

PC-2019/20 Course Project: Parallel Image Reader and 2D Pattern Recognition Asynchronous Execution

Alberto Baldrati

`alberto.baldrati@stud.unifi.it`

April 2020

Outline

Introduction

- Preview

- Technologies

Image Reader

- Implementation

Pattern Recognition

- Algorithm

- Trivial Example

Full Execution

- Implementations

Experimental results

- Image Reader results

- Full Execution results

Conclusions

Preview

- ▶ We will focus on multi threading and performance comparison between asynchronous and synchronous execution
- ▶ In the first part we will develop a simple Image Reader which reads all the images in a directory
- ▶ Later, with the aid of this Image Reader, we will apply the 2D Pattern Recognition algorithm to the read images.
- ▶ Such concatenation of operations has been conducted in synchronous and asynchronous manner for enabling a comparison between implementations

Technologies

- ▶ We have used the **C++17** which includes *Filesystem* library that allows us to manage files without relying on external libraries
- ▶ For reading and writing on the images **OpenCV** library has been used
- ▶ We have used the **Boost thread library** which provides useful features such as: **thread pool** and **then** operation on future. (Not included in standard C++ library)
- ▶ For all the synchronous implementations **OpenMP** library has been used

Image Reader implementations

Algorithm 1: *Sync* ImageReader

Data: inputDir, numThreads Nt**Result:** imagesVector

```
1 #pragma omp parallel for
  schedule(static)
  num_threads(numThreads)
2 for imageName in inputDir do
3   image =
    cv::imread(imageName)
4   #pragma omp critical
5   imagesVector.push_back(
    (image,imageName))
6 return imagesVector
```

Algorithm 2: *Async* ImageReader

Data: inputDir, ThreadPool Tp**Result:** imagesVector

```
1 futImagesVector futIV = []
2 for imageName in inputDir do
3   lambdaImage =
    []{return cv::imread(imageName)}
4   futIV.pus_back(
    (boost::async(Tp,lambdaImage),
    imageName))
5 for futImage in futIV do
6   imagesVector.push_back(
    (futImage[0].get(), futImage[1]))
7 return imagesVector
```

Pattern Recognition algorithm

Let's briefly recall the 2D Pattern Recognition algorithm

Algorithm 3: computeSAD

Data: queryMatrix Q,
targetMatrix T,
startRowIndex i,
startColIndex j

Result: localSadValue

```

1 localSadValue = 0
2 for from k = 0 to Q.rows do
3   for from l = 0 to Q.cols do
4     targetV = T[i+k][j+l]
5     queryV = Q[k][l]
6     localSadValue +=
      | targetV - queryV |
7 return localSadValue

```

Algorithm 4: PatternRecognition

Data: queryMatrix Q,
targetMatrix T

Result: SADMatrix S

```

1 Define SADMatrix S
2 S.rows = T.rows - Q.rows + 1
3 S.cols = T.cols - Q.cols + 1
4 for from i = 0 to S.rows do
5   for from j = 0 to S.cols do
6     S[i][j] =
      | computeSAD(P,Q,i,j)
7 cx, cy = argmin(S)

```

Trivial example

1	2	1	3
2	1	2	4
0	1	2	3

Target Matrix

0	3
2	1

Query Matrix

1	2	1	3
2	1	2	4
0	1	2	3

compute SAD[1][1]

1	2	1	3
2	1	2	4
0	1	2	3

compute SAD[1][2]

1	2	1	3
2	1	2	4
0	1	2	3

compute SAD[1][3]

1	2	1	3
2	1	2	4
0	1	2	3

compute SAD[2][1]

1	2	1	3
2	1	2	4
0	1	2	3

compute SAD[2][2]

1	2	1	3
2	1	2	4
0	1	2	3

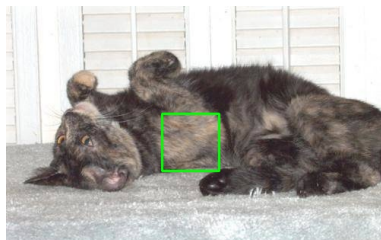
compute SAD[2][3]

