

**DETEKSI GENOMIC REPEATS MENGGUNAKAN ALGORITMA  
KNUTH-MORRIS-PRATT PADA R HIGH-PERFORMANCE  
COMPUTING PACKAGE**

**SKRIPSI**

Diajukan untuk Memenuhi Sebagian dari  
Syarat Memperoleh Gelar Sarjana Komputer  
Program Studi Ilmu Komputer



Oleh  
Ahmad Banyu Rachman  
1303465

**PROGRAM STUDI ILMU KOMPUTER  
DEPARTEMEN PENDIDIKAN ILMU KOMPUTER  
FAKULTAS PENDIDIKAN MATEMATIKA DAN ILMU PENGETAHUAN ALAM  
UNIVERSITAS PENDIDIKAN INDONESIA  
BANDUNG  
2017**

**DETEKSI GENOMIC REPEATS MENGGUNAKAN ALGORITMA  
KNUTH-MORRIS-PRATT PADA R HIGH-PERFORMANCE  
COMPUTING PACKAGE**

Oleh

Ahmad Banyu Rachman

NIM 1303465

Sebuah Skripsi yang Diajukan untuk Memenuhi Salah Satu Syarat Memperoleh  
Gelar Sarjana Komputer di Fakultas Pendidikan Matematika dan Ilmu  
Pengetahuan Alam

© Ahmad Banyu Rachman 2017

Universitas Pendidikan Indonesia

Juli 2017

Hak Cipta Dilindungi Undang-Undang

Skripsi ini tidak boleh diperbanyak seluruhnya atau sebagian, dengan dicetak  
ulang, difoto kopi, atau cara lainnya tanpa izin dari penulis

**AHMAD BANYU RACHMAN**

1303465

**DETEKSI GENOMIC REPEATS MENGGUNAKAN ALGORITMA  
KNUTH-MORRIS-PRATT PADA R HIGH-PERFORMANCE  
COMPUTING PACKAGE**

DISETUJUI DAN DISAHKAN OLEH PEMBIMBING:

Pembimbing I,

**Lala Septem Riza, M.T., Ph.D.**

NIP. 197809262008121001

Pembimbing II,

**Topik Hidayat, Ph.D.**

NIP. 197004101997021001

Mengetahui,

Ketua Departemen Pendidikan Ilmu Komputer

**Prof. Dr. H. Munir, M.IT.**

NIP. 196603252001121001

## **PERNYATAAN**

Dengan ini penulis menyatakan bahwa skripsi dengan judul “Deteksi *Genomic Repeats* Menggunakan Algoritma Knuth-Morris-Pratt pada R *High-Performance Computing Package*” ini beserta seluruh isinya adalah benar-benar karya penulis sendiri. Penulis tidak melakukan penjiplakan atau pengutipan dengan cara-cara yang tidak sesuai dengan etika ilmu yang berlaku dalam masyarakat keilmuan. Atas pernyataan ini, penulis siap menanggung risiko/sanksi apabila di kemudian hari ditemukan adanya pelanggaran etika keilmuan atau ada klaim dari pihak lain terhadap keaslian karya penulis ini.

Bandung, Agustus 2017

Yang Membuat Pernyataan,

Ahmad Banyu Rachman

NIM 1303465

# DETEKSI GENOMIC REPEATS MENGGUNAKAN ALGORITMA KNUTH-MORRIS-PRATT PADA R HIGH-PERFORMANCE COMPUTING PACKAGE

Oleh

Ahmad Banyu Rachman — banyurachman@outlook.com

1303465

## ABSTRAK

Proses pencarian *pattern* pada *string* untuk menemukan perulangan pasang basa pada sebuah sekuens *Deoxyribonucleic Acid* (DNA) membutuhkan waktu yang cukup lama. Beberapa pasang basa yang berulang sebanyak kondisi tertentu akan mengakibatkan penyakit genetik tertentu pula. Oleh karena itu, penelitian ini ingin membuat sebuah model komputasi paralel untuk pencarian *string* pada *pattern* dengan menggunakan algoritma Knuth-Morris-Pratt. Model tersebut akan diimplementasikan pada R *package high-performance computing* yang bernama pbdMPI. Data yang akan digunakan adalah sekuens DNA manusia yang diunduh dari laman resmi Ensembl. Penelitian ini menghasilkan model pemotongan *string* yang akan digunakan untuk pencarian pada setiap iterasi atau setiap *core* yang bekerja. Penelitian ini juga menghasilkan program yang mengimplementasikan model yang telah dirancang sebelumnya. Hasil dari penelitian ini menunjukkan adanya percepatan yang sangat signifikan antara penggunaan *stand alone* dengan penggunaan *parallel computing* yang menggunakan 2 *cores*, 4 *cores* dan 8 *cores*. Penelitian ini juga membuktikan bahwa jumlah *iterator* yang semakin banyak tidak beringinan dengan kecepatan pemrosesan yang lebih baik pula.

Kata Kunci: *DNA, human genom, genomic repeats, string matching, knuth-morris-pratt, high-performance computing*

**GENOMIC REPEATS DETECTION BY USING KNUTH-MORRIS-PRATT  
ALGORITHM ON R HIGH-PERFORMANCE PACKAGE**

*Arranged by*

*Ahmad Banyu Rachman — banyurachman@outlook.com*

*1303465*

**ABSTRACT**

*This research is motivated by pattern searching in a string to find repeated base pairs in the Deoxyribonucleic (DNA) sequences that used many times to do. Several repeated base pair in a condition will be affect a genetic disease. Because of that problem, this research would to make a parallel computing model fot pattern searching in a string by used Knuth-Morris-Pratt algorithm. The model will be implemented in R package high-performance computing that called pbdMPI. The data in this research is the human genome that downloaded from the official site of Ensembl. This research produce a partitioning string model that will be used in the process on each iteration or each working core. This research also produce a program that implementate the model has designed before. The result of this research shows a significant acceleration between stand alone and parallel computing that use 2 cores, 4 cores and 8 cores. This research also prove that the higher value of iterator is not equals with the better processing time.*

*Keywords: DNA, human genom, genomic repeats, string matching, knuth-morris-pratt, high-performance computing*

## KATA PENGANTAR

Puji serta syukur penulis panjatkan kehadirat Allah SWT., atas kehendak dan izin-Nya-lah penulis dapat menyelesaikan skripsi yang berjudul “Deteksi *Genomic Repeats* Menggunakan Algoritma Knuth-Morris-Pratt pada R *High-Performance Computing Package*” ini tepat pada waktunya. Skripsi ini ditulis untuk memenuhi sebagian syarat dalam meraih gelar sarjana komputer di Program Studi Ilmu Komputer, Fakultas Pendidikan Matematika dan Ilmu Pengetahuan Alam, Universitas Pendidikan Indonesia.

Selesainya skripsi ini tak lepas dari bantuan banyak pihak. Untuk itu penulis mengucapkan banyak terima kasih untuk pihak-pihak yang telah membantu penulis selama proses pembuatan skripsi.

Akhirnya penulis sampaikan permohonan maaf atas segala kesalahan dalam skripsi ini. Penulis mengharapkan kritik dan saran yang membangun dari pembaca. Semoga skripsi ini bermanfaat bagi kemajuan pendidikan, ilmu pengetahuan dan teknologi. Aamiin.

Bandung, Agustus 2017

Penulis

## UCAPAN TERIMA KASIH

Alhamdulillahhirabilalamin, puji dan syukur kehadiran Allah SWT. Yang telah melimpahkan rahmat dan hidayah-Nya sehingga penulis diberikan kelancaran dalam menyelesaikan penulisan skripsi ini. Dalam proses menyelesaikan penelitian dan penyusunan skripsi ini, peneliti banyak mendapat bimbingan, dorongan, serta bantuan dari berbagai pihak. Oleh karena itu, pada kesempatan ini peneliti mengucapkan terimakasih serta penghargaan yang setinggi-tingginya, kepada:

1. Kedua orang tua, Papah Aji Sutaji dan Mamah Ai Sutini serta kakak dan adik, Ahmad Purnama Zaelani dan Ahmad Rizky Suwandi yang tanpa henti-hentinya memberikan doa dan dukungan, baik itu dukungan moral, materil maupun spiritual sehingga dapat memotivasi penulis dalam menyelesaikan skripsi ini.
2. Bapak Lala Septem Riza, M.T., Ph.D. selaku pembimbing I atas segala waktu yang dicurahkan untuk membimbing penulis demi terselesaikannya skripsi ini.
3. Bapak Topik Hidayat, Ph.D. selaku pembimbing II yang telah memberikan saran kepada penulis selama proses penyelesaian penelitian dan penulisan skripsi.
4. Bapak Prof. Dr. H. Munir, M.IT., selaku Kepala Departemen Pendidikan Ilmu Komputer FPMIPA Universitas Pendidikan Indonesia.
5. Bapak Eddy Prasetyo Nugroho, M.T., selaku Ketua Program Studi Ilmu Komputer dan Bapak Jajang Kusnendar, M.T., selaku Ketua Program Studi Pendidikan Ilmu Komputer FPMIPA Universitas Pendidikan Indonesia.
6. Bapak Herbert Siregar, M.T., selaku dosen pembimbing akademik yang telah memberikan arahan dan bimbingan selama penulis menjalani perkuliahan.
7. Bapak dan Ibu Dosen Prodi Pendidikan Ilmu Komputer dan Ilmu Komputer yang telah berbagi ilmu yang sangat bermanfaat kepada penulis.
8. Tia Ayu Magfiroh yang selalu memberikan segala dukungan untuk penulis dalam menjalani hidup, perkuliahan serta proses penulisan skripsi ini.
9. Teman seperjuangan organisasi kampus BEM KEMAKOM 2014/2015 dan 2015/2016. Terkhusus untuk Ali, Dheana, Qori, Wiwi dan Irma yang berjuang



bersama di Nondepartemen 2015/2016 serta Kang Agung, Kang Willdan, Teh Rifka, Teh Nida, Ryan, Wandy, Tia, Destanti, Teti, Nissa, dan Muti yang berjuang bersama di Depadsospol 2014/2015.

10. Teman seperjuangan Mahasiswa Departemen Pendidikan Ilmu Komputer FPMIPA UPI 2013. Terkhusus untuk teman-teman dari C1 2013 yang telah berjuang bersama dari awal masa kuliah hingga selesai.
11. Ferldy Verdina Yusup dan Teti Suryati sebagai tim kompetisi Gemastik 8 dan Gemastik 9 serta para asisten Lab Multimedia satu periode: Kang Ghiffari, Kang Kukuh, Kang Handoko, Teh Pudji, Teh Lia, Fidela, Eagan dan Hisyam.
12. Kang Dimas, Indraga, Faisal, Fitri Ana dan Rizki Cahyana atas sumbangsih pemikirannya dalam terselesaikannya skripsi ini.
13. Teman-teman alumni Gemastik 8 UGM Yogyakarta, Club Ink, Studio 229, KKN Cimalaka 2016 yang memberi inspirasi dan kenangan manis bagi penulis.
14. Semua pihak yang tidak bisa disebutkan satu persatu yang telah memberi arti dan dukungan pada penulis.

Semoga semua amal baik yang telah diberikan kepada penulis mendapatkan balasan yang berlipat dari Allah SWT. Aamiin

Bandung, Agustus 2017

Ahmad Banyu Rachman

## DAFTAR ISI

ABSTRAK .....	i
<i>ABSTRACT</i> .....	ii
KATA PENGANTAR .....	iii
UCAPAN TERIMA KASIH.....	iv
DAFTAR ISI.....	vi
DAFTAR TABEL.....	ix
DAFTAR GAMBAR .....	x
DAFTAR LAMPIRAN.....	xii
BAB I PENDAHULUAN.....	1
1.1    Latar Belakang Masalah.....	1
1.2    Rumusan Masalah .....	3
1.3    Tujuan Penelitian.....	3
1.4    Manfaat Penelitian.....	3
1.5    Batasan Masalah.....	4
1.6    Sistematika Penulisan.....	4
BAB II KAJIAN PUSTAKA.....	6
2.1    Penelitian Bidang Biologi .....	6
2.1.1 <i>In Vivo</i> .....	6
2.1.2 <i>In Vitro</i> .....	6
2.1.3 <i>In Silico</i> .....	7
2.2    Genom .....	7
2.2.1.    Genom pada Manusia.....	7
2.2.2.    Genomic Repeats .....	10
2.2.3. <i>Microsatellite (SSR)</i> dan Penyakit .....	10
2.3    Penggunaan Teknik <i>String Matching</i> .....	12
2.4    Algoritma Knuth-Morris-Pratt .....	13
2.5    Bahasa Pemrograman R .....	19
2.5.1    Definisi Bahasa Pemrograman R .....	19
2.5.2    Instalasi Bahasa Pemrograman R.....	20
2.5.3    Contoh Penggunaan Bahasa Pemrograman R.....	22

2.5.4	RStudio.....	25
2.6	<i>High-Performance Computing</i> .....	25
2.7	<i>Package pbdMPI</i> .....	27
2.7.1	<i>Parallel Computing</i> di R.....	27
2.7.2	Definisi pbdMPI.....	28
2.7.3	Instalasi pbdMPI .....	29
2.7.4	Contoh Penggunaan pbdMPI .....	32
BAB III METODOLOGI PENELITIAN.....		33
3.1	Desain Penelitian .....	33
3.2	Alat dan Bahan Penelitian .....	35
3.2.1	Alat Penelitian .....	35
3.2.2	Bahan Penelitian.....	35
3.3	Metode Penelitian.....	36
3.3.1	Metode Pengumpulan Data .....	36
3.3.2	Metode Pengembangan Perangkat Lunak.....	36
BAB IV HASIL DAN PEMBAHASAN .....		38
4.1	Pengumpulan Data .....	38
4.1.1	Mengunduh Data dari Ensembl.....	39
4.1.2	Pengertian Format File .....	41
4.1.3	Penjelasan Isi File .....	41
4.2	Perancangan Model .....	43
4.2.1	Praproses Data.....	44
4.2.2	Pemotongan String Sekuens.....	44
4.2.3	Penggunaan Algoritma Knuth-Morris-Pratt.....	46
4.2.4	Penggabungan Hasil.....	48
4.2.5	Pencarian <i>Pattern</i> Berulang .....	49
4.3	Pengembangan Perangkat Lunak .....	49
4.3.1	Analisa.....	49
4.3.2	Desain.....	50
4.3.3	Implementasi .....	50
4.3.3.1	Masukan Pengguna.....	51
4.3.3.2	Penanganan Kesalahan .....	51

