Distributed data analytics (2016-2017)
Information Systems and Machine Learning Lab
University of Hildesheim

Exercise Sheet 4

Mario Rodríguez Ruiz

May 11, 2017

Contents

List of Figures

List of Tables

1 Explain your system

	Home System
Machine	Asus Notebook ROG G60Jx
Operating System	Windows 10 Pro 64-bit
CPU	Intel Core i7 720QM @1.60GHz
Number of cores	4
Number of threads	8
RAM	16GB @665MHz (9-9-9-24)
Programming language version Python	v3.6.1:69c0db5 64 bit
Programming language version Java	v1.8

Table 1.1: My system

2 Calculating Pi using collective communication

2.1 How you assign tasks to different processes?

Each process calculates its own calculation limits depending on the total calculation number and the number of processes that exist.

```
# Calculate calculation limits for each process
def limits():
  # Number of calculations for each process
  n_calc = int(infinity/size)
  # Remainder of the division of the calculations of each process
  rest = int(infinity % size)

  # If there is a remainder, it is divided into
  # one unit per process until do all allotted.
  if rest != 0:
    # If rank is smaller than rest, it is that
    # part of the rest still has to be distributed.
    if rank < rest :
      # Whenever an additional unit is assigned to a process, both the
      # start position and the end position of the other processes will be affected.
      n_calc = n_calc + 1
      if rank == 0:
        ini = 0
      else:
        ini = rank*n_calc
    # At this moment all the remainder has already been distributed.
    else:
      ini = rank*n_calc+rest
```

```
  else:
    ini = rank*n_calc
  end = ini + n_calc
  return ini,end
```

Once each process has calculated its calculation limits, they make calls to the function from the beginning of the range (**i**) to the end of it (**end**).

```
# Bailey-Borwein-Plouffe formula
def bbp(i):
  eightI = 8*i
  sixteen = (Decimal(1)/(16**i))
  one = (Decimal(4)/(eightI+1))
  four = (Decimal(2)/(eightI+4))
  five = (Decimal(1)/(eightI+5))
  six = (Decimal(1)/(eightI+6))
  return sixteen*(one-four-five-six)

def calc_pi(i , end):
  pi = Decimal(0)
  for i in range(i, end):
    pi += bbp(i)
  return pi
```

2.2 How you combine results from all the processes?

To combine all the results, I've used **AllReduce** (Figure 2.1). Reduces values on all processes to a single value onto all processes.

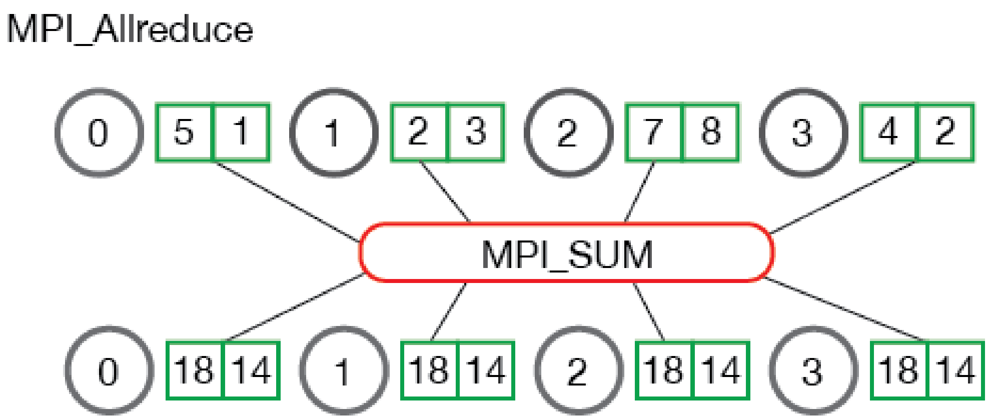

Figure 2.1: AllReduce Example

```
# Calculation limits for each process
ini, end = limits()

# User alterable precision (infinity)
getcontext().prec = (int(infinity))
pi = calc_pi(ini, end)
total_pi = comm.allreduce(pi, op = MPI.SUM)
```

