Building a parallel program step-by-step

Ole Nielsen
School of Mathematical Sciences
Australian National University, Canberra
THE AUSTRALIAN NATIONAL UNIVERSITY
Ole.Nielsen@anu.edu.au

April 29, 2002

Introduction

This text demonstrates the development of a simple Python MPI program for use in the ANU, SMS Summer school 2002. It is based entirely on 'Python Crash Course' and 'Maple Crash Course' and it will take you through a parallel version of the code for computing primes and prime pairs which we looked at previously. The demo is given in its entirety in Section 6 and is also available in the common area `/short/c23/Python_demos/` under the name `paraprime_demo.py` in case you want to peek at the final solution. The output from a run on four processors is given in Section 7.

1 Getting ready

First go to your local directory `Python_demos` where the parallel library is located. Write `ls` and verify that the files `pypar.py` and `mpi.so` are present. If not, copy everything from `/short/c23/Python_demos` to your local directory or ask one of the tutors what to do.

2 Identifying the processors

It is almost impossible to demonstrate parallelism with an interactive session at the command prompt as we did with the Python and Maple tutorials. Therefore we must write a *program* that can be executed in parallel. Start an editor of your choice with a new file (called `paraprime.py`, say). For example:

```
emacs paraprime.py &
```

Write the following in the file and save it

```
import pypar

numproc = pypar.size()
myid =    pypar.rank()

print "I am proc %d of %d" %(myid, numproc)
```

Then type the following at the Unix prompt to execute the program normally on one processor

```
python paraprime.py
```

You should see the following output

```
> python paraprime.py
I am proc 0 of 1
```

Now let us try and run the same program on four processors:

```
> prun -n 4 python paraprime.py
I am proc 0 of 4
I am proc 1 of 4
I am proc 2 of 4
I am proc 3 of 4
```

If you get a message from each of the four processors stating their id as above, we know that everything works. We have a parallel program!

Next define the interval in which we will compute the prime pairs and compute the *local* intervals on each processor by adding the following to your file:

```
lower = 100
upper = 1001
interval = upper-lower

myinterval = interval/numproc
mylower = lower + myid*myinterval
if myid == numproc-1:
  myupper = upper
else:
  myupper = mylower + myinterval - 1
```

The interval is divided among the processors using integer division, then based on the local id `myid` we determine a local lower and upper bound. The last statement takes into account that the interval length may not be a multiple of the number of processors - any left overs are assigned to the last processor (`myid == numproc-1`).

Processor	0	1	2	3
Interval	$[100, 324]$	$[325, 549]$	$[550, 774]$	$[775, 1001]$

Let us print out the assigned work. Add to your file:

```
print 'Processor %d has interval [%d, %d]' %(myid, mylower, myupper)
```

Running the program should produce the following output:

```
> prun -n 4 python paraprime.py
I am proc 1 of 4
Processor 1 has interval [325, 549]
I am proc 0 of 4
Processor 0 has interval [100, 324]
I am proc 2 of 4
Processor 2 has interval [550, 774]
I am proc 3 of 4
Processor 3 has interval [775, 1001]
```

Now we create the Maple command for computing primes and prime pairs in the given interval and call up Maple to execute it. This is essentially the same codes we developed in the Python and the Maple crash courses. Add to your file

```
# Get primes in local interval from Maple
#
maple_cmd = """
  primes := select(isprime, [seq(i,i=%d..%d)]):
  pripai := select(pn -> nextprime(pn)-pn = 2, primes):
  print(pripai):
  """ %(mylower, myupper)

from popen2 import popen2
maple_out, maple_in = popen2('maple -q')
maple_in.write(maple_cmd)
maple_in.close()

mypairs = maple_out.read()
print 'Processor %d has computed prime pairs %s' %(myid, mypairs)
```

Now try to run the program again. Every processor should write the first element of each prime pair it has identified. All that remains is to collect all data on processor 0 (the Master). This is where communication will take place.

First we write the code for the Master (Processor 0):

```
if myid == 0:
  print 'Start collecting the results on the Master'

  pairs = mypairs.strip()
  pairs = pairs[:-1]           # Remove trailing ']'
  for i in range(1,numproc):
    pairs = pairs + ', ' + pypar.receive(i)

  pairs = pairs + ']'

  print 'Final result: %s' %pairs
```

First the local pairs are assigned to the variable `pairs` which is used to aggregate the final result. Then we loop through all the slaves $i \in (1, 2, \ldots, (\texttt{numproc} - 1)$ and receive their local results using the message passing command `receive`.

All that remains is to add the code for the slaves. It is simply

```
else:
  mypairs = mypairs.strip()
  pypar.send(mypairs[1:-1], 0)   #Remove '[' and ']' and send to master
```

In other words slaves all send their local results to processor 0 – the Master. Now try to run the code.

