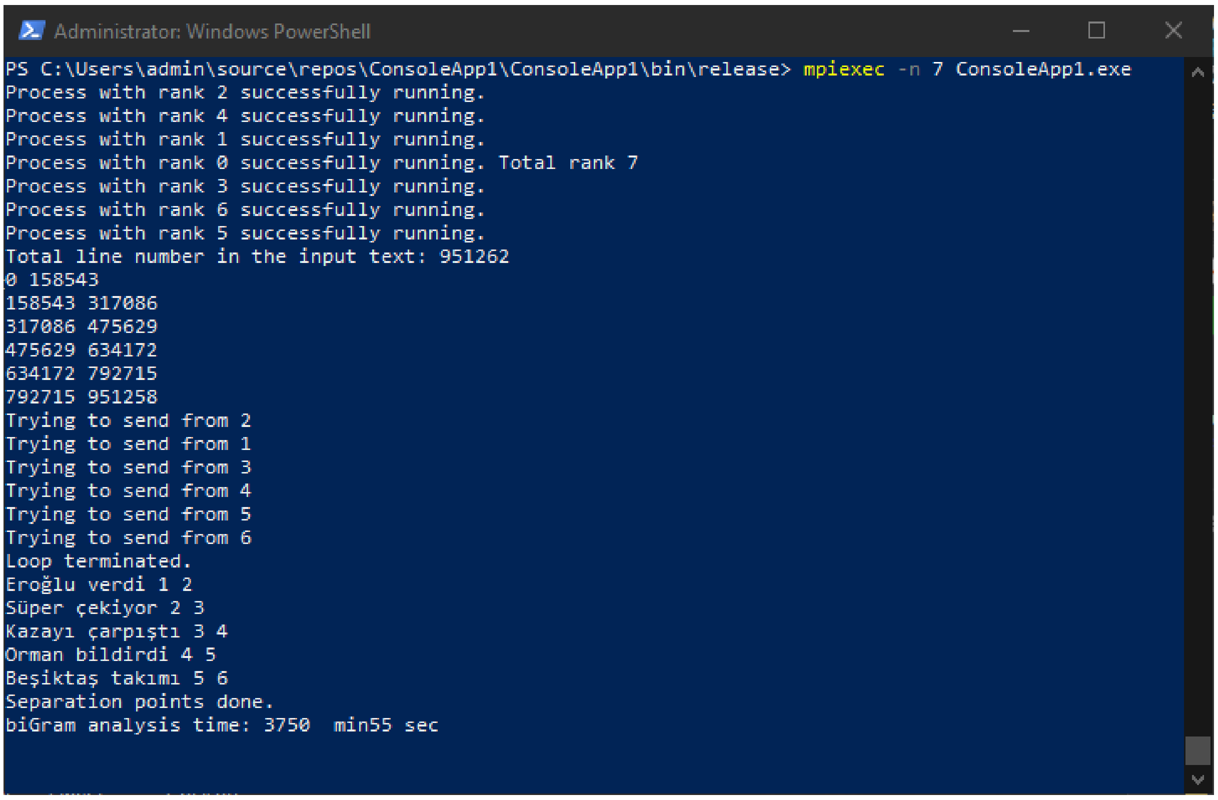
Distributed and Parallel Computing
2020 Spring - Final Project Report