2.3 Provide runtime analysis on varying number of processes

2.3.1 The value of ∞ set as 10^2

Processes	Time(s)
1	0,161
4	0,102
8	0,063
16	0,049
32	0,042

Table 2.1: Results for processes-time for the value of ∞ set as 10^2

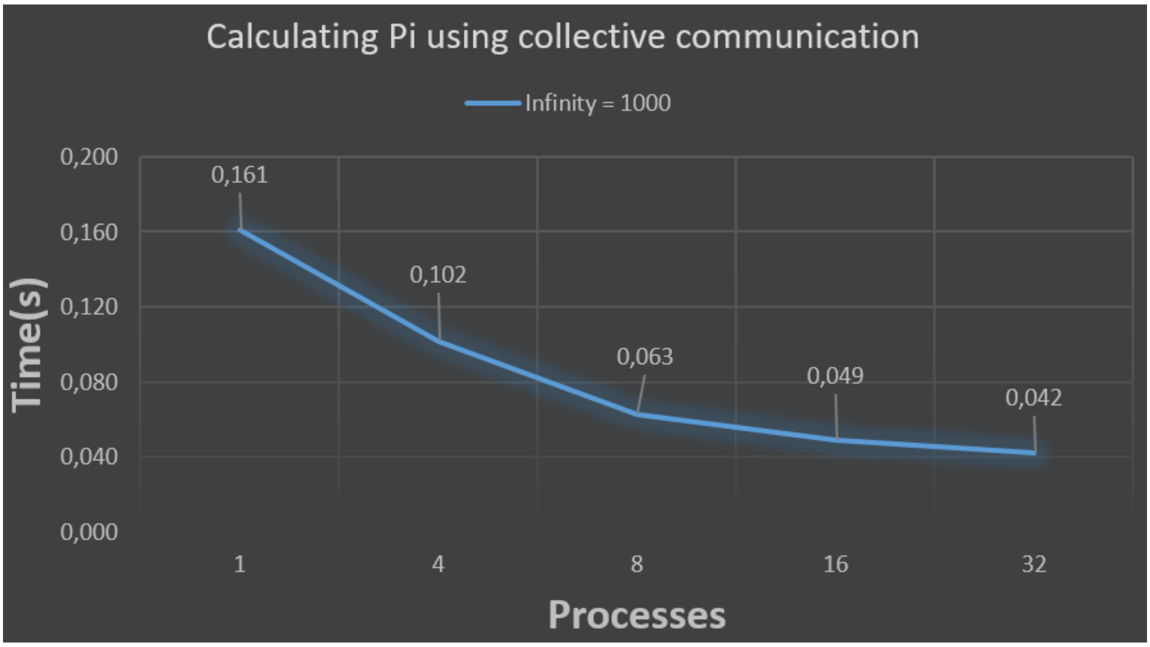

Figure 2.2: Combine results for the value of ∞ set as 10^2

It can be seen in Table 2.2 how results improve with respect to time as the number of processes running in parallel increases.