2	6	4
6	4	5

SADMatrix

Full Execution

- ▶ We chain together the image reader and the 2D Pattern Recognition
- ▶ The image reader reads all the images from a known input directory
- ▶ The pattern recognition algorithm is applied to each image with a random query matrix and we draw a rectangle to identify the closest portion to that query



Full Execution algorithms

Algorithm 5: Sync Full Execution

Data: inputDir, numThreads Nt, queryMatrix Q, outputDir

```

1 imagesVector =
  SyncImageReader(inputDir,Nt)
2 #pragma omp parallel for schedule(static)
  num_threads(numThreads)
3 for image, imageName in imagesVector do
4   cx, cy = PatternRecognition(Q, image)
5   topLeftPoint tlp = (cx,cy)
6   bottomRightPoint brp =
    (cx + Q.cols, cy + Q.rows)
7   cv::rectangle(image, tlp, brp, green)
8   cv::imwrite(outputDir + imageName,
    image)
  
```

Algorithm 6: Async Full Execution

Data: inputDir, ThreadPool Tp, queryMatrix Q, outputDir

```

1 futImagesVector =
  AsyncImageReader(inputDir,Tp)
2 futuresTask = []
3 for futImage, imageName in futImagesVector do
4   futuresTask.push_back(
    futImage.then(Tp,[Q])(futImage){
5     image = futImage.get()
6     cx,cy = PatternRecognition(Q,image)
7     topLeftPoint tlp = (cx, cy)
8     bottomRightPoint brp =
      (cx + Q.cols, cy + Q.rows)
9     cv::rectangle(image, tlp, brp, green)
10    return image, ImageName}
11   ).then(Tp,[outputDir](futOutput){
12     image = futOutput.get()
13     cv::imwrite(outputDir +
14       imageName, image) })))
15
16 for futTask in futuresTask do
17   futTask.wait();
  
```

Full Execution algorithms

Algorithm 7: Sync Full Execution

Data: inputDir, numThreads Nt, queryMatrix Q, outputDir

```

1 imagesVector =
  SyncImageReader(inputDir,Nt)
2 #pragma omp parallel for schedule(static)
  num_threads(numThreads)
3 for image, imageName in imagesVector do
4   cx, cy = PatternRecognition(Q, image)
5   topLeftPoint tlp = (cx,cy)
6   bottomRightPoint brp =
     (cx + Q.cols, cy + Q.rows)
7   cv::rectangle(image, tlp, brp, green)
8   cv::imwrite(outputDir + imageName,
     image)
  
```

Algorithm 8: Async Full Execution

Data: inputDir, ThreadPool Tp, queryMatrix Q, outputDir

```

1 futImagesVector =
  AsyncImageReader(inputDir,Tp)
2 futuresTask = []
3 for futImage, imageName in futImagesVector do
4   futuresTask.push_back(
     futImage.then(Tp,[Q])(futImage){
5     image = futImage.get()
6     cx,cy = PatternRecognition(Q,image)
7     topLeftPoint tlp = (cx, cy)
8     bottomRightPoint brp =
       (cx + Q.cols, cy + Q.rows)
9     cv::rectangle(image, tlp, brp, green)
10    return image, ImageName}
11   ).then(Tp,[outputDir](futOutput){
12     image = futOutput.get()
13     cv::imwrite(outputDir +
14       imageName, image) })))
16 for futTask in futuresTask do
17   futTask.wait();
  
```

Full Execution algorithms

Algorithm 9: Sync Full Execution

Data: inputDir, numThreads Nt, queryMatrix Q, outputDir

```

1 imagesVector =
  SyncImageReader(inputDir,Nt)
2 #pragma omp parallel for schedule(static)
  num_threads(numThreads)
3 for image, imageName in imagesVector do
4   cx, cy = PatternRecognition(Q, image)
5   topLeftPoint tlp = (cx,cy)
6   bottomRightPoint brp =
     (cx + Q.cols, cy + Q.rows)
7   cv::rectangle(image, tlp, brp, green)
8   cv::imwrite(outputDir + imageName,
     image)

