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

Exercise Sheet 1

Mario Rodríguez Ruiz

April 21, 2017

Contents

1 Exercise 0: Explain your system **3**

2 Exercise 1: Basic Parallel Vector Operations with Threading/process **3**

List of Figures

List of Tables

1.1 My system 3
2.1 Threads Python 3
2.2 Multiproccesing in python 6
2.3 Add two vectors and store results in a third vector. 8
2.4 Find a minimum number in a vector. 10
2.5 Find an average of numbers in a vector. 12

1 Exercise 0: 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 Exercise 1: Basic Parallel Vector Operations with Threading/process

The experiments I started to program in Python: First with multithreads and later with multiprocesses. In the first case, I took time to realize that the Global Interpreter Lock (GIL) limited the optimization of the execution time or, better said, did not improve anything.

__Minium Number__			
Size	**Result**	**Run (s)**	**Threads**
10000000.0	9.35299581117e-08	2.2600271701812744	1
10000000.0	9.35299581117e-08	2.2070693969726562	2
10000000.0	9.35299581117e-08	2.32605242729187	3
10000000.0	9.35299581117e-08	2.2390549182891846	4
10000000.0	9.35299581117e-08	2.377060651779175	5
10000000.0	9.35299581117e-08	2.430034637451172	6
10000000.0	9.35299581117e-08	2.34808087348938	7
10000000.0	9.35299581117e-08	2.427046537399292	8

Table 2.1: Threads Python

This is because this system does not allow the execution of two threads concurrently in Python. This happened when I already had all the code written, so I looked for information to be able to deactivate the GIL.

```
1 # Add two vectors and store results in a third vector
2 def AddVectors(v1, v2):
3   v3 = []
4   i = 0
```

```
5     for i in range(len(v1)):
6       v3.append(v1[i]+v2[i])
7     return v3
8
9   # Find a minimum number in a vector
10  def MiniumNumber(v, pos_ini, pos_fin):
11    global RES_MIN
12    for pos_ini in range(pos_ini, pos_fin):
13      if v[pos_ini] < RES_MIN:
14        lock.acquire()
15        RES_MIN = v[pos_ini]
16        lock.release()
17
18  # Find an average of numbers in a vector
19  def Average(v, i, f):
20    global RES_AV
21    av = 0
22    for i in range(f):
23      av += v[i]
24
25    lock.acquire()
26    RES_AV += av/(f-i)
27    lock.release()
```

The only thing I found was that IronPython and Jython are free of it, so I proceeded with their installation to not miss all the work I had already done.

```
1     # ------------------- EXPERIMENT b)
          ------------------------------
2
3     # initial nt = 1
4     for nt in range(nt, NUM_TH+1):
5       siz = len(v2)/nt        # work size for each thread
6       rem_tam = len(v2)%nt    # remainder of division
7       RES_MIN = v2[0]         # initialize minium result
8       pos_ini = 0
9       pos_fin = 0
10
11      start = time.time()
12      for i in range(nt):
13        pos_fin = int(pos_ini+siz)  # final position is the sum
              of initial position and work size
14        t = threading.Thread(target=MiniumNumber, args=(v2,
```

```
            pos_ini, pos_fin))
        pos_ini = pos_fin # update initial position for next
            thread
        t.start()
        t.join()

    # When remainder is distint of zero
    if rem_tam > 0:
    for i in range(rem_tam):
        pos_fin = pos_ini+1
        t = threading.Thread(target=MiniumNumber, args=(v2,
            pos_ini, pos_fin))
        pos_ini = pos_fin
        t.start()
        t.join()

    end = time.time()
    tim = end-start
```