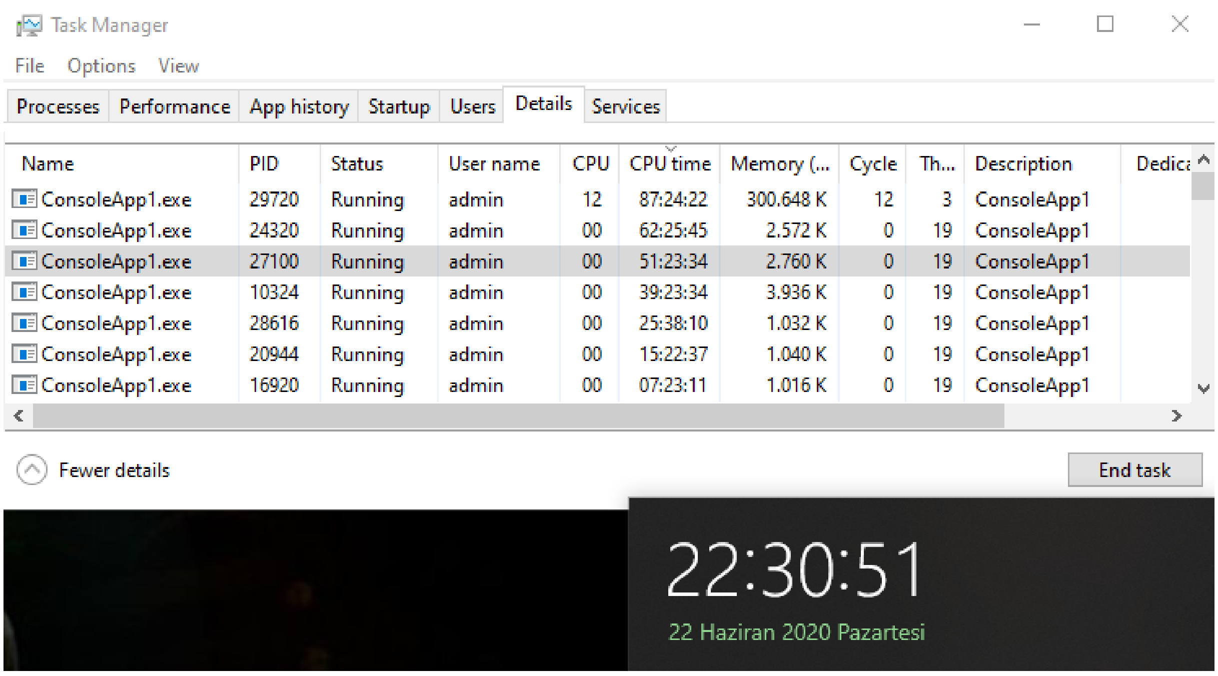
Project Topic: Bigram analysis in clustered computing environment by using MPI (Message passing interface).

Programming Language and Libraries: As programming language I used C# in order to keep coding simple, but I am not sure it was a good choice though. I used 3 external libraries besides standard ones. Microsoft's[1] [2] 'MPI' library for clustered computing, 'Nuve'[3] Natural Language Processing library for extracting bigrams in Turkish language and lastly a library called Easy.Common.Extensions[4] to count lines of text input with an effective performance.

Hardware Details and Execution Time: Execution was done on laptop that has got Intel 7700HQ 4 cores 8 threads CPU, 16 gigabyte RAM and a slow SSD. Total RAM usage of applications did not exceed approximately 400 mb and I never used hard drive prior to writing final output. Therefore, total execution time was completely based on CPU performance and the time was terrifyingly long. It has been running for 4 days and 4 hours. At the moment, last 'for' loop of 'counting sort' is being executed. If an external reason does not cause a failure hopefully, output text will be written to file in one day. Execution was made with 'release' version and compiler optimizations were active. 6 processes made bigram analysis and at the same time 1 process (master), received bigrams, counted same ones, and stored them. You may see last outputs printed to console and the situation of running processes on the task manager.



```
PS C:\Users\admin\source\repos\ConsoleApp1\ConsoleApp1\bin\release> mpiexec -n 7 ConsoleApp1.exe
Process with rank 2 successfully running.
Process with rank 4 successfully running.
Process with rank 1 successfully running.
Process with rank 0 successfully running. Total rank 7
Process with rank 3 successfully running.
Process with rank 6 successfully running.
Process with rank 5 successfully running.
Total line number in the input text: 951262
0 158543
158543 317086
317086 475629
475629 634172
634172 792715
792715 951258
Trying to send from 2
Trying to send from 1
Trying to send from 3
Trying to send from 4
Trying to send from 5
Trying to send from 6
Loop terminated.
Eroğlu verdi 1 2
Süper çekiyor 2 3
Kazayı çarpıştı 3 4
Orman bildirdi 4 5
Beşiktaş takımı 5 6
Separation points done.
biGram analysis time: 3750  min55 sec
```