Note how we use Pythons string method `strip()` to remove leading and trailing whitespace and indexing to make the final string look nice as we did in the Python crash course. As an exercise you can remove the lines with `.strip()` or change the indexing. What happens ?

3 Other examples

If you have more time then take a look at the code `mastslav` in the directory `MSummer`. Instead of just collecting the results on the Master it uses Maple again for postprocessing and generates a histogram of the distribution of prime pairs. Also take a look at the codes `master`, `slave`, `ncpus`, and `irank` in the director `NSummer`. These codes implement almost the same algorithm but using files and environment variables instead of message passing. This is simpler in some sense as it does *not* depend on the MPI library and can be very useful if communication speed is not too critical.

4 Timing of communication parameters

In the common area `/short/c23/Python_demos` you will find a code called `pytiming` which estimates the bandwidth, i.e. how many Megabytes are transmitted across the network with MPI. It also estimates roughly the *latency*, i.e. the time it takes to initiate a communication before any data are sent. To run it type

```
qsub runpytiming
```

You can also run the equivalent C program as `qsub runctiming` and compare. What is the difference ?

5 Strategies for efficient parallel programming

A final note on efficiency.

Parallel programming should lead to faster execution and/or ability to deal with larger problems.

The ultimate goal is to be P times faster with P processors. However *speedup* is usually less than P. One must address three critical issues to achieve good speedup:

- **Interprocessor communication:** The amount and the frequency of messages should be kept as low as possible. Ensure that processors have plenty of work to do between communications.
- **Data distribution and load balancing:** If some processors finish much sooner than others, the total execution time of the parallel program is bounded by that of the slowest processor and we say that the program is poorly load balanced. Ensure that each processor get its fair share of the work load.
- **Sequential parts of a program:** If half of a program, say, is inherently sequential the speedup can never exceed 2 no matter how well the remaining half is parallelised. This is known as Amdahls law. Ensure that the all cost intensive parts get parallelised.

Hope you enjoyed the demo !

6 The entire code

```
#!/bin/env python
############################################################
#
#  ANU Summer School in Computational Mathematics 2002
#
#  Example: Implementing Master-Slave parallelism in Python using MPI.
#
#  Author:  Ole Nielsen,  SMS, ANU, Jan. 2002.
#
############################################################
#
# The purpose of this code is to demonstrate how Python can be
# used in parallel using MPI.
#
# This demo uses Maple to compute prime pairs in parallel.
# Each processor computes prime pairs in different sub interval
```

```
# and results are collected on processor 0 to get prime pairs
# in the full interval.
#
# To execute on Alpha server:
#
#   prun -n 4 paraprime_demo.py
#



#
# Initialisation
#
import pypar  # The Python-MPI interface

numproc = pypar.size()
myid =    pypar.rank()

print "I am proc %d of %d" %(myid, numproc)



# Setup interval and sub-intervals
#
lower = 100
upper = 1001

interval = upper-lower

myinterval = interval/numproc
mylower = lower + myid*myinterval
if myid == numproc-1:
  myupper = upper
else:
  myupper = mylower + myinterval - 1

print 'Processor %d has interval [%d, %d]' %(myid, mylower, myupper)


# Get primes in local interval from Maple
```

```
#
maple_cmd = """
  primes := select(isprime, [seq(i,i=%d..%d)]):
  pripai := select(pn -> nextprime(pn)-pn = 2, primes):
  print(pripai):
  """ %(mylower, myupper)

from popen2 import popen2
maple_out, maple_in = popen2('maple -q')
maple_in.write(maple_cmd)
maple_in.close()

mypairs = maple_out.read()
print 'Processor %d has computed prime pairs %s' %(myid, mypairs)

# Collect results on Master (proc 0)
#
if myid == 0:
  print 'Start collecting the results on the Master'

  pairs = mypairs.strip()
  pairs = pairs[:-1]            #Remove trailing ']'
  for i in range(1,numproc):
    pairs = pairs + ', ' + pypar.receive(i)

  pairs = pairs + ']'

  print 'Final result: %s' %pairs
else:
  mypairs = mypairs.strip()
  pypar.send(mypairs[1:-1], 0)  #Remove '[' and ']' and send to master
```

7 Output

```
> prun -n 4 paraprime_demo.py
I am proc 0 of 4
Processor 0 has interval [100, 324]
```

```
I am proc 1 of 4
Processor 1 has interval [325, 549]
I am proc 3 of 4
Processor 3 has interval [775, 1001]
I am proc 2 of 4
Processor 2 has interval [550, 774]
Processor 2 has computed prime pairs [569, 599, 617, 641, 659]


Processor 0 has computed prime pairs [101, 107, 137, 149, 179,...


Start collecting the results on the Master
Processor 3 has computed prime pairs [809, 821, 827, 857, 881]


Processor 1 has computed prime pairs [347, 419, 431, 461, 521]


Final result: [101, 107, 137, 149, 179, 191, 197, 227, 239, 269,...
```