2.3.2 The value of ∞ set as 10^3

Processes	Time(s)
1	103,252
4	63,580
8	41,622
16	31,234
32	28,616

Table 2.2: Results for processes-time for the value of ∞ set as 10^3

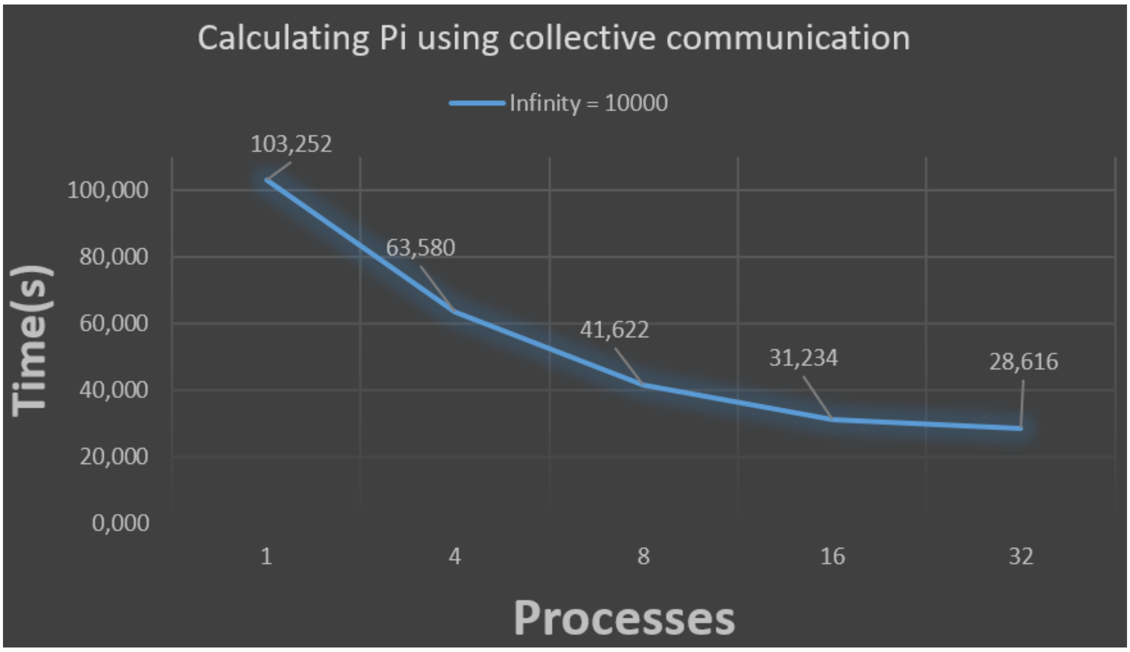

Figure 2.3: Combine results for the value of ∞ set as 10^3

It can be seen in Table 2.3 how results improve with respect to time as the number of processes running in parallel increases.

2.3.3 Run program

It is possible to store all decimals of Pi in a file, since when it is indicated a very high number the console is break. This option is deactivated by default, to verify its operation with small values and to see them directly in console.

```
if rank== 0:
  MPI.Finalize()
  #      outfile = open('texto.txt', 'w')
  #      outfile.write(str(total_pi))
  #      print("Pi:\t", total_pi)
  print("Sum processes time:\t{}\t{}".format(size, (totalTime/
     size)))
```

3 Matrix - matrix multiplication using collective communication

3.1 How you assign tasks to different processes?

To make a correct division of labor, the dimension of the matrix must be a **multiple** of the number of **processes**.

```
SIZE_ROWS = 3
ROWS = int(numpy.ceil(SIZE_ROWS/size)*size)
```

The division of labor will be done with respect to the columns of the first matrix (**matrix_A**) and of processes size.

```
size = comm.Get_size()
n_calc = int(ROWS/size)
```

Matrix to scatter communication (matrix_A chunks)

```
chunk_A = numpy.zeros((n_calc,COLS),dtype='i')
```

Broadcast of second matrix (matrix_B, the same data to all processes) because only matrix_A will be partitioned.

```
comm.Bcast([matrix_B,MPI.INT])
```

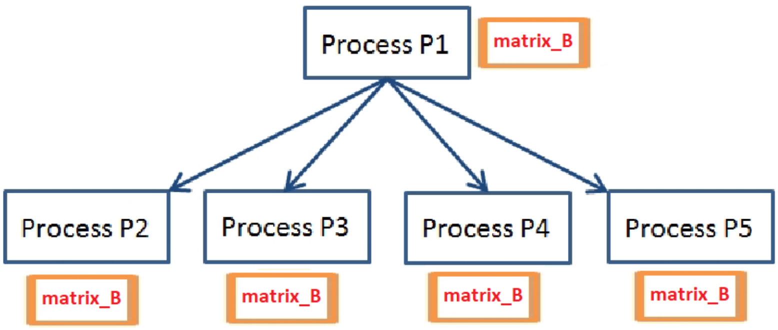

Figure 3.1: Broadcast of second matrix

Scatter of matrix_A (matrix_A chunks of data to each process)

```
comm.Scatter([matrix_A,MPI.INT],[chunk_A,MPI.INT])
```

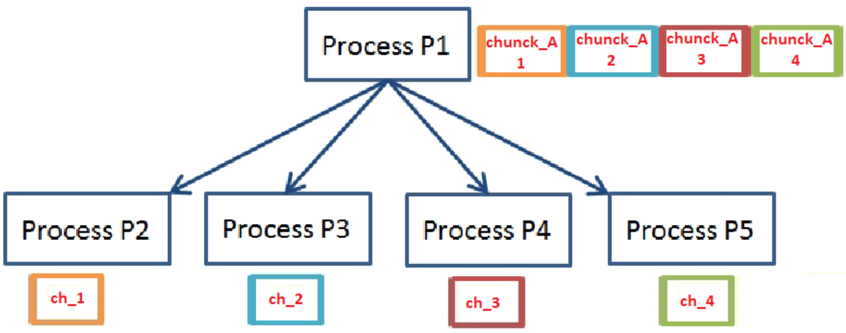

Figure 3.2: Scatter of first matrix

3.2 How you combine results from all the processes?

Once all the work has been distributed, each process performs its part of the multiplication, saving the result in an auxiliary matrix.

```
chunk_C = numpy.dot(chunk_A,matrix_B)
```