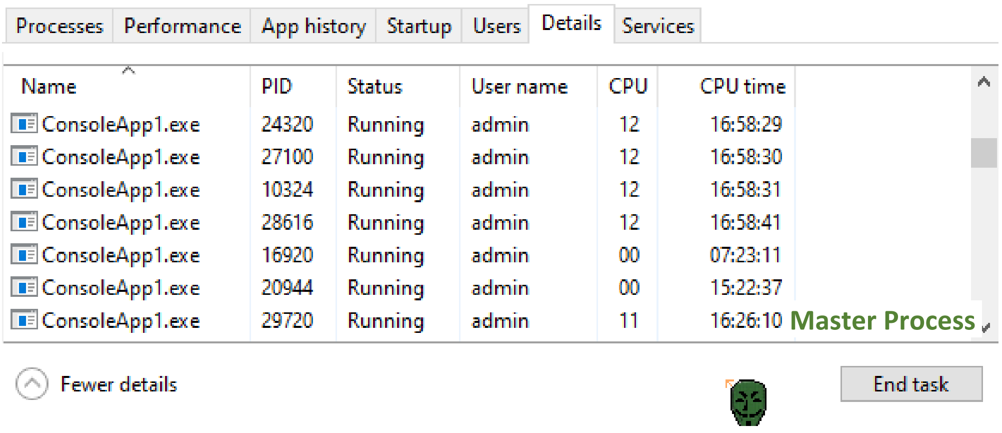
Name	PID	Status	User name	CPU	CPU time	Memory (...	Cycle	Th...	Description	Dedica
ConsoleApp1.exe	29720	Running	admin	12	87:24:22	300.648 K	12	3	ConsoleApp1	
ConsoleApp1.exe	24320	Running	admin	00	62:25:45	2.572 K	0	19	ConsoleApp1	
ConsoleApp1.exe	27100	Running	admin	00	51:23:34	2.760 K	0	19	ConsoleApp1	
ConsoleApp1.exe	10324	Running	admin	00	39:23:34	3.936 K	0	19	ConsoleApp1	
ConsoleApp1.exe	28616	Running	admin	00	25:38:10	1.032 K	0	19	ConsoleApp1	
ConsoleApp1.exe	20944	Running	admin	00	15:22:37	1.040 K	0	19	ConsoleApp1	
ConsoleApp1.exe	16920	Running	admin	00	07:23:11	1.016 K	0	19	ConsoleApp1	

Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
End task
22:30:51
22 Haziran 2020 Pazartesi

Design: Honestly, I could not realize that my design was not so good until I executed my program with given 160 mb input text file. During developing the program, I used a small input sample for testing. I never imagined execution would take hours with the actual input, therefore I kept my design very simple and later on had no time to change it. But I can say that I observed many things from this work.

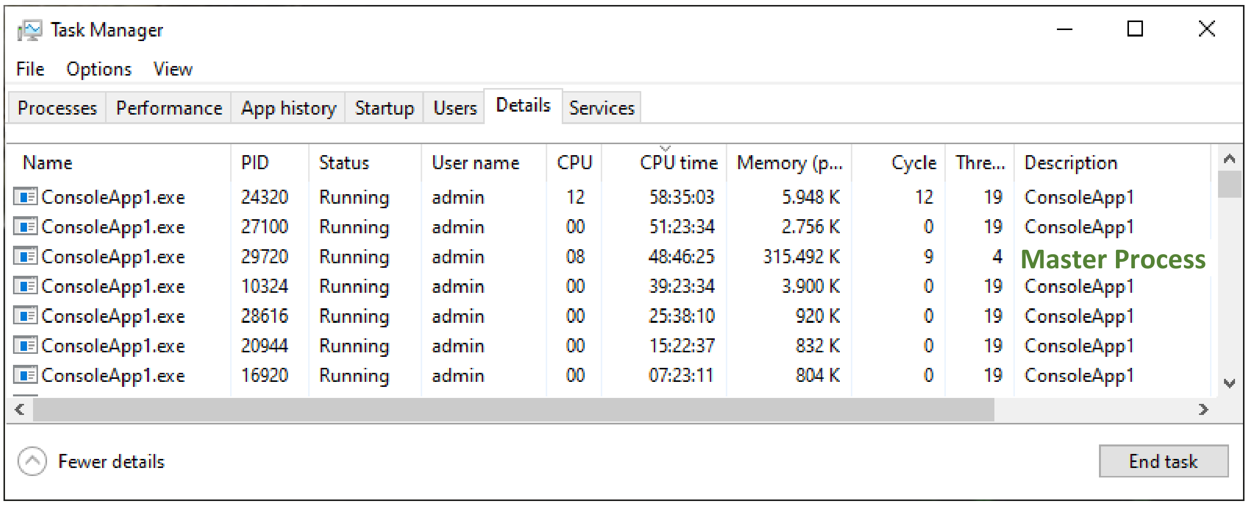
To mention about the design, I used All-To-One MPI design. There is one master process that makes everything excluding bigram analysis. Master process counts lines in the input file and informs other processes about intervals. Rest of processes make bigram analysis in the given intervals and send found bigrams consistently to the master process. Master process writes received bigrams to C#'s 'dictionary' data structure (with commonly known name HashMap). This data structure counts same bigrams and stores bigram and its count as mapped. In the end, I used 'counting sort'[5] algorithm (of course stable version) which has got an amazing performance *(I am not very sure anymore though* 😊*)* with inputs which contain many duplicated elements. *[O(n+k) linear complexity where n is number of elements and k is range of elements (in other words biggest value minus smallest value in the input elements)]*