After trying to compile with each one appeared a lot of errors, so I lost more time still.

```
# initial nt = 1
for nt in range(nt, NUM_TH+1):
  siz = len(v2)/nt        # work size for each thread
  rem_tam = len(v2)%nt    # remainder of division
  RES_MIN.value = v2[0]         # initialize minium result
  pos_ini = 0
  pos_fin = 0

start = time.time()
for i in range(nt):
  pos_fin = int(pos_ini+siz)  # final position is the sum of
      initial position and work size
  t = multiprocessing.Process(target=MiniumNumber, args=(v2,
      pos_ini, pos_fin, RES_MIN))
  pos_ini = pos_fin # update initial position for next thread
  t.start()
  t.join()

# When remainder is distint of zero
if rem_tam > 0:
  for i in range(rem_tam):
    pos_fin = pos_ini+1
```

```
    t = multiprocessing.Process(target=MiniumNumber, args=(v2
        , pos_ini, pos_fin, RES_MIN))
    pos_ini = pos_fin
    t.start()
    t.join()

end = time.time()
tim = end-start
```

Tired of the situation, I decided to switch to multiprocessing (also in Python). I made the appropriate changes and ran my program again. The results not only did not improve the execution time, but made it worse.

__Minium Number__			
Size	**Result**	**Run (s)**	**Threads**
10000000.0	4.506043649321612e-09	13.562278509140015	1
10000000.0	4.506043649321612e-09	14.489362716674805	2
10000000.0	4.506043649321612e-09	15.591330289840698	3
10000000.0	4.506043649321612e-09	14.982396841049194	4
10000000.0	4.506043649321612e-09	15.688313961029053	5
10000000.0	4.506043649321612e-09	17.651407957077026	6
10000000.0	4.506043649321612e-09	17.330454349517822	7
10000000.0	4.506043649321612e-09	16.745417594909668	8

Table 2.2: Multiproccesing in python

It was here when I thought about switching to Java, a decision I had to make a lot earlier and it would have saved me a lot of time.

```
for (int nt = 1; nt <= NUM_TH; nt++)
{
  siz = v1.size()/nt ;       // work size for each thread
  rem_tam = v1.size()%nt ;   // remainder of division
  pos_ini = 0 ;
  pos_fin = 0 ;
  ArrayL resA = new ArrayL(v1.size()) ;

  start = System.currentTimeMillis() ;
  for (int i = 0; i < nt; i++)
  {
    pos_fin = pos_ini+siz ;  // final position is the sum of
        initial position and work size
```

```
13      AddTwoVectors ch = new AddTwoVectors(v1, v2, pos_ini,
            pos_fin);
14      ch.start() ;
15      ch.join();
16      resA.AddValorsInArray(ch.getResult(), pos_ini, pos_fin) ;
17      pos_ini = pos_fin ;// update initial position for next
            thread
18    }
19    //  When remainder is distint of zero
20    if (rem_tam > 0){
21      for (int j = 0; j < rem_tam; j++){
22        pos_fin = pos_ini+1 ;
23        AddTwoVectors ch1 = new AddTwoVectors(v1, v2, pos_ini,
              pos_fin);
24
25        ch1.start() ;
26        ch1.join();
27        resA.AddValorsInArray(ch1.getResult(), pos_ini, pos_fin
              ) ;
28        pos_ini = pos_fin ;
29      }
30    }
31
32    end = System.currentTimeMillis() ;
33    time = (end-start);
34    v3 = resA.getResult() ;
35  }
```

For this case I have distributed the loop in proportional parts for each thread. Here, locks were not required since each strand only added the specific components to the output vector, which did not interfere with the calculation of the other threads.

SizeVector	NumThreads	Time(ms)
10000	1	8
10000	2	5
10000	3	9
10000	4	6
10000	5	8
10000	6	7
10000	7	7
10000	8	4
100000	1	35
100000	2	18
100000	3	8
100000	4	7
100000	5	5
100000	6	10
100000	7	12
100000	8	8
1000000	1	75
1000000	2	54
1000000	3	40
1000000	4	51
1000000	5	32
1000000	6	81
1000000	7	47
1000000	8	46