To combine all the results, I've used **Gather** (Figure 3.3). All processes send their auxiliary matrix to a root process that collects the data received and save it in the result matrix (**matrix_C**).

```
comm.Gather([chunk_C,MPI.INT],[matrix_C,MPI.INT])
```

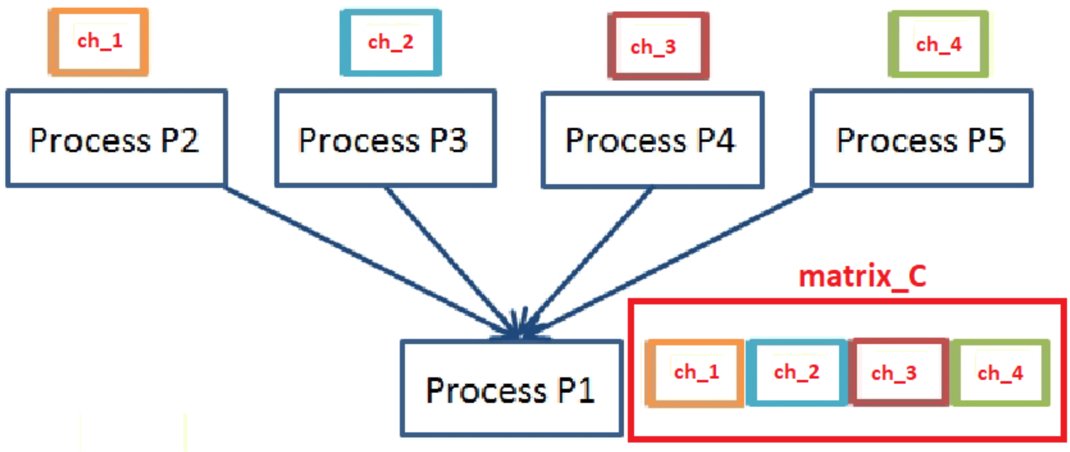

Figure 3.3: Gather of result matrix

3.3 Provide runtime analysis on varying number of processes

3.3.1 First run

$matrix\_A = 960 * 960$
$matrix\_B = 960 * 960$
$matrix\_A * B = 960 * 960 = matrix\_C$

Processes	**Time(seconds)**
1	13,582
4	6,029
8	5,678
16	4,868
32	4,808

Table 3.1: First run results time per processes

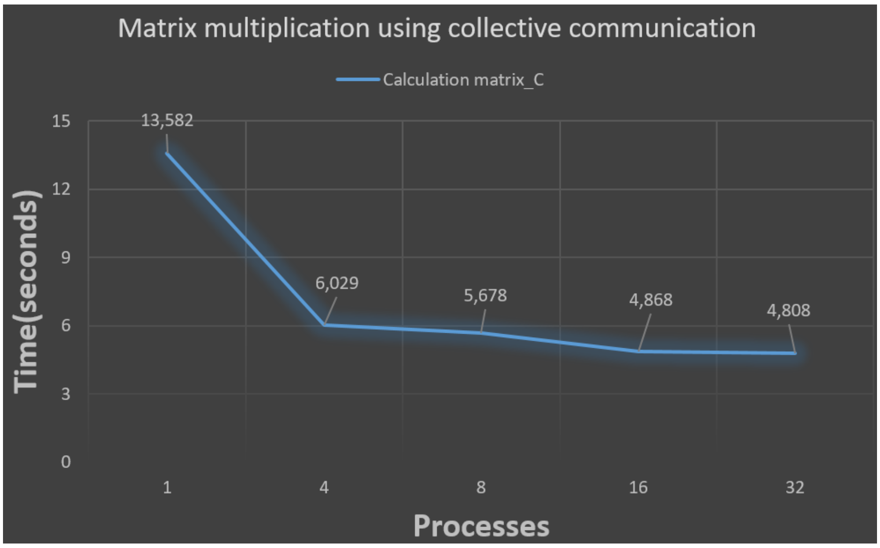

Figure 3.4: Combine first run results

3.3.2 Second run

$matrix\_A = 1920 * 960$
$matrix\_B = 960 * 1920$
$matrix\_A * B = 1920 * 1920 = matrix\_C$