Weaknesses of My Design: (1) At the first view to design, master process might seem like it may cause bottleneck. Because it has to receive bigrams from all computers in the cluster and write them by counting. But as you can see in the following screenshots, total CPU time of master process was always shorter *(by the time difference kept increasing up to 10 hours)* than processes doing bigram analysis. But obviously it will create bottleneck if there are many computers doing bigram analysis in the cluster.



Name	PID	Status	User name	CPU	CPU time
ConsoleApp1.exe	24320	Running	admin	12	16:58:29
ConsoleApp1.exe	27100	Running	admin	12	16:58:30
ConsoleApp1.exe	10324	Running	admin	12	16:58:31
ConsoleApp1.exe	28616	Running	admin	12	16:58:41
ConsoleApp1.exe	16920	Running	admin	00	07:23:11
ConsoleApp1.exe	20944	Running	admin	00	15:22:37
ConsoleApp1.exe	29720	Running	admin	11	16:26:10

Processes Performance App history Startup Users Details Services
Master Process
Fewer details
End task


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
End task

Name	PID	Status	User name	CPU	CPU time	Memory (p...	Cycle	Thre...	Description
ConsoleApp1.exe	24320	Running	admin	12	58:35:03	5.948 K	12	19	ConsoleApp1
ConsoleApp1.exe	27100	Running	admin	00	51:23:34	2.756 K	0	19	ConsoleApp1
ConsoleApp1.exe	29720	Running	admin	08	48:46:25	315.492 K	9	4	Master Process
ConsoleApp1.exe	10324	Running	admin	00	39:23:34	3.900 K	0	19	ConsoleApp1
ConsoleApp1.exe	28616	Running	admin	00	25:38:10	920 K	0	19	ConsoleApp1
ConsoleApp1.exe	20944	Running	admin	00	15:22:37	832 K	0	19	ConsoleApp1
ConsoleApp1.exe	16920	Running	admin	00	07:23:11	804 K	0	19	ConsoleApp1

But also dividing the work of master process into separated processes brings a big problem. Master process counts same elements when saving them. If this work is made on different processes, at the end merging their outputs will not be very cheap in terms of computation time too. Even if outputs will be created by binary insertion, and merging will be done from data structures holding elements as sorted, still many string comparisons will be needed to be done and will cost quite lot CPU time.

(2) Workload (in other words number of lines) is distributed equally without considering computing powers of computers in the cluster. If one computer in the cluster finishes its job much earlier than others, obviously it means a loss in computing power. Even on my own computer, there were quite big-time differences among execution duration of each process. Therefore, CPU utilization dropped under %100. Workload distribution way could have been different. For example, whole work will be divided into small pieces *(but also not too small, because creation and termination of processes are not very cheap too) (let's say a few ten thousand lines for that 1 million lines input).* Each computer connected to cluster will request a new job once it completes previously given job. With such design, we can benefit from computing power with maximum efficiency.