4.3.3.3	Praproses Data .....	52
4.3.3.4	Pembentukan <i>Prefix</i> .....	52
4.3.3.5	Pemotongan <i>String</i> .....	53
4.3.3.6	Eksekusi <i>Parallel Computing</i> .....	54
4.3.3.7	Pencarian dengan Algoritma Knuth-Morris-Pratt .....	55
4.3.3.8	Penggabungan Hasil .....	56
4.3.3.9	Pencarian <i>Pattern</i> Berulang.....	56
4.3.3.10	Pencetakan Hasil.....	57
4.3.4	Pengujian Program .....	57
4.4	Rancangan Skenario Eksperimen .....	59
4.4.1	Skenario <i>Stand Alone</i> .....	60
4.4.2	Skenario <i>Parallel Computing</i> .....	60
4.5	Hasil Eksperimen .....	62
4.5.1	Hasil Eksperimen <i>Stand Alone</i> .....	62
4.5.2	Hasil Eksperimen <i>Parallel Computing</i> .....	63
4.6	Pembahasan .....	64
4.6.1	Perbandingan Kecepatan di <i>Stand Alone</i> .....	64
4.6.2	Perbandingan Kecepatan Berdasarkan Nilai <i>Iterator</i> .....	65
4.6.3	Perbandingan Kecepatan Berdasarkan Jumlah <i>Core</i> .....	69
4.6.4	Perbandingan Kecepatan <i>Stand Alone dan Parallel Computing</i> ....	71
4.6.5	Perbandingan Akurasi .....	74
BAB V KESIMPULAN DAN SARAN.....		75
5.1	Kesimpulan.....	75
5.2	Saran .....	76
DAFTAR PUSTAKA .....		77

## DAFTAR TABEL

Tabel 2. 1 <i>Trinucleotide repeat disorders</i> Polyglutamine .....	11
Tabel 2. 2 <i>Trinucleotide repeat disorders</i> Nonpolyglutamine.....	12
Tabel 2. 3 <i>Package Parallel Computing</i> di R .....	28
Tabel 2. 4 Keterangan pbdMPI .....	29
Tabel 4. 1 File sekuens DNA dari FTP Ensembl .....	38
Tabel 4. 2 Pengujian Program .....	58
Tabel 4. 3 Skenario Eksperimen <i>Stand Alone</i> .....	60
Tabel 4. 4 Skenario Eksperimen <i>Parallel Computing</i> .....	61
Tabel 4. 5 Hasil Eksperimen <i>Stand Alone</i> .....	62
Tabel 4. 6 Hasil Eksperimen <i>Parallel Computing</i> .....	63

## DAFTAR GAMBAR

Gambar 2. 1 Representasi grafis genome manusia .....	8
Gambar 2. 2 Struktur DNA .....	8
Gambar 2. 3 Analisa sekuens DNA pada perulangan CAG .....	9
Gambar 2. 4 Alur Riset Penelitian <i>String Matching</i> .....	13
Gambar 2. 5 Skenario <i>Brute Force</i> pada <i>String Matching</i> .....	16
Gambar 2. 6 Skenario Knuth-Morris-Pratt pada <i>String Matching</i> .....	18
Gambar 2. 7 Pemberian <i>prefix</i> pada pattern 'ATATG' .....	18
Gambar 2. 8 Hasil survei <i>Big Data, Data Mining, dan Data Science</i> .....	20
Gambar 2. 9 Halaman untuk mengunduh R .....	21
Gambar 2. 10 Antarmuka bahasa pemrograman R. ....	21
Gambar 2. 11 Tampilan antar muka Rstudio .....	25
Gambar 2. 12 Kategori dan atribut HPC .....	26
Gambar 2. 13 Memasukan direktori di path environment variables .....	30
Gambar 3. 1 Desain proses penelitian .....	33
Gambar 3. 2 Model <i>Waterfall</i> .....	36
Gambar 4. 1 Spesies yang tersedia pada publikasi nomor 88 FTP Ensembl .....	39
Gambar 4. 2 Folder DNA Manusia pada Publikasi Nomor 88 FTP Ensembl .....	40
Gambar 4. 3 Perancangan model <i>parallel computing</i> algoritma KMP .....	43
Gambar 4. 4 Pemecahan string berdasarkan jumlah iterator .....	45
Gambar 4. 5 Penambahan jumlah karakter perpotongan string .....	46
Gambar 4. 6 Keluaran fungsi KMP string utuh dan potongan string .....	47
Gambar 4. 7 Hasil keluaran dari tiap potongan setelah pemberian adder .....	48
Gambar 4. 8 Perbandingan kecepatan pada skenario <i>stand alone</i> .....	64
Gambar 4. 9 Pencarian <i>pattern</i> 'CCG' dengan 8 <i>cores</i> .....	66
Gambar 4. 10 Pencarian <i>pattern</i> 'CAG' dengan 8 <i>cores</i> .....	67
Gambar 4. 11 Pencarian <i>pattern</i> 'TTAGGG' dengan 8 <i>cores</i> .....	68
Gambar 4. 12 Pencarian <i>pattern</i> 'CCG' dengan nilai <i>iterator</i> 500 .....	70
Gambar 4. 13 Pencarian <i>pattern</i> 'CAG' dengan nilai <i>iterator</i> 500 .....	70
Gambar 4. 14 Pencarian <i>pattern</i> 'TTAGGG' dengan nilai <i>iterator</i> 500 .....	70
Gambar 4. 15 Pencarian 'CCG' di semua eksperimen .....	71

Gambar 4. 16 Pencarian ‘CAG’ di semua eksperimen .....	72
Gambar 4. 17 Pencarian ‘TTAGGG’ di semua eksperimen .....	73

## **DAFTAR LAMPIRAN**

Lampiran 1 Panduan Pengguna (demo.r)

Lampiran 2 Tabel Skenario Eksperimen *Parallel Computing*

Lampiran 3 Tabel Hasil Eksperimen *Parallel Computing*

Lampiran 4 Keluaran Program Eksperimen *Stand Alone*

Lampiran 5 Keluaran Program Eksperimen *Parallel Computing*



# **BAB I**

## **PENDAHULUAN**

### **1.1 Latar Belakang Masalah**

Sekuensing genom dari banyak spesies memungkinkan ilmuwan untuk mempelajari seluruh perangkat gen beserta interaksinya (Campbell dan Reece, 2008). Dalam satu dekade terakhir para ilmuwan harus melakukan penelitian laboratorium selama 3 tahun untuk menganalisa DNA (Pahadia, Srivastava, Srivastava dan Patil, 2015). Salah satu kasus dari analisa DNA yang membutuhkan waktu dan tenaga dalam skala besar tersebut adalah untuk menganalisa penyakit yang disebabkan oleh pola genom yang berulang atau disebut dengan *genomic repeats* (Edgar dan Myers, 2005) seperti tiga pasang basa berulang yang dapat menyebabkan penyakit dalam kategori *trinucleotide repeat disorders* (Orr dan Zoghbi, 2007).

Upaya dari sekuensing telah menghasilkan banyak sekali data sehingga juga melahirkan bidang baru yang disebut dengan bioinformatika. Dengan data sekuens yang dihasilkan, ilmuwan dapat menganalisa kepentingan biologi dengan penerapan metode-metode komputasi yang memungkinkan analisa jauh lebih efisien dari segi waktu dan tenaga seperti yang dilakukan banyak laboratorium riset hari ini (Liu, dkk., 2015).

Dalam menganalisa masalah *genomic repeats* dilakukan analisa *string matching* atau *pattern matching* yang mana akan dicari sebuah pola dalam sebuah teks yang berukuran besar. Algoritma dasar untuk pencarian string atau pola ini adalah dengan cara mencocokkan semua kemungkinan yang terdapat dalam data dari indeks pertama dalam teks hingga habis. Algoritma ini dikenal dengan *Brute Force (Naïve) Algorithm* yang memiliki kompleksitas dengan kemungkinan terburuknya adalah  $O(mn)$  yang akan sangat memakan waktu yang lama jika semakin banyak teks yang akan dijadikan objek pencarian string atau pola (Kindhi dan Sardjono, 2015).

Kebutuhan akan pencarian string atau pola dari data yang semakin besar membuat para ilmuwan membuat algoritma yang lebih efisien dari pada algoritma *brute force* yang mencocokkan satu persatu pola dengan teks yang akan mengakibatkan kompleksitas yang besar. Oleh karena itu beberapa algoritma *string matching* dikembangkan seperti algoritma Knut-Morris-Pratt (Knuth, Morris dan Pratt, 1977). Algoritma pencarian string paling terkenal ini akhirnya memberi inspirasi bagi ilmuwan lainnya untuk terus mengembangkan algoritma yang lebih efisien. Salah satu algoritma pengembangan dari Knuth-Morris-Pratt adalah algoritma Ukkonen (Ukkonen, 1985) yang digagas oleh ilmuwan dari Finlandia dan Fast Hybrid Pattern-Matching Algorithm (Franek, Jennings dan Smyth, 2005).

Namun seiring dengan perkembangan zaman dan semakin banyaknya data yang dihasilkan dalam upaya sekuensing juga menuntut para ilmuwan untuk dapat mengatasi permasalahan komputasi dengan data yang lebih besar (Liu, dkk., 2015). Maka dari itu para ilmuwan komputer membuat sebuah konsep komputasi paralel atau sistem terdistribusi yang memungkinkan sebuah pekerjaan komputasi dapat diselesaikan oleh banyak *core*, *node* atau komputer secara bersamaan. Salah satunya adalah konsep MapReduce (Dean dan Ghemawat, 2010) yang menjadi dasar teknologi pencarian Google dalam skala besar dan memungkinkan para ilmuwan juga menerapkan konsep MapReduce untuk berbagai kasus penelitian.

Contoh lainnya adalah dengan *Package High-Performance Computing* pada bahasa pemrograman R yang beberapa di antaranya adalah Rmpi (Yu, 2002) dan pbdMPI (Chen, Ostrouchov, Schmidt, Patel dan Yu, 2012) yang mengembangkan *parallel computing* dengan *MPI (Message Passing Interface)* pada bahasa pemrograman R. Beberapa contoh selanjutnya adalah randomForestSRC (Ishwaran dan Kogalur, 2007), dclone (dclone: Data Cloning in R, 2010), dll.

Berbagai *package* memiliki prosedur tersendiri dalam penggunaannya seperti harus adanya kompilasi di *prompt/terminal* atau dapat dilakukan di dalam *console* R itu sendiri. Juga dengan konsep pemrogramannya seperti pemecahan data yang akan dianalisa, dsb. Skripsi ini akan mendesain model dan mengimplementasikan algoritma Knuth-Morris-Pratt sebagai algoritma *string matching* terbaik (S.Vijayarani dan R.Janani, 2017) pada R *Package High-Performance Computing*

pbdMPI agar dapat digunakan untuk skala data yang lebih besar di masa yang akan datang.

## 1.2 Rumusan Masalah

Sesuai latar belakang masalah yang telah diuraikan pada sub bab sebelumnya, maka munculah rumusan masalah sebagai berikut:

1. Bagaimana model *parallel computing* dengan menggunakan algoritma Knuth-Morris-Pratt untuk pencarian *pattern* pada *string*?
2. Bagaimana penerapan algoritma Knuth-Morris-Pratt dapat bekerja pada R *Package High-Performance Computing* pbdMPI?
3. Bagaimana hasil keluaran dari program yang telah didesain dan diimplementasikan sebelumnya?
4. Bagaimana perbandingan kecepatan, akurasi dari tiap komputasi yang dikerjakan oleh berbagai jumlah *core* dan *iterator* sesuai model yang dibuat dan diimplementasikan?

## 1.3 Tujuan Penelitian

Setelah diketahui rumusan masalahnya, maka tujuan dari penelitian ini adalah:

1. Merancang model *parallel computing* untuk pencarian *pattern* pada *string* menggunakan algoritma Knuth-Morris-Pratt.
2. Implementasi model pada tujuan pertama pada *Package High-Performance Computing* pbdMPI di bahasa pemrograman R.
3. Melakukan eksperimen dari program yang telah dibuat pada data yang dikumpulkan dari laman FTP Ensembl.
4. Melakukan analisa terkait kecepatan dan akurasi dari hasil eksperimen.

## 1.4 Manfaat Penelitian

Adapun manfaat penelitiannya adalah sebagai berikut:

1. Mempermudah para peneliti di bidang biologi untuk menganalisa *genomic repeats* pada data yang telah tersedia.

2. Membuat sebuah program dan model komputasi paralel untuk dikembangkan oleh peneliti selanjutnya.
3. Memberikan pengetahuan tentang Bioinformatika, khususnya tentang analisa *genomic repeats*.

### 1.5 Batasan Masalah

Adapun batasan masalahnya adalah sebagai berikut:

1. Program ini bekerja untuk data dengan format standar NCBI/Ensembl.
2. Data yang digunakan pada penelitian ini adalah contoh sekuens DNA manusia dari publikasi nomor 88 di laman FTP Ensembl yang dapat diunduh di [ftp://ftp.ensembl.org/pub/release-88/fasta/homo\\_sapiens/dna/](ftp://ftp.ensembl.org/pub/release-88/fasta/homo_sapiens/dna/)
3. Jumlah *core* yang digunakan dalam eksperimen penelitian ini adalah 2, 4, dan 8 *cores*.

### 1.6 Sistematika Penulisan

Pada bagian sistematika penulisan ini akan diuraikan mengenai penjelasan tiap bab.

## BAB I PENDAHULUAN

Bab ini menjelaskan mengapa dan bagaimana penelitian akan dilakukan. Diawali dengan latar belakang masalah, rumusan masalah, tujuan penelitian, manfaat penelitian, batasan masalah, dan sistematika penulisan.

## BAB II TINJAUAN PUSTAKA

Bab ini menjelaskan tentang teori pendamping atau pendukung untuk melakukan penelitian. Teori yang dijelaskan dalam bab ini yaitu mengenai *genomic repeats*, algoritma Knuth-Morris-Pratt dan pbdMPI.