Processes	**Time(seconds)**
1	58,746
2	34,593
4	25,280
16	21,060
32	20,809

Table 3.2: Second run results time per processes

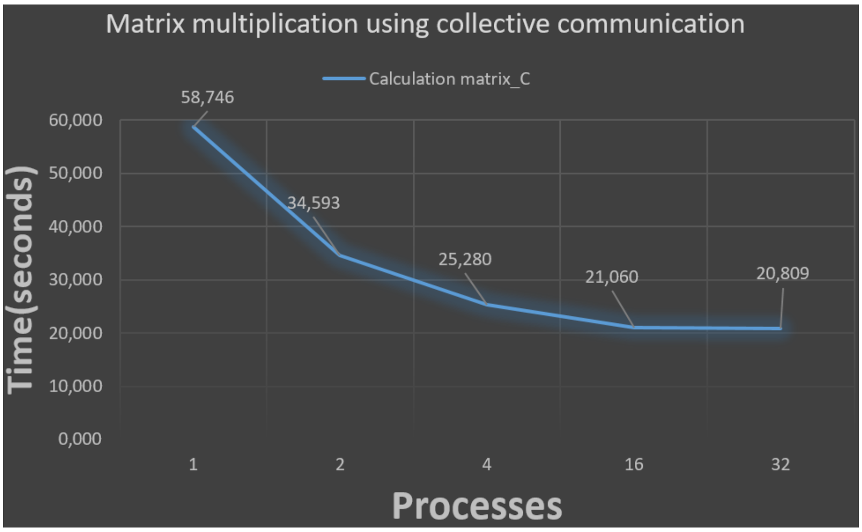

Figure 3.5: Combine second run results

3.3.3 Third run

$matrix\_A = 3840 * 1920$
$matrix\_B = 1920 * 3840$
$matrix\_A * B = 3840 * 3840 = matrix\_C$

Processes	**Time(seconds)**
1	569,995
2	352,940
4	263,712
16	197,030
32	194,124

Table 3.3: Third run results time per processes

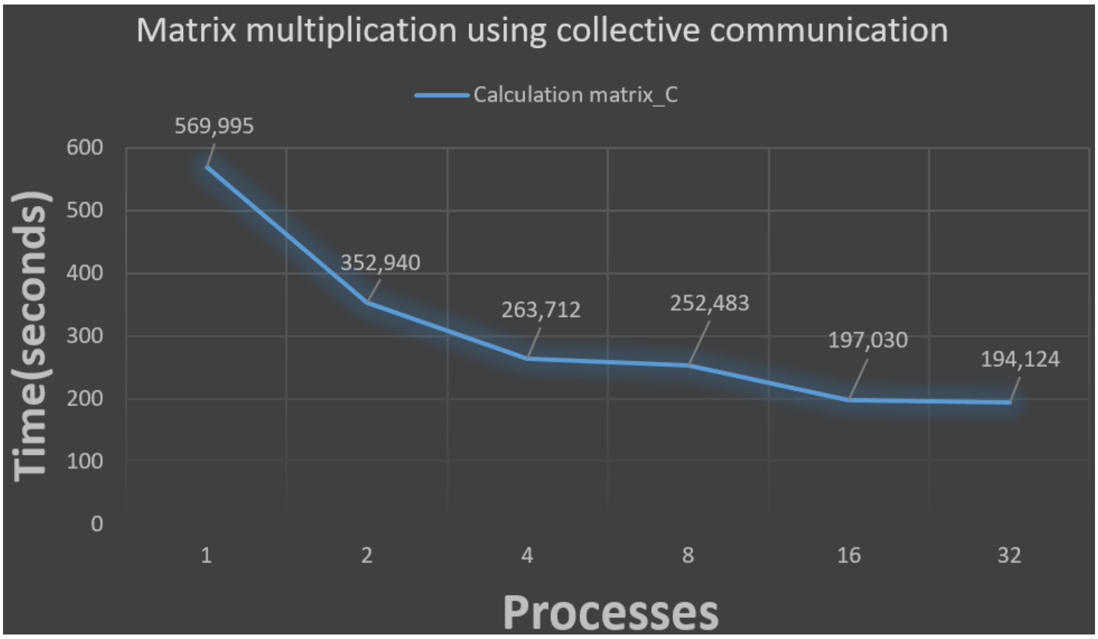

Figure 3.6: Combine third run results

3.3.4 Conclusions

It can be seen in Figure 3.6, Figure 3.5 and Figure 3.4 how results improve with respect to time as the number of processes running in parallel increases.

In all three cases it is coincided that the two contiguous points in which the greatest time difference is greatest is from one process to two processes.
On the other hand, also in the three cases it is coincided that the smallest decrease of time occurs from sixteen processes to thirty-two processes.