CAB401   
  
PARALLELIZATION PROJECT REPORT

Dac Duy Anh Nguyen n10603280

Promoter

Table of Contents

[I. Original sequential application 3](#_Toc85369899)

[1) What it does 3](#_Toc85369900)

[2) How it works 3](#_Toc85369901)

[II. Potential parallelism analysis 3](#_Toc85369902)

[III. Tools and techniques utilized 4](#_Toc85369903)

[IV. Outcomes 4](#_Toc85369904)

[V. Difficulties 4](#_Toc85369905)

[VI. Reflection 4](#_Toc85369906)

[VII. References 5](#_Toc85369907)

[VIII. Appendixes 5](#_Toc85369908)

# 1. Original sequential application

## 1.1 What it does

The application is utilized to identify different promoters listed in reference list from the Ecoli bacteria genes.

## 1.2 How it works

Through a brief analysis in run() function, the program first loads Ecoli DNA and reference genes using ParseReferenceGenes function.

After that, in the **first** **For** **loop** (number 1), the program iterates through each Ecoli DNA file to retrieve its gene record using Parse function.

Another **inner For loop** (number 2) iterates through the reference genes to compare it with each Ecoli gene in its **inner For loop** (number 3).

The comparation process is handled by Homologous function to determine if 2 genes serve the same purpose. In this case, they are an Ecoli gene from Ecoli records (from For loop 3) and another gene from reference genes (from For loop 2).

Inside the Homologous function, gene sequences are passed to align function of SmithWatermanGotoh class. This function uses Smith-Waterman algorithm with Gotoh’s improvement for biological local pairwise sequence alignment.

If the Homologous function return true, the program proceeds to extract upstream region of the Ecoli gene by GetUpStreamRegion function and use it to predict if that Ecoli gene is a promoter. If true, the results are stored in ‘consensus’ variable by adding 1 to the counter of that promoter type and the total matches.

# 2. Potential parallelism analysis

From the initial analysis of the given sequential program, Homologous function dominantly consumes 99.7% of the CPU computation. More specifically, it’s the align function in SmithWatermanGotoh class that get called by the Homologous function.

Text

Description automatically generated with medium confidence

Graphical user interface, application

Description automatically generated

Since SmithWatermanGotoh function is a complicated algorithm and required a deep understanding of problem to rewrite for better performance, the work of this report mainly focuses to parallelize the app and increase performance at other parts where possible.

More importantly, this app uses 3-nested for loop which is being executed in a sequential way. As a consequence, the computing time takes a substantial amount of time and could be reduced if applying parallelization.

# Graphical user interface, application Description automatically generated3. Tools and techniques utilized

## 3.1 Hardware specs:

* CPU: Intel Core i7-8565U @1.8GHz
* 4 cores
* 8 virtual cores
* 8GB RA

## 3.2 Software:

- Integrated development environment: IntelliJ

- Java 15

- OS: run on Window

1. My laptop's specs in CPU-Z

- Profiler tool: Jprofiler

## 3.3 Techniques:

In this project, I restructured the For loops code block. I made For loop 2 wrap For loop 1 and 3. Now, the order of the 3 For loops looks like this:

2: referenceGene -> 1:filename -> 3:gene

Text

Description automatically generated

For loops initial order

Text

Description automatically generated

For loops order changed

The aim of this transformation is to increase spatial locality by reducing gap time between accessing ‘record’ in For loop 1 and 3 and therefore, increase the number of cache hits for ‘record’ when transiting from For loop 1 to 3

While this puts referenceGene further from the inner part (*if (Homologous(…){ …}*), the high frequency in calling the variable will keep its memory location stored in cache.

There are 3 parallelization approaches I have implemented in this project:

* Explicit Threading
* Parallel Stream
* Executor Service

### 3.3.1 Explicit Threading

### 3.3.2 Parallel stream

### 3.3.3 Executor Service

# 4. Outcomes

d. timing and profiling results, speedup graphs

e. same results (number of promoters identified)?

# 5. Difficulties

Have to turn *sigma70\_pattern* to a List of ThreadLocal variable due to the use of parallelStream and executorService. <https://www.baeldung.com/java-threadlocal>

Speedup after parallelizing 3 for loops is still not good. Identify causes…

# 6. Reflection

# 7. References

<https://www.baeldung.com/java-executor-service-tutorial>

<https://www.baeldung.com/java-when-to-use-parallel-stream>

# 8. Appendixes

parallelStream 111 50s – 76s

A picture containing chart

Description automatically generated

Diagram, timeline

Description automatically generated

101 57s 78s 79s

Diagram

Description automatically generated

Chart

Description automatically generated with medium confidence

011 72 less overhead?

Diagram

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

001 77

Diagram

Description automatically generated

Chart

Description automatically generated

110 83

Thread waiting for other thread to finish

Timeline

Description automatically generated

A picture containing chart

Description automatically generated

100 108

Too many thread waiting

Timeline

Description automatically generated with medium confidence

Timeline

Description automatically generated

010 101s

Only create 5 threads

Timeline

Description automatically generated A picture containing timeline

Description automatically generated

Sheep1 62 77 78 85 50 60

Sheep2 52

Sheep2 PS 12 threads average 63s

Me 8 threads average 65s

Me 12 threads average 70s

Me 8 threads without specify average 84s