## BAB III METODOLOGI PENELITIAN

Bab ini menjelaskan langkah-langkah penelitian yang akan dilakukan dimulai dari desain penelitian, fokus penelitian, alat dan bahan yang digunakan untuk penelitian dan yang terakhir adalah metode penelitian.

#### BAB IV HASIL DAN PEMBAHASAN

Bab ini menjabarkan hasil penelitian dan eksperimen yang telah dilakukan. Semua pertanyaan mengenai masalah yang diangkat dalam tema skripsi dibahas pada bab ini. Beberapa hal di antaranya adalah tentang proses pengumpulan data, pengembangan model, implementasi sistem, studi kasus, desain eksperimen, dan analisa.

#### BAB V KESIMPULAN DAN SARAN

Bab ini berisi kesimpulan dan saran bagi peneliti selanjutnya dari hasil penelitian yang telah dilakukan.

## **BAB II**

### **KAJIAN PUSTAKA**

#### **2.1 Penelitian Bidang Biologi**

Penelitian atau eksperimen pada bidang biologi dapat dikategorikan menjadi tiga kategori studi: *in vivo*, *in vitro*, dan *in silico*. Ketiga kategori studi tersebut akan dijelaskan pada subsubbab berikutnya.

##### **2.1.1 *In Vivo***

Dalam bahasa latin, *in vivo* berarti “berada pada yang hidup” (Iverson dan Cheryl, 2007) di mana penelitian dilakukan langsung dalam organisme yang masih hidup atau sudah mati. Penelitian kategori ini dinilai berisiko terutama untuk penelitian yang melibatkan manusia yang masih hidup. Mikrobiologis asal Inggris, Prof. Harry Smith dan rekannya menunjukkan betapa pentingnya penelitian *in vivo* pada saat meneliti tentang efek dari hewan yang terinfeksi *Bacillus anthracis*.

##### **2.1.2 *In Vitro***

Berbeda dengan *in vivo*, dalam bahasa latin *in vitro* berarti “berada dalam gelas” (Iverson dan Cheryl, 2007) yang bermakna teknik atau prosedur penelitian dilakukan pada lingkungan yang terkontrol di luar organisme yang hidup. Penelitian ini juga kadang disebut dengan “*test-tube experiments*” karena biasanya dikerjakan pada tabung atau alat-alat sejenis.

Salah satu keunggulan *in vitro* dibandingkan dengan *in vivo* adalah kesederhanaan penelitian. Pada *in vivo* yang menggunakan organisme hidup memiliki sistem fungsional yang sangat kompleks yang setidaknya terdiri dari puluhan ribu gen, molekul protein, molekul RNA, ion organik, dll (Albert, 2008). Kompleksitas tersebut membuat sulit untuk mengidentifikasi interaksi antara komponen individu dan mengeksplorasi fungsi dasar biologisnya. Dengan *in vitro* peneliti bisa bekerja dan fokus pada komponen yang lebih kecil (Nairn dan Price, 2009).

Namun *in vitro* juga memiliki kekurangan yang salah satunya adalah akan adanya kecenderungan intrepetasi hasil dari setiap peneliti yang dinilai terlalu menyimpang pada keadaan sebenarnya (Rothman, 2002).

### 2.1.3 *In Silico*

Kategori ini bermakna penelitian yang menggunakan bantuan komputer untuk simulasi. Kata '*silico*' sendiri berasal dari kata '*silicon*' yang merujuk pada bahan semikonduktor pada chip komputer. Penelitian jenis ini memiliki keunggulan untuk melakukan penelitian yang lebih cepat dan mengurangi kebutuhan pekerjaan laboratorium yang mahal. Seperti pada tahun 2007 di mana diadakan penelitian untuk *tuberculosis* yang hanya dilakukan dengan hitungan menit dari yang sebelumnya membutuhkan waktu satu bulan pengerjaan di laboratorium (University of Surrey, 2007).

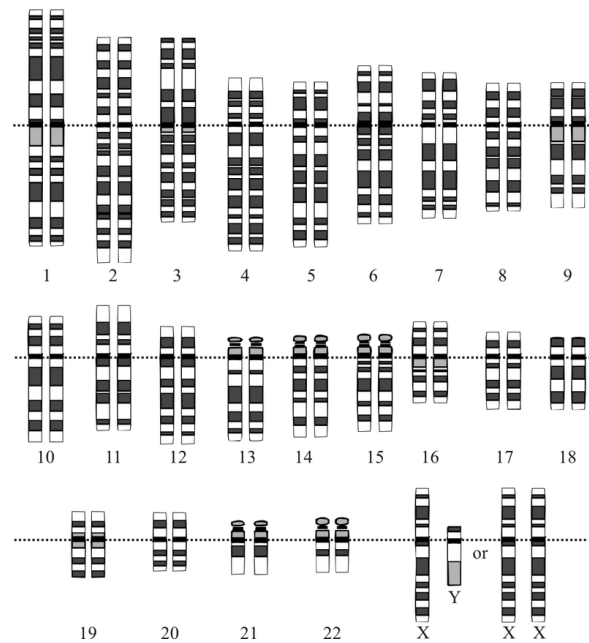
## 2.2 Genom

Genom adalah sekumpulan informasi genetik dalam sel yang pertama kali dikemukakan oleh Hans Winkler, seorang professor botani dari University of Hamburg, Jerman (Winkler, 1920). Ukuran genom ditentukan oleh jumlah pasang basa dari tiap makhluk hidup. Sampai saat ini, ukuran genom terbesar yang diketahui adalah tipe organisme amuba *Polychaos dubium* dengan jumlah pasang basa sebesar 670 miliar (Parfrey, Lahr dan Katz, 2008).

### 2.2.1. Genom pada Manusia

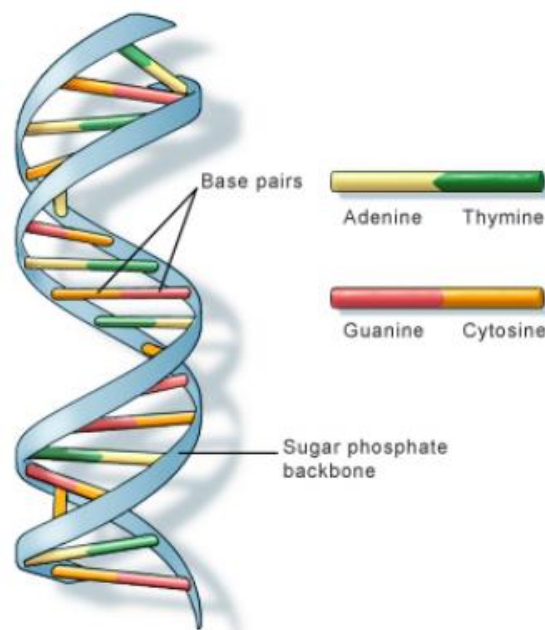
Genom manusia memiliki ukuran kurang lebih 3 miliar pasang basa dari total 23 kromosom yang ada penelitiannya dimulai dengan *Human Genome Project* (HGP) yang dimulai pada tahun 1990 (Venter, dkk., 2001). Sedangkan sejarah untuk sekuensing DNA dimulai oleh Sanger sejak tahun 1997 (Sanger, 1977).

Pada gambar 2.1 diperlihatkan representasi grafis dari genom manusia yang terbagi ke dalam 23 kromosom, di mana kromosom XX untuk ilustrasi jenis kelamin perempuan dan kromosom XY untuk ilustrasi jenis kelamin laki-laki. Ilustrasi tersebut adalah DNA yang berada pada inti sel (DNA pada mitokondria tidak ditampilkan).



**Gambar 2. 1 Representasi grafis genome manusia (U.S. National Library of Medicine, n.d.)**

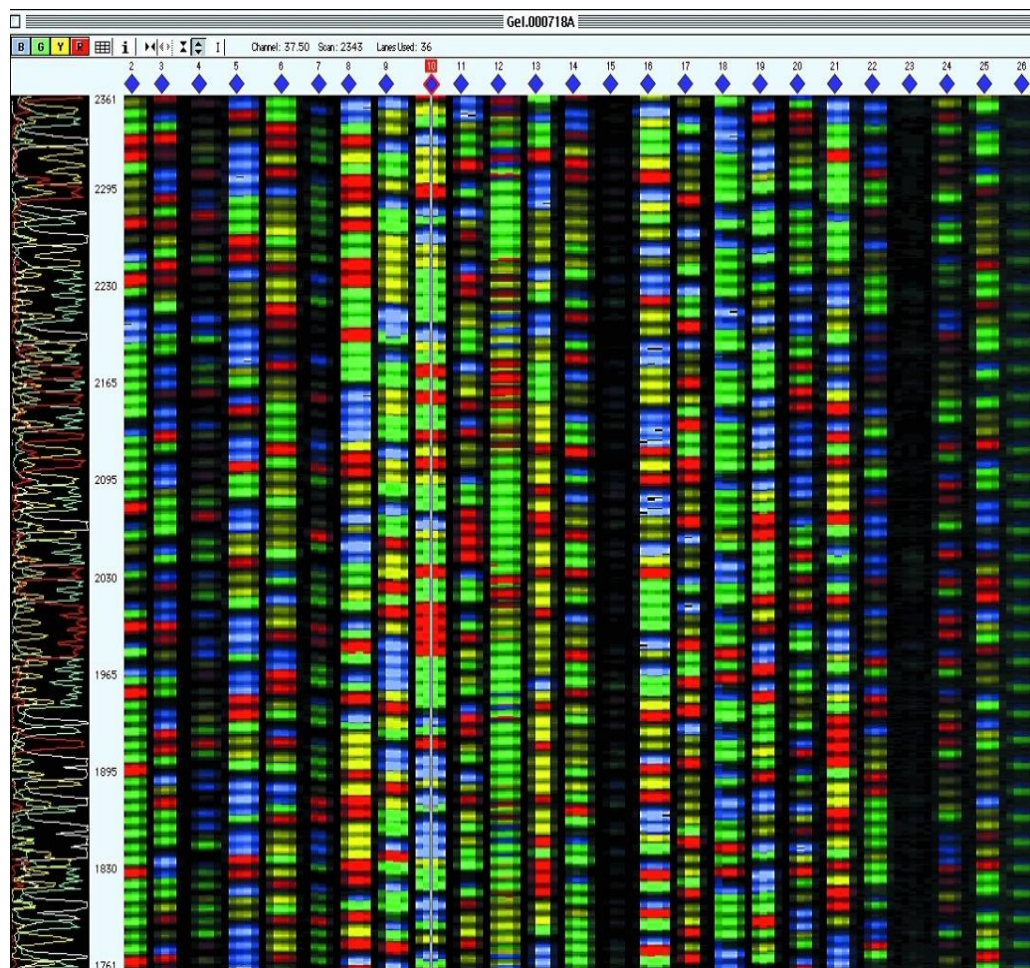
DNA pada manusia sendiri sama seperti hewan yang mana terletak pada inti sel dan mitokondria, berbeda dengan tumbuhan yang juga memiliki DNA yang terletak pada kloroplas. DNA atau asam deoksiribonukleat adalah molekul asam nukleat beruntai ganda dan berbentuk heliks yang tersusun atas monomer-monomer nukleotida dengan gula deoksiribosa (Campbell dan Reece, 2008).



**Gambar 2. 2 Struktur DNA (U.S. National Library of Medicine, n.d.)**



Sebagian besar DNA terletak di sel nukleus namun juga dapat dijumpai pada mitokondria. Informasi DNA disimpan sebagai kode dan menjadi empat basa kimia: *adenine* (A), *guanine* (G), *cytosine* (C), dan *thymine* (T) diilustrasikan pada Gambar 2.2. Basa DNA memiliki pasangan masing-masing, A dengan T sedangkan C dengan G. Setiap basa juga terlampir pada molekul gula dan molekul fosfat. Secara bersamaan, mereka disebut dengan *nucleotide* (U.S. National Library of Medicine, n.d.).



**Gambar 2. 3 Proses sekuensing DNA (Foundation, n.d.)**

Gambar 2.3 memperlihatkan proses sekuensing DNA. Setiap satu warna merepresentasikan pasang basa (A, G, C, T). Setiap baris diartikan pada kromatogram pada sebelah kiri. Dengan ini peneliti dapat menentukan sekuens DNA.

### 2.2.2. Genomic Repeats

*Genomic Repeats* atau *repeated sequences* adalah pola dari asam nukleat yang berulang pada genom. Berdasarkan urutannya, *genomic repeats* terbagi menjadi *minisatellite* dan *microsatellite*.

#### a. *Minisatellite*

*Minisatellite* adalah perulangan pola asam nukleat dengan panjang 10-60 pasang basa yang berulang sekitar 5-50 kali dalam satu sekuens (U.S. National Library of Medicine, n.d.). Pada manusia, *minisatellite* pertama ditemukan pada tahun 1980 (Wyman dan White, 1980).

#### b. *Microsatellite*

*Microsatellite* adalah perulangan pola asam nukleat dengan panjang 2-5 pasang basa yang berulang sekitar 5-50 kali dalam satu sekuens (Turnpenny P, 2005). Tipe ini juga sering disebut dengan *simple sequence repeats (SSR)* oleh para ilmuwan genetika pada tumbuhan. Pola seperti TATATATATATA disebut dengan *dinucleotide microsatellite*, sedangkan pola seperti GTCGTCGTCGTC disebut dengan *trinucleotide microsatellite*.

### 2.2.3. *Microsatellite (SSR) dan Penyakit*

Perulangan pola-pola pendek (*microsatellite*) yang berlebihan atau melewati batas normal dapat menyebabkan penyakit genetik. Studi (Calladine, Drew, Luisi dan Travers, 2004) menjelaskan bahwa tiga pasang basa atau *triplet/trinucleotide* yang paling berpengaruh dalam penyakit manusia adalah CGG, CCG, CTG, CAG, GAA dan TTC selain dari 58 *trinucleotide* lainnya. Penyakit yang disebabkan oleh tiga basa yang berulang ini disebut juga dengan *trinucleotide repeat disorders*.

