallel class instance proxies

To make it easier to use parallel instances of a class, PMI provides a metaclass `Proxy`, that can be used to create a serial frontend class to a parallel instance of the given class. Using the metaclass, the module `hello_pmi` would be defined as follows:

```
import pmi
from mpi4py import MPI
pmi.exec_('import hello_pmi')

# This is the class to be used in parallel
class HelloParallel(object):
  def __init__(self, name):
    self.name = name
    self.rank = MPI.COMM_WORLD.rank
  def printmsg(self):
    print("Hello %s, I'm task %d!" %
          (self.name, self.rank))

# This is the proxy of the parallel class,
# to be used in the serial script
class Hello(object):
  __metaclass__ = pmi.Proxy
  pmiproxydefs = \
    dict(cls = 'HelloParallel',
         pmicall = [ 'printmsg' ])
```

Given these definitions, the parallel class could be used in a script:

```
import pmi, hello_pmi
pmi.setup()
hello = hello_pmi.Hello('Olaf')
hello.printmsg()
```

Summary

The PMI module provides a way to call arbitrary functions and to invoke methods in parallel. Using it, modules and packages can provide parallelized functions and classes to their users, without requiring the users to write error-prone parallel script code.

References

[PMI] http://www.espresso-pp.de/projects/pmi/

[Bar07] B. Barney, *Introduction to Parallel Computing*, Lawrence Livermore National Laboratory, 2007, http://www.llnl.gov/computing/tutorials/parallel_comp/

[Boy08] C. Boyd, *Data-parallel computing*, ACM New York, NY, USA, 2008, http://doi.acm.org/10.1145/1365490.1365499

[OpenMP] http://openmp.org/wp/

[CDK01] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon, *Parallel Programming in OpenMP*, Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2001

[MPI] http://www.mcs.anl.gov/research/projects/mpi/

[MPIF09] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 2.2*, High Performance Computing Center Stuttgart, Germany, 2009, http://www.mpi-forum.org/docs/docs.html

[ESPResSo] http://www.espresso-pp.de

[mpi4py] http://mpi4py.scipy.org/

[boostmpi] http://mathema.tician.de/software/boostmpi

[PyPar] M. Cieślik and C. Mura, *PaPy: Parallel and distributed data-processing pipelines in Python*, in Proc. SciPy 2009, G. Varoquaux, S. van der Walt, J. Millman (Eds), pp. 17–24, http://sourceforge.net/projects/pypar/

[IPython] F. Perez and B. Granger: *Ipython, a system for interactive scientific computing*, Computing in Science & Engineering, 9(3), 21-29, 2007 http://ipython.scipy.org/doc/stable/html/parallel/index.html