```

Algorithm 10: Async Full Execution

Data: inputDir, ThreadPool Tp, queryMatrix Q, outputDir

```

1 futImagesVector =
  AsyncImageReader(inputDir,Tp)
2 futuresTask = []
3 for futImage, imageName in futImagesVector do
4   futuresTask.push_back(
     futImage.then(Tp,[Q])(futImage){
5     image = futImage.get()
6     cx,cy = PatternRecognition(Q,image)
7     topLeftPoint tlp = (cx, cy)
8     bottomRightPoint brp =
       (cx + Q.cols, cy + Q.rows)
9     cv::rectangle(image, tlp, brp, green)
10    return image, ImageName}
11   ).then(Tp,[outputDir])(futOutput){
12     image = futOutput.get()
13     cv::imwrite(outputDir +
14       imageName, image) })))
16 for futTask in futuresTask do
17   futTask.wait();

```

Full Execution algorithms

Algorithm 11: Sync Full Execution

Data: inputDir, numThreads Nt, queryMatrix Q, outputDir

```

1 imagesVector =
  SyncImageReader(inputDir,Nt)
2 #pragma omp parallel for schedule(static)
  num_threads(numThreads)
3 for image, imageName in imagesVector do
4   cx, cy = PatternRecognition(Q, image)
5   topLeftPoint tlp = (cx,cy)
6   bottomRightPoint brp =
     (cx + Q.cols, cy + Q.rows)
7   cv::rectangle(image, tlp, brp, green)
8   cv::imwrite(outputDir + imageName,
     image)
  
```

Algorithm 12: Async Full Execution

Data: inputDir, ThreadPool Tp, queryMatrix Q, outputDir

```

1 futImagesVector =
  AsyncImageReader(inputDir,Tp)
2 futuresTask = []
3 for futImage, imageName in futImagesVector do
4   futuresTask.push_back(
     futImage.then(Tp,[Q](futImage){
5     image = futImage.get()
6     cx,cy = PatternRecognition(Q,image)
7     topLeftPoint tlp = (cx, cy)
8     bottomRightPoint brp =
       (cx + Q.cols, cy + Q.rows)
9     cv::rectangle(image, tlp, brp, green)
10    return image, ImageName}
11   ).then(Tp,[outputDir](futOutput){
12     image = futOutput.get()
13     cv::imwrite(outputDir +
14       imageName, image) })))
15
16 for futTask in futuresTask do
17   futTask.wait();
  
```

Full Execution algorithms

Algorithm 13: Sync Full Execution

Data: inputDir, numThreads Nt, queryMatrix Q, outputDir

```

1 imagesVector =
  SyncImageReader(inputDir,Nt)
2 #pragma omp parallel for schedule(static)
  num_threads(numThreads)
3 for image, imageName in imagesVector do
4   cx, cy = PatternRecognition(Q, image)
5   topLeftPoint tlp = (cx,cy)
6   bottomRightPoint brp =
     (cx + Q.cols, cy + Q.rows)
7   cv::rectangle(image, tlp, brp, green)
8   cv::imwrite(outputDir + imageName,
     image)
  
```

Algorithm 14: Async Full Execution

Data: inputDir, ThreadPool Tp, queryMatrix Q, outputDir

```

1 futImagesVector =
  AsyncImageReader(inputDir,Tp)
2 futuresTask = []
3 for futImage, imageName in futImagesVector do
4   futuresTask.push_back(
     futImage.then(Tp,[Q])(futImage){
5     image = futImage.get()
6     cx,cy = PatternRecognition(Q,image)
7     topLeftPoint tlp = (cx, cy)
8     bottomRightPoint brp =
       (cx + Q.cols, cy + Q.rows)
9     cv::rectangle(image, tlp, brp, green)
10    return image, ImageName}
11   ).then(Tp,[outputDir](futOutput){
12     image = futOutput.get()
13     cv::imwrite(outputDir +
14       imageName, image) })))
16 for futTask in futuresTask do
17   futTask.wait();
  
```

Points of interest

- ▶ Synchronous execution uses OpenMP for enabling parallelism
- ▶ In Asynchronous execution we can highlight:
 - ▶ We need the *futuresTask* vector since at the end of the algorithm we need to wait that all images have been processed
 - ▶ Thanks to **then** function in line 4 and 12 we can fully decouple the task **I/O** bound from the task **CPU** bound.
 - ▶ The function **get** applied to the futures in line 5 and in line 13 is not blocking.