Pada awalnya, *trinucleotide repeat disorders* terbagi menjadi dua grup berdasarkan pasang basa yang berulang: perulangan CGG atau CCG sebagai alternatif dan perulangan CAG atau CTG sebagai alternatif (Claude T. Ashley dan Warre, 1995). Kini kedua grup *trinucleotide repeat disorders* terbagi menjadi *Polyglutamine (PolyQ) Diseases* dan *Nonpolyglutamine Diseases*. Grup penyakit

yang disebabkan oleh basa CAG (*polyglutamine*) sebagian besar disebabkan racun fungsi perluasan mutan protein (Lutz, 2007).

**Tabel 2. 1 *Trinucleotide repeat disorders* yang termasuk grup Polyglutamine (Lutz, 2007)**

Disease	CAG Repeat Size	
	Normal	Disease
Spinobulbar Muscular Atrophy (Kennedy Disease)	9-36	38-62
Huntington's Disease	6-35	36-121
Dentatorubral-pallidoluysian Atrophy (Haw-River Syndrome)	6-35	49-88
Spinocerebellar Ataxia Type 1	6-44	39-82
Spinocerebellar Ataxia Type 2	15-31	36-63
Spinocerebellar Ataxia Type 3 (Machado-Joseph Disease)	12-40	55-84
Spinocerebellar Ataxia Type 6	4-18	21-33
Spinocerebellar Ataxia Type 7	4-35	37-306

Tabel 2.1 menunjukkan beberapa contoh penyakit *trinucleotide repeat disorders* yang termasuk grup *Polyglutamine*. Nama penyakit dapat terlihat pada kolom "*Disease*" diikuti dengan jumlah perulangan normal dan jumlah perulangan yang terdeteksi penyakit bagi setiap deskripsinya. Contoh pada *Huntington's Disease* memiliki keterangan perulangan terdeteksi penyakit pada 36-121 yang berarti terdapat perulangan CAGCAGCAGCAGCAG... sebanyak minimal 36 kali.

*Huntington's Disease* yang selanjutnya dapat disingkat HD biasanya adalah penyakit yang dibawa oleh keturunan yang dapat menyerang usia anak-anak sekalipun. Penyakit ini diakibatkan pengulangan CAG yang terletak pada *N-terminus* dengan jumlah pengulangan 36-121 kali di mana seharusnya memiliki pengulangan normal sebanyak 6-34 kali (Orr dan Zoghbi, 2007). Sedangkan penyakit *trinucleotide repeat disorders* yang termasuk grup *Polyglutamine* dapat dilihat pada Tabel 2.2.

Tabel 2.2 memperlihatkan beberapa penyakit beserta tipe pengulangan yang menyebabkan penyakit genetik tersebut muncul. Contohnya pada *Fragile XE Syndrome* atau *FRAXE* yang memiliki pengulangan 200-900 pola CCG di mana seharusnya hanya 4-39 kali pengulangan untuk ukuran normal.

**Tabel 2. 2 Trinucleotide repeat disorders katogori nonpolyglutamine (Lutz, 2007)**

Disease	Type of Repeat	Normal Repeat	Abnormal Repeat
Fragile X Syndrome	CGG	6-53	>230
Fragile X Mental Retardation	GCC	6-53	>200
Fragile XE Syndrome	CCG	4-39	200-900
Friedreich Ataxia	GAA	7-34	>100
Myotonic Dystrophy	CTG	5-37	>50
Spinocerebellar Ataxia type 8	CTG	16-37	110-250
Spinocerebellar Ataxia type 10	ATTCC	10-22	100-1000
Congenital Failure of Autonomic Control (Ondine's Curse)	GCG	20	25-33
Oculopharyngeal Muscular Dystrophy	GCG	6	9-13

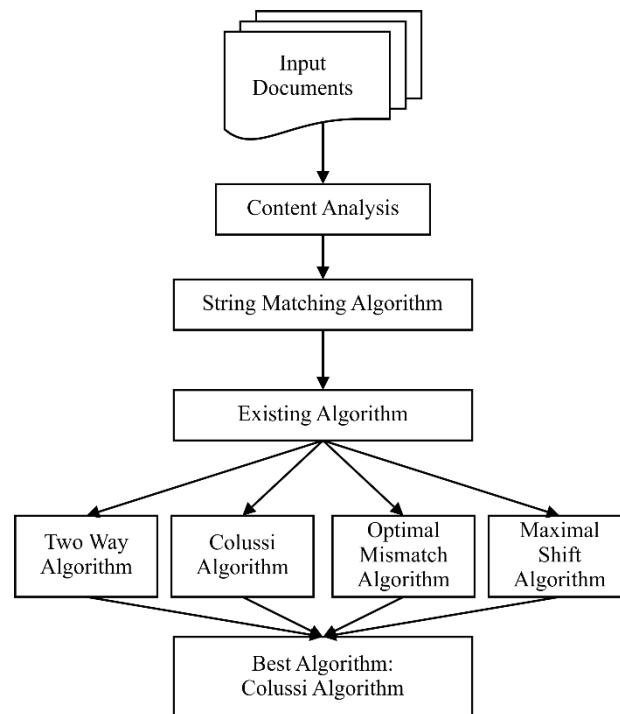
### 2.3 Penggunaan Teknik *String Matching*

*String matching* atau juga biasa disebut juga *pattern matching* atau *pattern searching* adalah teknik yang termasuk dalam *information retrieval*, yaitu metode untuk mengambil informasi dari sesuatu yang ingin diketahui. Penggunaan teknik *string matching* saat ini digunakan untuk berbagai hal terutama dalam keamanan informasi, bioinformatika, deteksi plagiarisme, pemrosesan teks dan pencocokan dokumen (Vijayarani dan Janani, 2017).

Salah satu yang telah disebutkan di atas adalah bidang keamanan informasi atau keamanan jaringan. Dengan perkembangan teknologi informasi yang cepat, keamanan jaringan menjadi isu yang hangat. Protokol HTTP adalah protokol komunikasi terbesar yang digunakan di internet di seluruh dunia. Penggunaan *string matching* membantu dalam perbaikan data paket HTTP secara *real-time* yang tentunya mengakibatkan kebutuhan akan algoritma *string matching* yang efisien menjadi sangat penting (Zhang, dkk., 2015).

Bidang lain yang tak kalah populer untuk penggunaan teknik *string matching* adalah *bioinformatics*. Menganalisa sekuen DNA dengan menggunakan teknik *string matching* dapat memberikan informasi yang dibutuhkan secara cepat. Ini yang menyebabkan *string matching* menjadi topik riset yang menarik dan penting dalam bidang ilmu komputer (Yangjun Chen, 2016).

Riset dari Vijayarani juga membagi algoritma *string matching* ke dalam empat kelompok, yaitu *Two way algorithm*, *Colussi algorithm*, *Optimal mismatch algorithm* dan *Maximal shift algorithm*. Hasil riset tersebut menjelaskan bahwa algoritma terbaik dari keempat kelompok tersebut adalah *Colussi algorithm* yang memiliki relevansi tertinggi, yaitu 100% untuk *single word*, 99% untuk *multiple words*, dan 92% untuk *file*.



**Gambar 2. 4 Alur Riset Penelitian *String Matching* (S.Vijayarani dan R.Janani, 2017)**

Gambar 2.4 memperlihatkan alur riset untuk menentukan algoritma pencarian *string* terbaik dengan hasil algoritma terbaik yaitu *Colussi algorithm* dibangun berdasarkan analisa algoritma Knuth-Morris-Pratt yang memperhitungkan posisi dan *prefix* dari *pattern* yang akan dicari. Pada subbab selanjutnya akan dibahas mengenai algoritma Knuth-Morris-Pratt yang menjadi algoritma yang akan diimplementasikan pada penelitian ini.

## 2.4 Algoritma Knuth-Morris-Pratt

Algoritma Knuth-Morris-Pratt ditemukan oleh ketiga ilmuwan bernama Knuth, Morris dan Pratt untuk menemukan posisi *string* yang diberikan untuk *text*-

*editing programs*. Algoritma ini memberikan informasi *prefix* pada *string* atau *pattern* yang akan dicari sebelum melakukan pencarian.

Algoritma 1 Modifikasi Knuth-Morris-Pratt	
1	Input: nama file, <i>pattern</i> Output: vektor dari indeks yang ditemukan
2	<b>begin</b>
3	file ← Masukkan nama file
4	<b>if</b> nama file ada <b>then</b>
5	<i>string</i> ← PreProcessing(file) //tahap praproses
6	<b>endif</b>
7	prefix ← KMP_Prefix( <i>pattern</i> ) //pembentukan prefix
8	hasil ← KMP( <i>string</i> , <i>pattern</i> , prefix) //pencarian <i>pattern</i> pada <i>string</i>
9	<b>end</b>

Pada Algoritma 1 digambarkan *pseudocode* dari modifikasi algoritma Knuth-Morris-Pratt. Dalam algoritma tersebut terdapat tiga fungsi yaitu PreProcessing, KMP\_Prefix dan KMP.

Algoritma 2 PreProcessing	
1	Input: nama file Output: string
2	<b>begin</b>
3	file ← membaca isi file dari nama yang diberikan
4	awal ← mencari indeks kata 'ORIGIN' dalam file
5	akhir ← mencari indeks '\n//' dalam file
6	<i>string</i> ← mengambil bagian file antara indeks awal dan akhir
7	<i>string</i> ← melakukan penghapusan karakter yang tidak digunakan dengan regex "  \r \n ORIGIN [0-9]"
8	<b>return</b> <i>string</i>
9	<b>end</b>

Algoritma 2 menggambarkan praproses yang mana hasilnya akan digunakan sebagai *string* pada algoritma KMP. Namun sebelum itu terdapat fungsi untuk mencari *prefix* yang akan dijelaskan pada Algoritma 3.

Algoritma 2 KMP_Prefix	
1	Input: <i>pattern</i> Output: vektor nilai prefix
2	<b>begin</b>
3	length ← mengambil panjang <i>pattern</i>
4	prefix[1] ← 0 //inisiasi variabel vektor prefix
5	a ← 0 //inisiasi variabel a
6	<b>for</b> b=2 <b>to</b> length <b>do</b>

7	<b>while</b> a>0 <b>and</b> P[a+1]≠P[b] <b>do</b>
8	a ← Prefix[a]
9	<b>endwhile</b>
10	<b>if</b> P[a+1]=P[b] <b>then</b>
11	a ← a+1
12	<b>end if</b>
13	Prefix[b] ← a
14	<b>endfor</b>
15	<b>return</b> Prefix[]
16	<b>end</b>

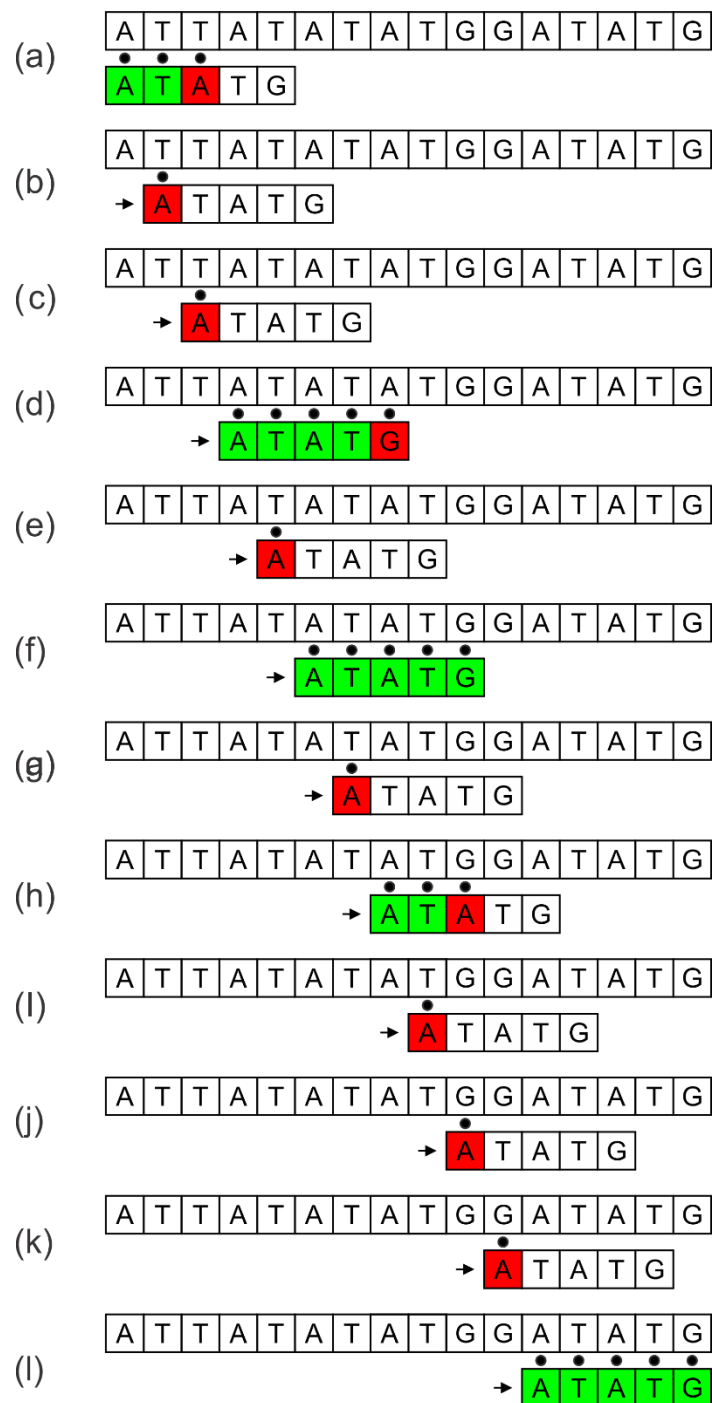
Setelah mendapatkan nilai *prefix*, langkah terakhir adalah mengeksekusi fungsi KMP yang dipaparkan pada Algoritma 3.