Table 2.3: Add two vectors and store results in a third vector.

```
1  start = System.currentTimeMillis() ;
2  for (int i = 0; i < nt; i++)
3  {
4    pos_fin = pos_ini+siz ;  // final position is the sum of
       initial position and work size
5    MiniumNumber ch = new MiniumNumber(v1, pos_ini, pos_fin,
       min_res);
6    pos_ini = pos_fin ;// update initial position for next
       thread
7    ch.start() ;
8    ch.join();
9
10   // Critical section
11   lock.lock();
```

```
12    min_res = ch.getResult() ;
13    lock.unlock();
14  }
15  //  When remainder is distint of zero
16  if (rem_tam > 0){
17    for (int j = 0; j < rem_tam; j++){
18      pos_fin = pos_ini+1 ;
19      MiniumNumber ch1 = new MiniumNumber(v1, pos_ini, pos_fin,
            min_res);
20      pos_ini = pos_fin ;
21      ch1.start() ;
22      ch1.join();
23
24      // Critical section
25      lock.lock();
26      min_res = ch1.getResult() ;
27      lock.unlock();
28    }
29  }
30  end = System.currentTimeMillis() ;
31  time = (end-start);
```

In this case, a lock has been required. The lock has been placed in a critical area in which an important variable is modified. In the case that two strands were to modify that variable at a time, the results would not be correct at the end of the execution.

SizeVector	MiniumNumber	NumThreads	Time(ms)
1000000	-999	1	19
1000000	-999	2	12
1000000	-999	3	6
1000000	-999	4	5
1000000	-999	5	4
1000000	-999	6	7
1000000	-999	7	8
1000000	-999	8	6
10000000	-999	1	41
10000000	-999	2	36
10000000	-999	3	31
10000000	-999	4	29
10000000	-999	5	33
10000000	-999	6	34
10000000	-999	7	33
10000000	-999	8	34
100000000	-999	1	321
100000000	-999	2	297
100000000	-999	3	299
100000000	-999	4	300
100000000	-999	5	299
100000000	-999	6	302
100000000	-999	7	294
100000000	-999	8	303

Table 2.4: Find a minimum number in a vector.

```
1  start = System.currentTimeMillis() ;
2  for (int i = 0; i < nt; i++)
3  {
4    pos_fin = pos_ini+siz ;  // final position is the sum of
       initial position and work size
5    Average ch = new Average(v1, pos_ini, pos_fin, avg);
6    pos_ini = pos_fin ;// update initial position for next
       thread
7    ch.start() ;
8    ch.join();
9
10   // Critical section
11   lock.lock();
12   final_avg += ch.getResult() ;
```

```
13    lock.unlock();
14  }
15  //  When remainder is distint of zero
16  if (rem_tam > 0){
17    for (int j = 0; j < rem_tam; j++){
18      pos_fin = pos_ini+1 ;
19      Average ch1 = new Average(v1, pos_ini, pos_fin, avg);
20      pos_ini = pos_fin ;
21      ch1.start() ;
22      ch1.join();
23
24      // Critical section
25      lock.lock();
26      final_avg += ch1.getResult() ;
27      lock.unlock();
28    }
29  }
30  end = System.currentTimeMillis() ;
```

In this case, a lock has been required. The lock has been placed in a critical area in which an important variable is modified. In the case that two strands were to modify that variable at a time, the results would not be correct at the end of the execution.

SizeVector	Average	NumThreads	Time(ms)
1000000	0.260164	1	18
1000000	0.260164	2	11
1000000	0.260164	3	6
1000000	0.26016400000000006	4	4
1000000	0.26016400000000006	5	4
1000000	0.260164	6	6
1000000	0.260164	7	6
1000000	0.26016399999999995	8	6
10000000	-0.160883	1	47
10000000	-0.160883	2	38
10000000	-0.160883	3	31
10000000	-0.16088300000000003	4	32
10000000	-0.16088300000000003	5	33
10000000	-0.16088300000000003	6	34
10000000	-0.16088299999999997	7	33
10000000	-0.160883	8	34
100000000	-0.02936205	1	301
100000000	-0.029362049999999997	2	266
100000000	-0.02936205	3	266
100000000	-0.029362049999999994	4	273
100000000	-0.029362050000000004	5	260
100000000	-0.029362050000000004	6	255
100000000	-0.029362049999999994	7	269
100000000	-0.02936205	8	265

Table 2.5: Find an average of numbers in a vector.