Equipment, metrics and profiling

- ▶ The tests have been conducted on an Ubuntu 18.04 LTS machine equipped with:
 - ▶ Intel Core i7-4790 3.6GHz with Turbo Boost up to 4Ghz, 4 core/8 thread processor
 - ▶ RAM 16 GB DDR4
 - ▶ Western Digital Blue 1TB Hard Disk 7200rpm
- ▶ The metrics used are execution time and SpeedUp S_P , it is calculated as

$$S_P = \frac{t_s}{t_p}$$

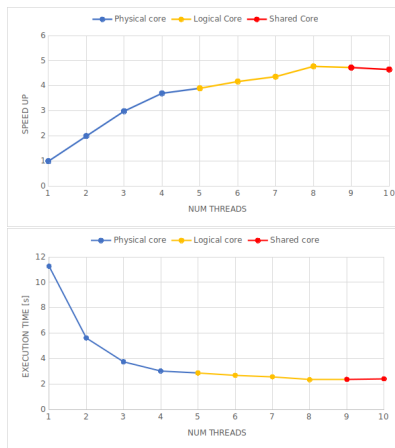
- ▶ The high precision C++11 library *chrono* has been used for measuring the execution time.

Tests

- ▶ Each time has been measured running each test 5 times and taking the average as a result
- ▶ All the images come from dogs vs cat Kaggle competition, which includes 37.5K (cats and dogs) images and has a size of 850MB
- ▶ For testing the image reader a subset of 25K images (the original training set) has been used
- ▶ For testing the Full Execution a subset of 5K images taken from the original test set has been used . The query matrix used has dimensions dimension 50×50 pixels.

Synchronous Image Reader results

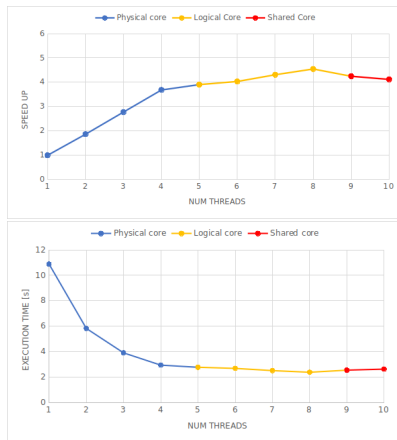
Test executed on 25K images with a total size of 582MB



Sync Image Reader		
Num threads	Execution time	SpeedUp
1	11,26s	1,00x
2	5,64s	2,00x
3	3,77s	2,99x
4	3,04s	3,70x
5	2,89s	3,90x
6	2,70s	4,17x
7	2,58s	4,36x
8	2,36s	4,77x
9	2,38s	4,73x
10	2,42s	4,65x

Asynchronous Image Reader results

Test executed on 25K images with a total size of 582MB



Async Image Reader		
Num threads	Execution time	SpeedUp
1	10,88s	1,00x
2	5,84s	1,86x
3	3,92s	2,78x
4	2,95s	3,68x
5	2,79s	3,91x
6	2,70s	4,03x
7	2,53s	4,31x
8	2,39s	4,55x
9	2,56s	4,25x
10	2,64s	4,12x