Algoritma 3 KMP	
1	Input: string, pattern, prefix Output: vektor indeks hasil
2	<b>Begin</b>
3	n_string ← masukkan panjang string
4	n_pattern ← masukkan panjang pattern
5	index ← 0 //inisiasi variabel index berupa vektor
6	I ← 0 //inisiasi variabel i
7	<b>for</b> j=1 <b>to</b> n_string <b>do</b>
8	<b>while</b> i > 0 <b>and</b> pattern[i+1] != string[j] <b>do</b>
9	i <- prefix[i]
10	<b>endwhile</b>
11	<b>if</b> pattern[i+1] == string[j] <b>then</b>
12	i <- i+1
13	<b>endif</b>
14	<b>if</b> i == n_pattern <b>then</b>
15	index <- c(index, j-n_pattern+1)
16	total <- total+1
	i <- prefix[i]
	<b>endif</b>
	<b>endfor</b>
	<b>end</b>

Algoritma Knuth-Morris-Pratt ini sangat berbeda dengan *brute force* dalam segi pencocokan *pattern* pada *string* yang diberikan. Algoritma *brute force* atau terkadang dikenal dengan *naïve algorithm* untuk *string matching* memiliki kompleksitas  $O(mn)$  karena mencocokkan semua kemungkinan dari setiap karakter pada teks (Barth, 1981).

Untuk contoh kita memiliki *pattern* atau *string* yang akan dicari adalah 'ATATG' dalam teks 'ATTATATATGGATATG'. Algoritma *brute force* mencocokkan setiap 'ATATG' pada lima karakter pada teks dan akan bergeser satu

indeks untuk mencocokkan lima karakter tersebut. Algoritma ini dapat dimodifikasi agar dapat menggeser indeks jika karakter yang sedang dibandingkan tidak cocok atau *mismatch* walaupun berada di karakter pertama dari 'ATATG'. Walaupun demikian, pergeseran indeks tetaplah per satu langkah yang masih memiliki kompleksitas yang besar.



**Gambar 2. 5 Skenario Brute Force pada String Matching**



Gambar 2.5 adalah skenario *string matching* menggunakan algoritma *brute force* pada kasus di atas. Pada Gambar 2.5(a) terlihat pencocokan 'A' dan 'T' yang cocok namun ditemukan *mismatch* saat mencocokkan 'A' dengan 'T' pada teks. Karena sudah tidak cocok di indeks ke-tiga *pattern*, maka pencocokan berikutnya adalah dengan menggeser satu indeks *pattern* terhadap teks. Gambar 2.5(b) dan 2.5(c) memperlihatkan *mismatch* pada indeks pertama *pattern*. Setelahnya dilakukan penggeseran indeks seperti sebelumnya. Akhirnya *pattern* dapat cocok ketika berada pada Gambar 2.5(f) dan 2.5(l).

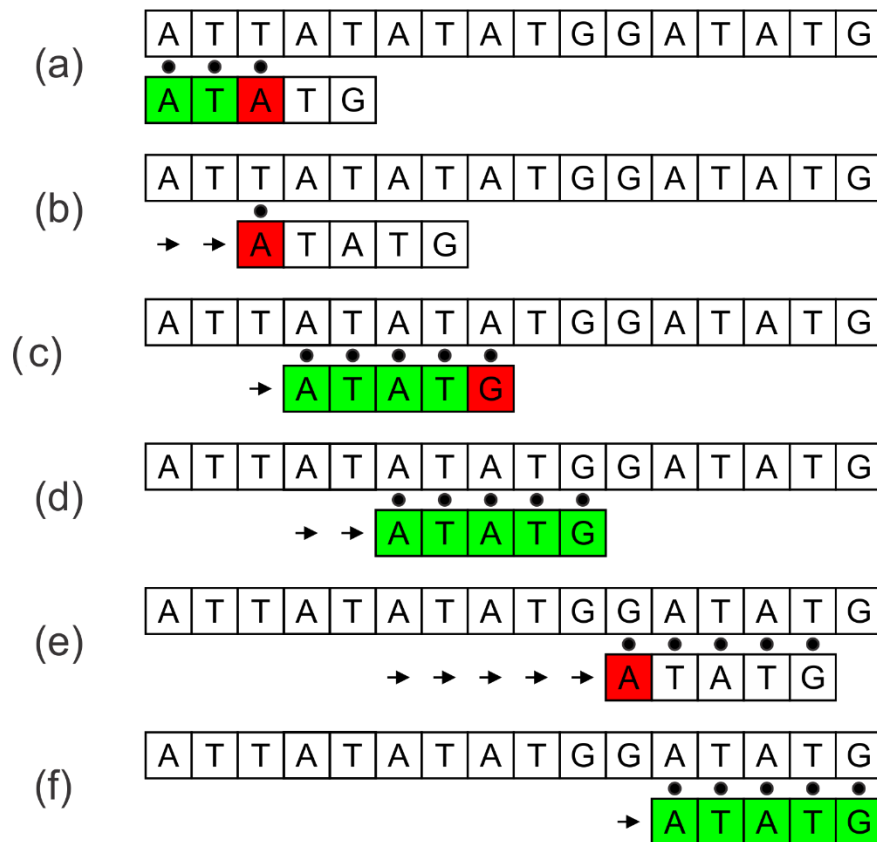
Dari skenario tersebut terlihat bahwa algoritma *brute force* membutuhkan 28 kali pencocokkan untuk menemukan dua *pattern* yang terdapat dalam teks yang disediakan.

Berbeda dengan algoritma *brute force*, algoritma Knuth-Morris-Pratt menyimpan informasi pada pencocokan sebelumnya untuk menghindari pencocokan yang sia-sia. Praproses pada *pattern* dengan memberikan *prefix* dari tiap karakter *pattern* akan menjadi petunjuk untuk melakukan pengabaian pencocokkan yang dinilai tak perlu karena sudah pasti tak cocok atau sudah pasti cocok.

Gambar 2.6 adalah skenario *string matching* dengan algoritma Knuth-Morris-Pratt pada permasalahan yang sama. Terlihat pada Gambar 2.6(a) masih sama seperti 2.5(a) yang membandingkan *pattern* dengan teks. Namun saat di langkah ke-dua, Gambar 2.6(b) melakukan pergeseran dua indeks karena informasi yang tersedia pada *pattern*. Salah satu langkah yang sangat baik dari algoritma ini dapat terlihat pada Gambar 2.6(c) menuju 2.6(d) di mana pergeseran indeks sebesar dua langkah juga ditambah dengan pencocokkan *pattern* pada indeks ke-tiga. Hal ini terjadi karena dengan algoritma yang akan dijelaskan pada subsubbab berikutnya, indeks pertama dan ke-dua *pattern* pada Gambar 2.6(d) sudah dipastikan cocok dengan teks yang se-indeks. Maka pencocokkan dimulai dari indeks ke-tiga *pattern*.

Berikutnya dapat dilihat pada Gambar 2.6(e) melakukan pergeseran indeks sebanyak lima langkah karena dengan algoritma Knuth-Morris-Pratt memastikan di

lima langkah tersebut *pattern* tidak akan menemukan kecocokkan sempurna dari tiap karakter di *pattern* dengan teks yang se-indeks pada tiap langkahnya.



**Gambar 2. 6 Skenario Knuth-Morris-Pratt pada *String Matching***

*Prefix* itu sendiri adalah derajat pengulangan karakter yang akan dapat menentukan karakter yang tidak perlu dicocokkan kembali seperti pada Gambar 2.6(d) dan juga untuk melakukan pergeseran indeks untuk mengabaikan pencocokkan yang sia-sia.

Index	1	2	3	4	5
Pattern	A	T	A	T	G
Prefix	0	0	1	2	0

**Gambar 2. 7 Pemberian *prefix* pada pattern 'ATATG'**

Berikut adalah algoritma untuk penentuan *prefix* pada algoritma Knuth-Morris-Pratt seperti yang ditampilkan pada Gambar 2.7.

Kompleksitas algoritma KMP-Prefix adalah  $O(m)$  di mana variabel ‘ $m$ ’ adalah panjang sekuen dari *pattern* yang akan dicari. Sedangkan kompleksitas untuk KMP-Search adalah  $O(n)$  di mana variabel ‘ $n$ ’ adalah panjang sekuen dari teks yang menjadi objek pencarian. Maka dari itu keseluruhan kompleksitas dari algoritma Knuth-Morris-Pratt adalah  $O(m+n)$ .

## 2.5 Bahasa Pemrograman R

Pada subbab ini akan dijelaskan mengenai bahasa pemrograman R dari definisi, instalasi dan contoh penggunaannya serta mengenai RStudio.

### 2.5.1 Definisi Bahasa Pemrograman R

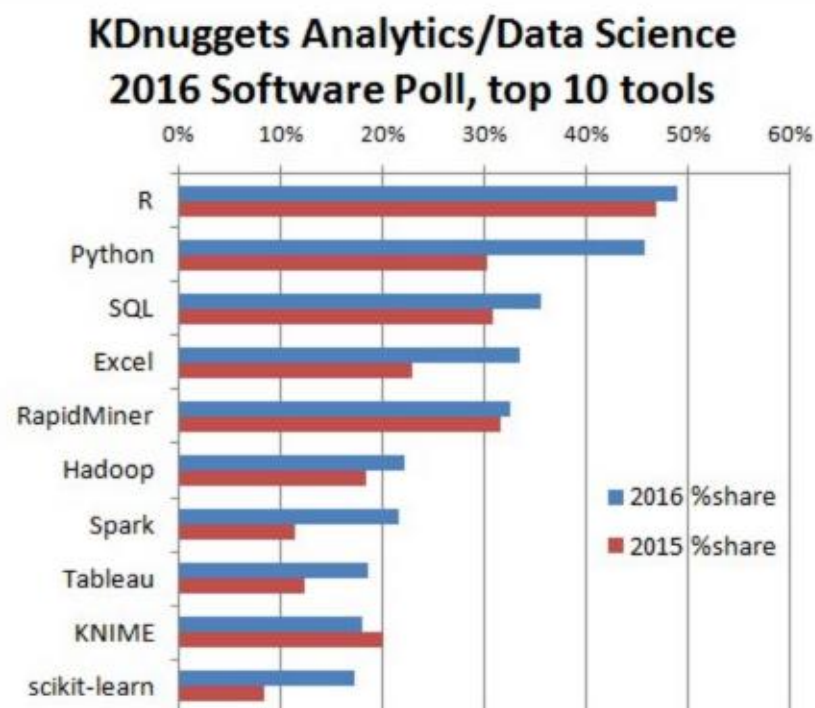
R adalah bahasa pemrograman yang digunakan untuk analisis statistika dan grafik. R dibuat oleh Ross Ihaka dan Robert Gentleman di Universitas Auckland, Selandia Baru. Saat ini bahasa R dikembangkan oleh R Development Core Team. Bahasa R menjadi standar *de facto* diantara statistikawan untuk pengembangan perangkat lunak statistika dan dipergunakan secara luas untuk pengembangan perangkat lunak statistika. Bahasa R merupakan bersi *open-source* dari bahasa pemrograman S. Bahasa R memiliki kemampuan yang tidak kalah dengan paket program pengolahan data komersial.

R *Project* pertama kali dikembangkan oleh Robert Gentleman dan Ross Ihaka yang bekerja di departemen statistic Universitas Auckland tahun 1995. Sejak saat itu program R mendapat sambutan yang luar biasa dari kalangan statistikawan, *industrial engineering*, peneliti, *programmer*, dan lain sebagainya. Saat ini, *source code* kernel R dikembangkan oleh R Core Team yang beranggotaan 17 orang statistisi dari berbagai penjuru dunia.

R pertama kali diluncurkan pada tahun 1997, dan terus dikembangkan sehingga fitur dari waktu ke waktu semakin bertambah dan *bug* dari program pun semakin diminimalisir. Bahasa R memiliki banyak kelebihan, diantaranya tersedia untuk berbagai sistem operasi seperti Linux, Mac OS dan Windows, memiliki kemampuan membuat graph, dll.

Rexer’s Annual Data Miner Survey 2010 menyebutkan bahwa R telah menjadi alat data mining yang digunakan oleh mayoritas pengguna. Salah satu

penyebabnya adalah adanya Rattle, yaitu suatu *library* yang digunakan secara khusus untuk Data Mining melalui GUI Rattle dapat menyajikan ringkasan data secara statistik dan secara visual dari berbagai sumber data seperti Excel, SQL, ataupun XML, yang selanjutnya dapat mentransformasi data ke dalam bentuk yang siap untuk dimodelkan. Untuk pemodelannya dapat digunakan berbagai metode baik *supervised* maupun *unsupervised* dan sekaligus mampu membuat laporan secara grafis untuk menampilkan kerja model yang dibangun.



**Gambar 2. 8 Hasil survei dari penggunaan bahasa program untuk *Big Data*, *Data Mining*, dan *Data Science* oleh KDnuggets**

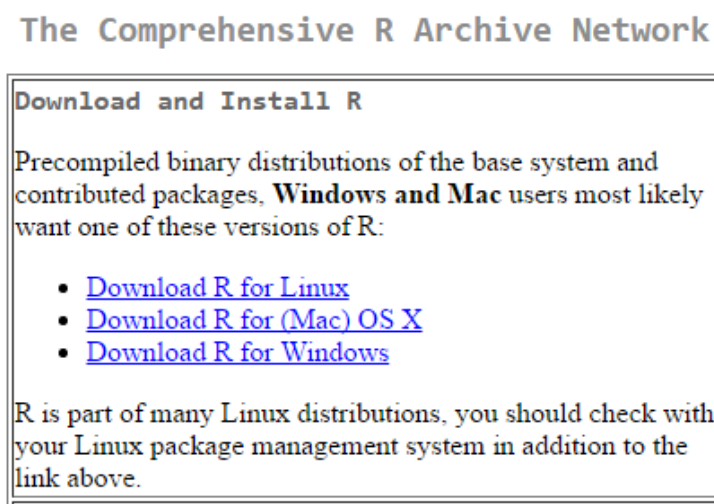
Sebuah survei yang dilakukan oleh KDnuggets pada tahun 2015 tentang prediksi analisis / Data Mining / Data science software, menunjukkan bahwa R merupakan pemrograman yang paling banyak digunakan baik oleh ilmuwan ataupun programmer untuk mengatasi masalah-masalah yang berkaitan dengan ilmu pengetahuan, yang ditunjukkan pada Gambar 2.8.