(3) Since I never thought, execution with given input would take terrifyingly long time, I did not consider writing a functionality that program saves the progress it makes to hard drive. Therefore, if anything goes wrong such as operating system failure, power cut etc. execution has to start from beginning.

Why were there too big-time differences among times each process finished: Honestly, I checked my code and the input file several times and I could not find anything that can be a reason related that issue. Input file seems quite homogeneous in terms of line lengths. I think, there could be several reasons behind of this. In the beginning of the execution, CPU utilization was %100. Maybe, at that time operating system or MPI framework did not distribute CPU cycles equally. Only this idea of course can not explain that big difference.

My other opinion which I believe it could be the actual reason, CPU design. As we know, CPUs have got optimized segments for some specific algorithms and operations i.e. for hashing, encrypting etc. But also, if you execute many hashing algorithms at once, optimized segment cannot handle with all of it and some part of code will be executed on CPU segment which is for general-purpose operations. So maybe, in my CPU there is an optimized segment for operations that my program is doing. But that optimized segment is able to handle only some part of input and rest of input was operated in segments of CPU which are for general-purpose operations.

Another reason could have affected it in a small percentage could be Intel's Turbo Boost Technology. As you may know, thanks to this technology each core of CPU can run on different frequencies. This means, performance of each core may vary during execution.

What I could have done different: If I knew performance would have been a real issue, first of all I would have used C++ which is a lower level programming language instead of high-level programming language C#. Maybe not much, but still it would work a bit faster.

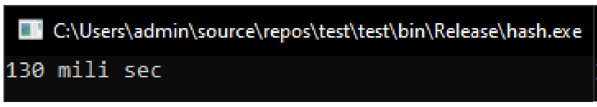
I used a natural language processing library for extracting bigrams. It works flawlessly in terms of extracting bigrams in Turkish language but also probably consumes too much computation power. A simpler tokenizer written with C++ could have run much faster.

I could have tried multi-threaded counting sort. According to the document[6], multi-threaded counting sort does not give an extensive performance gain though. For 100 million elements in the range of 256, only 3x times faster for array holding 8-bit unsigned numbers, and 2x times faster for array holding 16-bit unsigned numbers. Because merging operation is expensive. Considering my input is not an *array (therefore linked list might cause additional performance loss)* and 16-bit unsigned integer is risky for our case *(so need to use 32 bit integer),* still it would worth to give it a try for some performance gain.

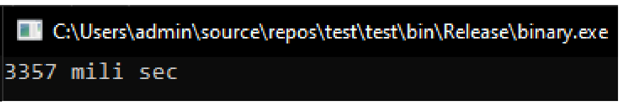
Moreover, I could have tried doing bigram analysis on GPU by using nVidia's Cuda library. [7]String operations are not recommended to be done on GPUs though. But I just cannot think of any better solution that can produce result with better performance.

PS: My program did not process last 4 lines of the input file due to a small mistake I made in my code. 45th line of my code was `if(communicator.rank != communicator.Size - 1)` whereas it should have been `if(rank != communicator.Size - 1).` I realized and fixed it after I started execution.

Last Word: Even though operations made with data structure HashMap (C# Dictionary) didn't increase the execution time by causing a bottleneck, I wanted to try to write my own binary insertion linked list and compare it with HashMap's performance by using a small input. Input was first 20 000 lines of text file you provided. And result was very surprising for me. C#'s HashMap was much faster than my binary insertion linked list 😊


C:\Users\admin\source\repos\test\test\bin\Release\hash.exe

```
130 mili sec
```


C:\Users\admin\source\repos\test\test\bin\Release\binary.exe

```
3357 mili sec
```

20195156022
Tahir Özdilek

References:

[1]https://github.com/mpidotnet/MPI.NET

[2]https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi

[3]https://github.com/altinsoft/nuve

[4]https://www.nimaara.com/counting-lines-of-a-text-file/

[5]https://www.techiedelight.com/counting-sort-algorithm-implementation/

[6]https://web.cs.dal.ca/~arc/teaching/CS4125/2014winter/Assignment3/DrDobb.html

[7]http://notapaper.de/article2.html#:~:text=Introduction,is%20able%20to%20handle%20them.

Makefile: https://blogs.msmvps.com/luisabreu/blog/2007/08/31/compiling-c-code-from-de-code-line/