- Experimental results
 - Image Reader results

Image Reader comparison

Test executed on 25K images with a total size of 582MB

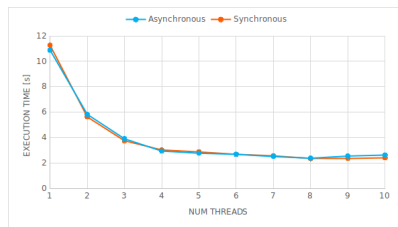
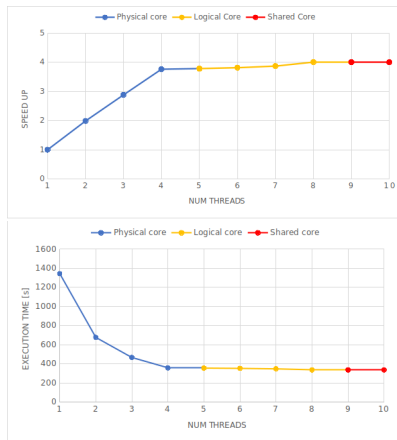


Image Reader Comparison		
Num threads	Asynchronous	Synchronous
1	10,88s	11,26s
2	5,84s	5,64s
3	3,92s	3,77s
4	2,95s	3,04s
5	2,79s	2,89s
6	2,70s	2,70s
7	2,53s	2,58s
8	2,39s	2,36s
9	2,56s	2,38s
10	2,64s	2,42s

- Experimental results
- Full Execution results

Synchronous Full Execution results

Test executed on 5K images with a query matrix 50×50 pixels

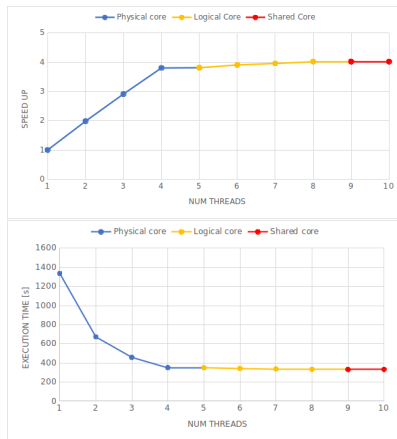


Sync Full Execution		
Num threads	Execution time	SpeedUp
1	1343,88s	1,00x
2	676,92s	1,99x
3	466,57s	2,88x
4	357,43s	3,76x
5	355,48s	3,78x
6	352,65s	3,81x
7	347,69s	3,87x
8	335,72s	4,00x
9	336,09s	4,00x
10	336,16s	4,00x

- Experimental results
- Full Execution results

Asynchronous Full Execution results

Test executed on 5K images with a query matrix 50×50 pixels

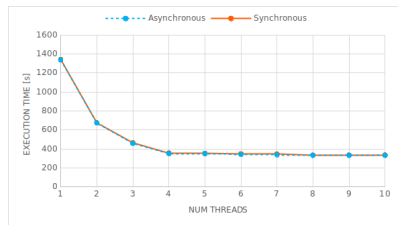


Async Full Execution		
Num threads	Execution time	SpeedUp
1	1336,63s	1,00x
2	672,91s	1,99x
3	458,49s	2,92x
4	351,09s	3,81x
5	350,43s	3,81x
6	341,90s	3,91x
7	337,35s	3,96x
8	332,10s	4,02
9	332,15s	4,02x
10	332,267s	4,02x

- Experimental results
- Full Execution results

Full Execution comparison

Test executed on 5K images with a query matrix 50×50 pixels



Full Execution Comparison		
Num threads	Asynchronous	Synchronous
1	1336,63s	1343,88s
2	672,91s	676,92s
3	458,49s	466,57
4	351,09s	357,43s
5	350,43s	355,48s
6	341,90s	352,65s
7	337,35s	347,69s
8	332,10s	335,72s
9	332,15s	336,09s
10	332,26s	336,16s

Conclusions

- ▶ Both implementations of the Image Reader achieve very similar execution time
- ▶ We have a **4.5x** SpeedUp in asynchronous implementation and a **4,7x** SpeedUp in the synchronous one
- ▶ Due to the CPU bound nature of the problem also in the Full Execution both implementations achieve very similar results, actually we have a little improvement in the asynchronous implementation
- ▶ Thanks to the parallel structure, in the Full Execution we have achieved a **4x** SpeedUp