### 2.5.2 Instalasi Bahasa Pemrograman R

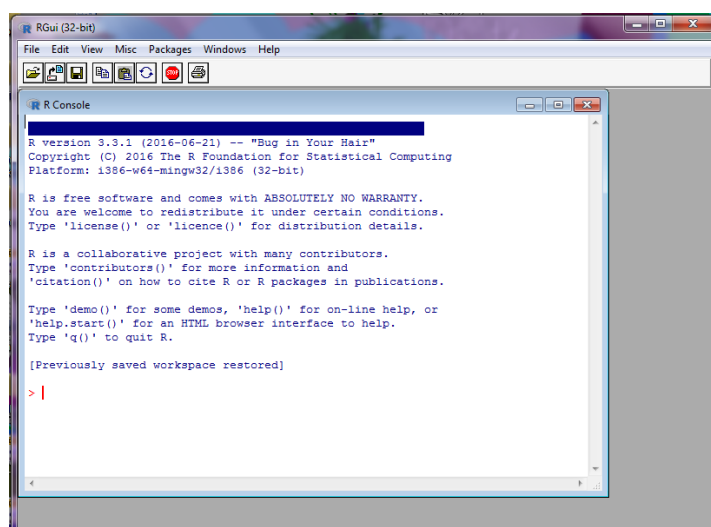
Setiap bahasa pemrograman memiliki sebuah *compiler* untuk menjalankan kode programnya, begitu juga bahasa R untuk menggunakan bahasa R pertama-tama harus meng-*install compiler* bahasa R yang dapat didapatkan secara gratis di

situs resminya yaitu <http://www.r-project.org/>. Ketika mengunduh *installer* bahasa R, biasanya sudah termasuk dengan *Graphical User Interface* (GUI), jadi kita tidak harus meng-*compile* kode program seperti bahasa pemrograman lainnya, dengan kelebihan tersebut kita bisa menjalankan perintah per baris mirip seperti produk perangkat lunak dari MathWorks yaitu Matrix Laboratory atau MATLAB.

Langkah selanjutnya yaitu meng-*install* program bahasa R sesuai dengan sistem operasi anda. Jika kita mengunjungi halaman unduhan bahasa R (<https://cloud.r-project.org/>), kita akan dipandu untuk mengunduh dan meng-*install* sesuai dengan sistem operasi yang kita gunakan. Halaman tersebut terlihat seperti pada Gambar 2.9.



**Gambar 2. 9 Halaman untuk mengunduh R**



**Gambar 2. 10 Antarmuka bahasa pemrograman R.**

Setelah ter-*install* programnya, program bahasa R akan tampak seperti pada gambar 2.10. Jika sudah menempuh proses sebelumnya, kita sudah dapat menjalankan kode program bahasa R pada tab *console* yang berfungsi untuk menjalankan perintah per-baris.

### 2.5.3 Contoh Penggunaan Bahasa Pemrograman R

Beberapa contoh pengenalan dasar dari bahasa pemrograman R adalah tentang struktur data, operator, membaca dan menulis file, serta tentang pembuatan dan penggunaan *function*.

#### 1. Struktur Data

Struktur data dari bahasa R memiliki lima jenis, diantaranya adalah variabel, vector, list, matrix dan data frame. Variabel adalah tipe data yang berdiri sendiri atau tidak berkelompok. Pada bahasa R, variabel akan otomatis ditentukan tipe datanya ketika inisialisasi variabel seperti pada bahasa pemrograman lainnya (*char*, *integer*, dll). Vector adalah sekumpulan elemen (variabel) yang tipe datanya sama. Elemen dari vector biasanya disebut dengan *components*. List yaitu struktur data yang mirip dengan vector, tetapi di dalam elemen list diizinkan memiliki tipe data yang berbeda. Matrix adalah kumpulan elemen yang dibentuk berdasarkan dua dimensi. Matrix ini sama dengan matrix dalam matematik, dimana elemen di dalam matrix digambarkan seperti pada notasi matematika biasanya. Data frame adalah struktur data untuk merepresentasikan sebuah tabel. Struktur datanya mirip seperti vector namun memiliki nama untuk kolom dan dapat diisi dengan tipe data selain angka.

#### 2. Operator

Pada R prompt kita memasukkan berbagai macam ekspresi. Setiap simbol atau ekspresi memiliki banyak fungsi contohnya adalah simbol `<-` yang merupakan operator penugasan atau *assignment operator*. Simbol-simbol operator yang digunakan pada bahasa R hampir sama dengan bahasa pemrograman lainnya.. Contoh penggunaan operator pada bahasa R dapat dilihat seperti berikut.

```

> #Memasukan variable ke x dan y
> x <- 10
> y <- 2
>
> #Cetak x dan y
> x
[1] 10
> y
[1] 2

> #Penjumlahan
> x+y
[1] 12

> #Pengurangan
> x-y
[1] 8

> #Perkalian
> x*y
[1] 20

> #Pembagian
> x/y
[1] 5

> #Hasil bagi
> x%%y
[1] 0

> #Pemangkatan
> x^y
[1] 100

```

### 3. Membaca dan Menulis File

Ada beberapa fungsi yang digunakan untuk membaca data kedalam R:

- a. *read.table*, *read.csv*, digunakan untuk membaca data table
- b. *readLines*, untuk membaca garis dari sebuah file teks
- c. *source*, untuk membaca dalam file kode R
- d. *dget*, untuk membaca dalam file kode R
- e. *load*, untuk membaca dalam tempat kerja yang tersimpan
- f. *unserialize*, untukng membaca satu objek R dalam bentuk biner

Kita juga dapat menuliskan objek R kedalam bentuk file. Ada beberapa fungsi yang digunakan untuk menuliskan data ke dalam file:

- a. *write.table*, digunakan untuk menuliskan data tabular atau table ke dalam file teks

- b. *writeLines*, digunakan untuk menuliskan data karakter baris per baris ke dalam sebuah file atau koneksi
- c. *dump*, digunakan untuk membuang sebuah representasi teks dari objek R yang berjumlah banyak
- d. *dput*, digunakan untuk mengeluarkan sebuah representasi teks dari sebuah objek R
- e. *save*, digunakan untuk menyimpan semua nomer dari objek R dalam bentuk biner ke dalam file
- f. *serialize*, untuk mengubah sebuah objek R ke dalam bentuk biner untuk diubah menjadi sebuah koneksi atau file

Contoh penggunaan dalam membaca dan menulis file pada bahasa R dapat dilihat contoh berikut.

```
> #Membaca file dari 'sumber.csv' ke dalam variabel 'data'
> data <- read.csv("sumber.txt")
> #Menulis file 'hasil.txt' dari variabel 'data'
> write(data, file = "hasil.txt")
> #Load file kode.R pada
> source("D:/folder/kode.R")
```

#### 4. Penggunaan *function*

Fungi atau *function* sangat dibutuhkan dalam pemrograman. Contoh dalam membuat dan menggunakan *function* dalam bahasa pemrograman R dapat dilihat pada contoh berikut.

```
> #Membuat fungsi penjumlahan
> Penjumlahan <- function(x, y){
+ return(x+y)
+ }

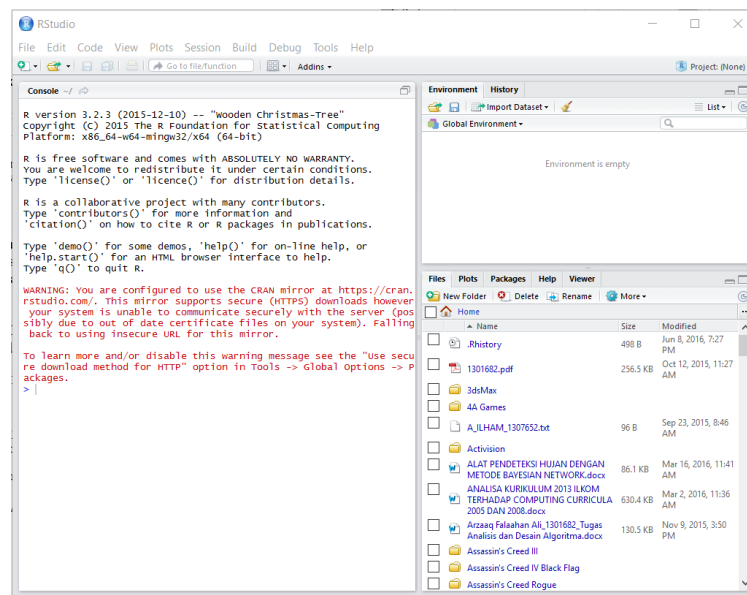
> #Menggunakan fungsi penjumlahan
> Penjumlahan(4, 6)
[1] 10

> #Menggunakan fungsi dengan input variabel dan melempar
  hasil pada variable> x <- 3
> y <- 2
> hasil <- Penjumlahan(x, y)
> hasil
[1] 5
```



### 2.5.4 RStudio

R menyediakan banyak GUI yang berbasis sistem menu, diantaranya R Studio, Tinn-R, R Commander. Berikut ini adalah salah satu contoh GUI dari bahasa R, yaitu RStudio yang digambarkan pada Gambar 2.11. Rstudio dapat diunduh di <https://www.rstudio.com/products/rstudio/download/>.

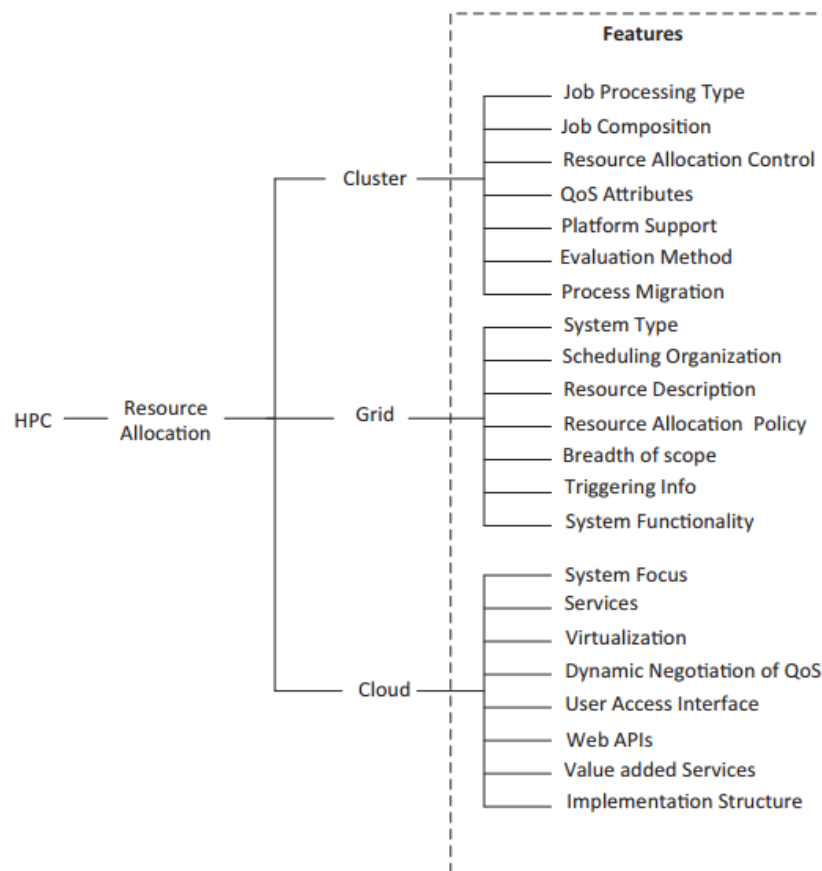


**Gambar 2. 11 Tampilan antar muka Rstudio**

## 2.6 High-Performance Computing

*High-performance computing (HCP)* atau juga sering disebut dengan *parallel computing* atau *distributed computing* adalah konsep usaha untuk menghubungkan kekuatan sumber daya dalam jumlah besar yang terdistribusi dalam sebuah jaringan (Hussain, dkk., 2013). HPC digunakan karena tiga alasan utama. Pertama, sifat aplikasi terdistribusi yang menggunakan komunikasi jaringan yang menghubungkan beberapa komputer. Seperti jaringan yang memproduksi data yang dibutuhkan untuk eksekusi sebuah pekerjaan pada sumber daya jarak jauh. Kedua, banyak dari aplikasi paralel memiliki multi-proses yang menjalankan banyak *nodes* untuk berkomunikasi dalam kecepatan yang sangat tinggi. Penggunaan HPC pada jaringan yang luas dinilai lebih menguntungkan karena memiliki *cost* yang lebih rendah dan membuat performa lebih efisien. Ketiga, ketahanan HPC lebih baik dibandingkan dengan *single-processor*. Kerusakan atau kesalahan pada salah satu *node* tidak akan menghentikan keseluruhan proses jika dibandingkan dengan *single*

CPU. Beberapa teknik ketahanan dalam HPC adalah melakukan *check pointing* dan replikasi data (Amir, Awerbuch, Barak, dan R. Borgstrom, 2000).



**Gambar 2. 12 Kategori dan atribut HPC (Hussain, dkk., 2013)**

Sebuah studi (Hussain, dkk., 2013) mengelompokkan HPC menjadi 3 kategori: *cluster*, *grid*, dan *cloud* berdasarkan pada mekanisme alokasi sumber daya.

Pada Gambar 2.12 digambarkan pembagian kategori beserta atributnya berdasarkan alokasi sumber daya. Secara singkat berikut adalah penjelasan dari ketiga kategori tersebut.

#### 1. *Cluster Computing System*

*Cluster computing* adalah penggunaan multi-komputer. Multi-alat penyimpanan, dan interkoneksi redundan pada suatu sistem (Valentini, dkk., 2013). Tujuan dari *cluster computing* adalah untuk mendesain platform komputasi yang efisien yang menggunakan sumber daya komputer komoditas yang terintegrasi pada perangkat keras, perangkat lunak dan jaringan untuk

meningkatkan performa dan ketahanan sumber daya satu komputer (Pinel, Pecero, Khan dan Bouvry, 2011).

## 2. *Grid Computing Systems*

Konsep dari *grid computing systems* adalah berdasarkan pada penggunaan internet sebagai media untuk ketersediaan yang luas untuk sumber daya komponen komoditas yang murah (Kołodziej, dkk., 2012). Konsep ini juga dapat dikatakan sebagai sistem terdistribusi pada *cluster*. Latar belakang dari *grid computing* adalah untuk pembagian sumber daya dan pemecahan masalah pada multi-institusional dan organisasi virtual dinamis.

## 3. *Cloud Computing System*

Konsep ini menggambarkan model baru dari dunia pelayanan teknologi informasi pada internet (Amir, Awerbuch, Barak, Borgstrom dan Keren, 2000). *Cloud computing* menyediakan *ease-of-access* untuk mengakses situs komputasi secara jarak jauh menggunakan internet (Irwin, Grit dan Chas, 2004).

## 2.7 *Package pbdMPI*

Subbab ini akan membahas mengenai *package pbdMPI* dimulai dari *parallel computing* di R, definisi, instalasi dan contoh penggunaan.

### 2.7.1 *Parallel Computing di R*

Bahasa pemrograman R memiliki banyak *package* untuk melakukan *parallel computing*. Salah satunya adalah ‘snow’ yang merupakan singkatan dari *Simple Network of Workstation* (Rossini, Tierney dan Li, 2007) yang dapat digunakan untuk *parallel computing* sederhana pada bahasa pemrograman R. *Package* ini juga dikembangkan terus menerus hingga terbentuk *package* seperti ‘snowFT’ atau ‘snowfall’ (Knaus, 2008).

Perbedaan paling mencolok dari setiap *package* adalah teknologi yang digunakan seperti ‘Rmpi’ yang menggunakan MPI, ‘rvm’ yang menggunakan PVM, ‘pnmath’ yang menggunakan OpenMP, ‘biopora’ yang menggunakan socket, dll.

**Tabel 2. 3 *Package Parallel Computing* di R berdasarkan pengembangan (Schmidberger, dkk., 2009)**

Package	Version	First Version	Development
<i>Computer Cluster</i>			
Rmpi	0.5-6	2002-05-10	++
rpvm	1.0.2	2001-09-04	++
nws	1.7.0.0	2006-01-28	+
snow	0.3-3	2003-03-10	++
snowFT	0.0.-2	2004-09-11	0
snowfall	1.60	2008-01-19	+
papply	0.1	2005-06-23	0
biopora	1.5	2005-04-22	--
taskPR	0.31	2005-05-12	-
<i>Grid Computing</i>			
GridR	0.8.4	2008-12-04	-
<i>Multi-core systems</i>			
fork	1.2.1	2003-12-19	+

Pada Tabel 2.3 ditunjukkan sebuah studi (Schmidberger, dkk., 2009) tentang perbandingan perkembangan beberapa *package parallel computing* di R. Tanda ‘+’ berarti semakin baik dalam pengembangannya atau bisa dikatakan lebih stabil. Sebaliknya tanda ‘-’ menggambarkan *package* yang masih dikembangkan atau masih belum stabil. Contohnya seperti Rmpi yang versi pertamanya dirulis pada tanggal 10 Mei 2002 memiliki keterangan pengembangan ‘++’ yang berarti sangat baik dan stabil. Sedangkan *package* biopora dinilai kurang baik dalam hal pengembangannya.

### 2.7.2 Definisi pbdMPI

*Package* pbdMPI adalah salah satu dari *Programming with Big Data in R* atau disingkat dengan pbdR (Ostrouchov, Chen, Schmidt dan Patel, 2012). Perbedaan signifikan dari pbdR dan kode R biasa adalah pbdR fokus pada pendistribusian sistem memori, di mana data didistribusikan pada beberapa *processor* dan dianalisa pada setiap cabang dengan komunikasi yang dijalankan oleh MPI yang mempermudah dalam penggunaan HPC.

Terdapat dua implementasi R pada MPI, yaitu Rmpi dan pbdMPI. Rmpi menggunakan *manager/workers parallelism* di mana satu *processor* utama menjadi

kontrol untuk para *workers*. Sedangkan pbdMPI menggunakan SPMD (*Single Program Multi Data*) *parallelism* di mana setiap *processor* adalah *worker* dan memiliki potongan data. Konsep ini memungkinkan setiap *processor* melakukan pekerjaan yang sama pada tiap potongan data yang berbeda untuk data yang sangat besar (Houston, 2007). Keterangan pbdMPI dari laman resmi CRAN yang dapat dilihat pada Tabel 2.4.

**Tabel 2. 4 Keterangan pbdMPI pada laman resmi CRAN**

<b>Title</b>	Programming with Big Data -- Interface to MPI
<b>Depends</b>	R (>= 3.3.0), methods, rlecuyer
<b>Lazy Load</b>	Yes
<b>Lazy Data</b>	Yes
<b>Description</b>	An efficient interface to MPI by utilizing S4 classes and methods with a focus on Single Program/Multiple Data ('SPMD') parallel programming style, which is intended for batch parallel execution
<b>System Requirements</b>	OpenMPI (>= 1.5.4) on Solaris, Linux, Mac, and FreeBSD. MS-MPI (Microsoft MPI v7.1 (SDK) and Microsoft HPC Pack 2012 R2 MS-MPI Redistributable Package) on Windows.
<b>License</b>	Mozilla Public License 2.0
<b>URL</b>	<a href="http://r-pbd.org/">http://r-pbd.org/</a>
<b>Bug Reports</b>	<a href="http://group.r-pbd.org/">http://group.r-pbd.org/</a>
<b>Mailing List</b>	Please send questions and comments regarding pbdR to <a href="mailto:RBigData@gmail.com">RBigData@gmail.com</a>
<b>Needs Compilation</b>	Yes
<b>Maintainer</b>	Wei-Chen Chen
<b>Author</b>	Wei-Chen Chen [aut, cre], George Ostrouchov [aut], Drew Schmidt [aut], Pragneshkumar Patel [aut], Hao Yu [aut], Christian Heckendorf [ctb] (FreeBSD), Brian Ripley [ctb] (Windows HPC Pack 2012), R Core team [ctb] (some functions are modified from the base packages)
<b>Repository</b>	CRAN

Dari tabel 2.4 dapat dilihat bahwa pbdMPI berjalan pada bahasa R dengan versi terendah 3.3.0 dan aplikasi MPI seperti OpenMPI pada Linux/MacOS atau Microsoft MPI pada Windows.

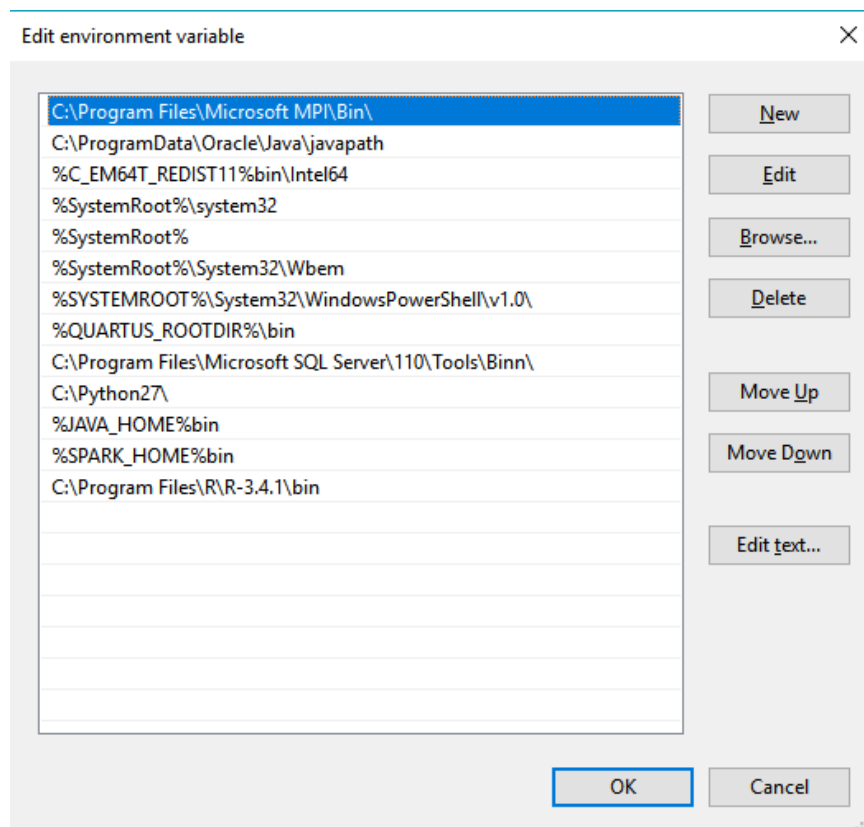
### 2.7.3 Instalasi pbdMPI

Untuk melakukan instalasi pbdMPI pada bahasa pemrograman R dapat dilakukan pada *console* R jika terkoneksi dengan internet. Pada *console* dapat

dituliskan perintah `>install.packages("pbdMPI")` lalu menunggu hingga proses instalasi selesai. Sekarang, *package* pbdMPI sudah ter-*install* pada R.

Langkah selanjutnya adalah mengunduh dan melakukan instalasi MPI (Microsoft MPI jika pada Windows). Untuk versi Windows dapat diunduh di <https://www.microsoft.com/en-us/download/detail.aspx?id=55494>. Lalu lakukan instalasi setelah proses pengunduhan selesai.

Tahap berikutnya adalah melakukan pengaturan *environment variables* untuk Rscript dan mpiexec agar siap digunakan. Untuk Windows, dapat menambahkan direktori penyimpanan Rscript.exe dan mpiexec.exe seperti pada Gambar 2.13.



**Gambar 2. 13 Memasukan direktori Rscript dan mpiexec di path environment variables**

Jika sudah, untuk mengecek apakah langkah sebelumnya sudah berhasil atau tidak dapat melihat dengan menggunakan *terminal/prompt* dengan langkah seperti berikut untuk melihat Rscript.

```

> Rscript
Usage: /path/to/Rscript [--options] [-e expr [-e expr2 ...] |
file] [args]

--options accepted are
  --help             Print usage and exit
  --version          Print version and exit
  --verbose          Print information on progress
  --default-packages=list
                     Where 'list' is a comma-separated set
                     of package names, or 'NULL'
or options to R, in addition to --slave --no-restore, such as
  --save             Do save workspace at the end of the
session
  --no-environ       Don't read the site and user environment
files
  --no-site-file     Don't read the site-wide Rprofile
  --no-init-file     Don't read the user R profile
  --restore          Do restore previously saved objects at
startup
  --vanilla          Combine --no-save, --no-restore, --no-
site-file
                     --no-init-file and --no-environ

'file' may contain spaces but not shell metacharacters
Expressions (one or more '-e <expr>') may be used *instead* of
'file'
See also ?Rscript from within R

```

Kemudian diikuti dengan pengecekan mpiexec seperti berikut,

```

> mpiexec
Microsoft MPI Startup Program [Version 8.1.12438.1084]

Launches an application on multiple hosts.

Usage:

    mpiexec [options] executable [args] [ : [options] exe [args]
: ... ]
    mpiexec -configfile <file name>

Common options:

-n <num_processes>
-env <env_var_name> <env_var_value>
-wdir <working_directory>
-hosts n host1 [m1] host2 [m2] ... hostn [mn]
-cores <num_cores_per_host>
-lines
-debug [0-3]

Examples:

    mpiexec -n 4 pi.exe
    mpiexec -hosts 1 server1 master : -n 8 worker

For a complete list of options, run mpiexec -help2

```

For a list of environment variables, run `mpiexec -help3`

You can reach the Microsoft MPI team via email at `askmpi@microsoft.com`

### 2.7.4 Contoh Penggunaan pbdMPI

Penggunaan pbdMPI mengharuskan adanya kompilasi, berbeda dengan cara penggunaan R pada umumnya yang dapat memanggil fungsi pada *console*. Contoh sederhana dapat dilakukan dengan membuat sebuah file dengan nama “demo.r” yang berisi kode program untuk mencari akar dari angka 1 hingga 1000 secara paralel seperti berikut.

```
### Initial.
suppressMessages(library(pbdMPI, quietly = TRUE))
init()
.comm.size <- comm.size()
.comm.rank <- comm.rank()

### Examples.
N <- 1000
x <- (1:N)
y <- allgather(x, sqrt(x))
comm.print(y)

### Finish.
finalize()

## End(Not run)
```

Lalu pada *terminal/prompt* masuk pada direktori di mana “demo.r” disimpan dan lakukan perintah berikut.

```
mpiexec -np 2 Rscript demo.r
```

Perintah di atas akan mengeksekusi kode program “demo.r” dengan menggunakan 2 *cores*. Untuk menggunakan *cores* dengan jumlah yang berbeda dapat dilakukan dengan mengganti angka ‘2’ pada contoh di atas dengan maksimal jumlah *cores* yang dimiliki pada komputer.

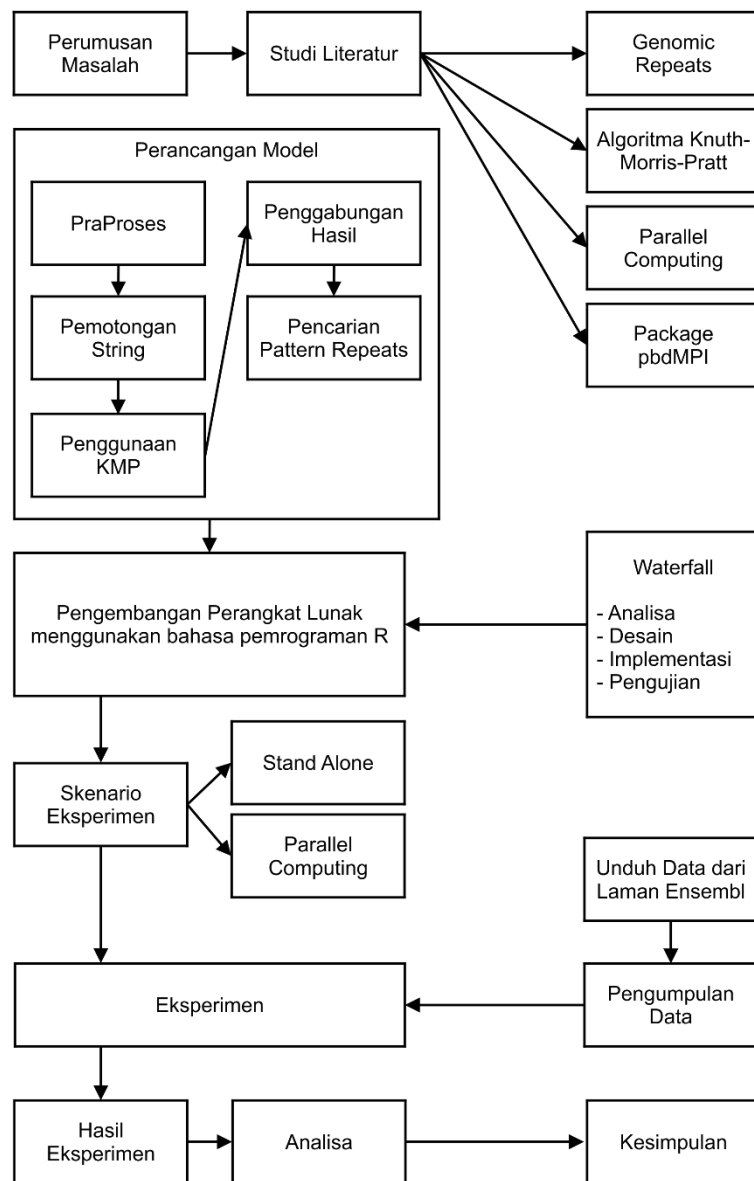


## BAB III

### METODOLOGI PENELITIAN

#### 3.1 Desain Penelitian

Desain penelitian adalah kerangka kerja yang digunakan untuk melakukan penelitian. Pada bagian ini penulis akan memaparkan kerangka kerja dari mulai penelitian sampai selesai. Desain penelitian digambarkan pada Gambar 3.1.



**Gambar 3. 1 Desain proses penelitian deteksi *genomic repeats* menggunakan algoritma Knuth-Morris-Pratt pada R *high-performance computing package***

1. Pada tahap pertama penulis melakukan diskusi untuk menemukan dan menentukan rumusan masalah yang dapat diangkat menjadi topik pada penelitian skripsi ini. Pada tahap ini pula penulis merumuskan tujuan dari penelitian yang berkaitan dengan rumusan masalah yang juga telah tertulis pada bab pertama skripsi ini.
2. Selanjutnya penulis melakukan studi literatur berkaitan dengan topik yang telah disetujui pada tahap pertama. Pada tahap ini dilakukan studi literature tentang *genomic repeats* yang mengandung pengetahuan umum tentang DNA serta *trinucleotide repeat disorders*, penyakit genetik yang disebabkan perulangan sejumlah 3 asam pasang basa. Terdapat pula pengetahuan tentang algoritma *string matching* Knuth-Morris-Pratt, pemrograman bahasa R, *parallel computing* dan tentang *package High-Performance Computing* pada bahasa pemrograman R yang akan digunakan pada penelitian ini, pbdMPI.
3. Tahap berikutnya penulis melakukan pengumpulan data yang dapat diunduh dari *genbank* yang tersedia: FTP NCBI atau FTP Ensembl. Pada penelitian kali ini penulis mengunduh 24 file kromosom (kromosom 1-22, kromosom X dan Y) manusia dari publikasi nomor 88 di laman FTP Ensembl yang dapat diunduh pada [ftp://ftp.ensembl.org/pub/release-88/fasta/homo\\_sapiens/dna/](ftp://ftp.ensembl.org/pub/release-88/fasta/homo_sapiens/dna/)
4. Selanjutnya penulis merancang model *parallel computing* untuk pencarian *pattern* pada *string* menggunakan algoritma Knuth-Morris-Pratt. Rancangan model yang dibuat dimulai dari praproses, pemotorngan *string*, penggunaan algoritma Knuth-Morris-Pratt pada setiap iterasi, penggabungan hasil dari setiap iterasi dan pencarian *pattern* yang berulang.
5. Dilanjutkan dengan membuat program pencarian *pattern* pada *string* dengan menggunakan algoritma Knuth-Morris-Pratt yang diimplementasikan secara paralel dengan menggunakan *package* pbdMPI pada bahasa pemrograman R dari model yang telah dirancang pada proses sebelumnya.
6. Setelah melewati pembuatan program dari analisa hingga pengujian, penulis membuat skenario eksperimen yang akan dilakukan selanjutnya.
7. Penulis melakukan eksperimen komputasi *stand alone* dan *parallel computing* dengan berbagai *core* dan nilai iterasi untuk didapatkannya perbandingan dari setiap satu kali percobaan.

8. Langkah berikutnya penulis melakukan analisa dari hasil eksperimen yang telah dilakukan pada tahap sebelumnya. Pada tahap ini penulis diharapkan mendapatkan grafik dari hasil eksperimen berbagai penggunaan jumlah *core* dan nilai iterasi pada setiap satu kali percobaan.
9. Terakhir penulis akan mendokumentasikan hasil penelitian ini dan menyusun laporan penelitian dalam bentuk skripsi.

### 3.2 Alat dan Bahan Penelitian

Bagian ini menjelaskan secara detail alat dan bahan yang digunakan untuk melakukan penelitian.

#### 3.2.1 Alat Penelitian

1. Perangkat Keras (*Hardware*) yaitu komputer dengan spesifikasi:
  - *Processor Intel® Core™ i7-5820K CPU @ 3.30Ghz (12 CPUs), ~3.3GHz*
  - Memory 32768MB RAM
  - NVIDIA GeForce GTX 980
  - HDD 730 GB
2. Perangkat lunak (*software*) sebagai berikut:
  - Sistem Operasi Windows 10 Pro 64-bit (10.0, Build 14393)
  - R 3.4.0
  - Rstudio 1.0.143
  - Microsoft MPI 8.1.12438.1084
  - Microsoft MPI SDK 8.1.12438.1084
  - Microsoft Office Excel
  - ConEmu 161206.x64

#### 3.2.2 Bahan Penelitian

Bahan yang diperlukan untuk melakukan penelitian yaitu data sekuens DNA manusia yang diunduh dari Ensembl yang dapat diunduh pada [ftp://ftp.ensembl.org/pub/release-88/fasta/homo\\_sapiens/dna/](ftp://ftp.ensembl.org/pub/release-88/fasta/homo_sapiens/dna/) serta beberapa artikel untuk menunjang pengetahuan guna terlaksanakannya penelitian ini.

### 3.3 Metode Penelitian

Adapun metode yang dilakukan dalam penelitian ini dibagi kedalam dua bagian, yaitu metode pengumpulan data dan metode pengembangan perangkat lunak.

#### 3.3.1 Metode Pengumpulan Data

Penulis berusaha mendapatkan data yang valid dan mampu menunjang penelitian. Ada pun metode pengeumpulan data pada penelitian ini adalah sebagai berikut:

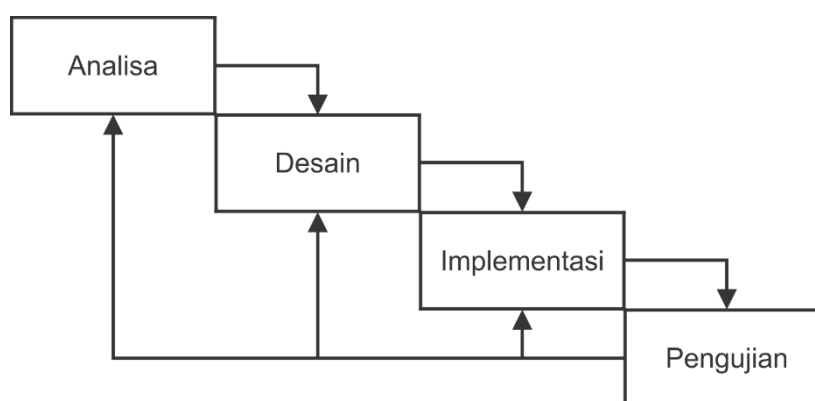
1. Studi Literatur

Studi literatur dilakukan dengan mempelajari teori dan konsep yang menjadi pendukung dalam penelitian ini, yaitu tentang *genomic repeats*, algoritma Knuth-Morris-Pratt, *parallel computing* dan pbdMPI.

2. Mengunduh Data Sekuens

Pengumpulan data sekuens yang akan digunakan pada penilitian ini diunduh dari *genbank* terpercaya, Ensembl. Data yang digunakan dapat diunduh pada sumber yang telah disebutkan pada poin bahan penelitian.

#### 3.3.2 Metode Pengembangan Perangkat Lunak



**Gambar 3. 2 Model Waterfall (Sommerville, 2011)**

Metode pengembangan perangkat lunak dilakukan dengan metode *waterfall*. Model SDLC air terjun (*waterfall*) sering juga disebut model sekuensial linier (*sequential linier*). Model *waterfall* menyediakan pendekatan alur hidup perangkat lunak secara sekuensial atau urut dimulai dari analisis, desain,

pengkodean, pengujian dan tahap *support* (Sukamto dan Shalahuddin, 2011). Penulis menggunakan metode *modern waterfall* seperti pada Gambar 3.2 agar jika suatu saat ada kesalahan pada salah satu tahap, bisa dikembalikan ke tahap sebelumnya. Berikut pengertian dari tahap-tahap pada model *waterfall* pada Gambar 3.2 menurut Ian Sommerville (2011) :

1. *Requirments Analysis and Definition* (Analisa)

Analisis adalah tahap menentukan aplikasi atau *software* seperti apakah yang akan dibuat. Analisis merupakan tahapan penetapan fitur, kendala dan tujuan sistem melalui konsultasi dengan pengguna sistem. Semua hal tersebut akan ditetapkan secara rinci dan berfungsi sebagai spesifikasi sistem. Analisis ini terdiri dari analisis kebutuhan dan analisis pembuatan sistem.

2. *System and Software Design* (Desain)

Dalam tahapan ini akan dibentuk suatu arsitektur sistem berdasarkan persyaratan yang telah ditetapkan. Dan juga mengidentifikasi dan menggambarkan abstraksi dasar sistem perangkat lunak dan hubungan-hubungannya. Desain terdiri dari desain database, desain arsitektur system, dan desain antarmuka (*user interface*)

3. *Implementation and Unit Testing* (Implementasi)

*Coding* adalah tahap proses implementasi dari desain, dalam tahapan ini, hasil dari desain perangkat lunak akan direalisasikan sebagai satu set program atau unit program. Setiap unit akan diuji apakah sudah memenuhi spesifikasinya.

4. *Integration and System Testing* (Pengujian)

Proses testing atau pengujian dilakukan pada logika internal untuk memastikan semua pernyataan sudah diuji. Dalam tahapan ini, setiap unit program akan diintegrasikan satu sama lain dan diuji sebagai satu sistem yang utuh untuk memastikan sistem sudah memenuhi persyaratan yang ada. Setelah itu sistem akan dikirim ke pengguna sistem.

## BAB IV

### HASIL DAN PEMBAHASAN

#### 4.1 Pengumpulan Data

Data yang digunakan pada penelitian ini adalah data sekeuns DNA manusia yang dapat diunduh secara bebas pada laman [ftp://ftp.ensembl.org/pub/release-88/fasta/homo\\_sapiens/dna/](ftp://ftp.ensembl.org/pub/release-88/fasta/homo_sapiens/dna/). Data tersebut adalah contoh sekuens DNA manusia pada publikasi nomor 88 yang disediakan pada situs *File Transfer Protocol* (FTP) Ensembl. Pada laman tersebut terdapat 24 file sekuens kromosom DNA yang dapat dilihat pada Tabel 4.1.

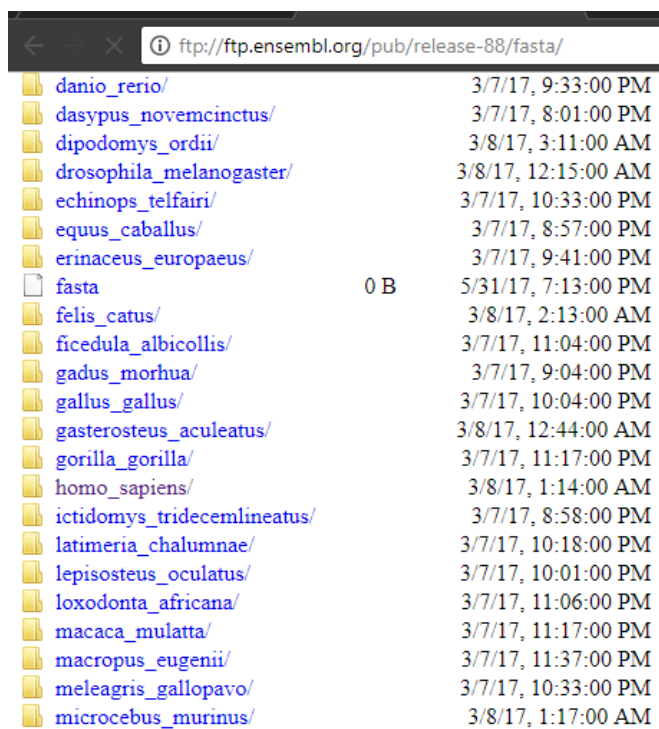
**Tabel 4. 1 File sekuens DNA dari FTP Ensembl**

<b>Nama File</b>	<b>Ukuran File (KB)</b>	<b>Jumlah Pasang Basa</b>
Homo_sapiens.GRCh38.88.chromosome.1.dat	356.015	248.956.422
Homo_sapiens.GRCh38.88.chromosome.2.dat	339.136	242.193.529
Homo_sapiens.GRCh38.88.chromosome.3.dat	277.110	198.295.559
Homo_sapiens.GRCh38.88.chromosome.4.dat	257.483	190.214.555
Homo_sapiens.GRCh38.88.chromosome.5.dat	252.007	181.538.259
Homo_sapiens.GRCh38.88.chromosome.6.dat	239.069	170.805.979
Homo_sapiens.GRCh38.88.chromosome.7.dat	230.205	159.345.973
Homo_sapiens.GRCh38.88.chromosome.8.dat	199.779	145.138.636
Homo_sapiens.GRCh38.88.chromosome.9.dat	190.517	138.394.717
Homo_sapiens.GRCh38.88.chromosome.10.dat	185.415	133.797.422
Homo_sapiens.GRCh38.88.chromosome.11.dat	203.017	135.086.622
Homo_sapiens.GRCh38.88.chromosome.12.dat	195.182	133.275.309
Homo_sapiens.GRCh38.88.chromosome.13.dat	152.170	114.364.328
Homo_sapiens.GRCh38.88.chromosome.14.dat	152.755	107.043.718
Homo_sapiens.GRCh38.88.chromosome.15.dat	145.664	101.991.189
Homo_sapiens.GRCh38.88.chromosome.16.dat	136.950	90.338.345
Homo_sapiens.GRCh38.88.chromosome.17.dat	135.301	83.257.441
Homo_sapiens.GRCh38.88.chromosome.18.dat	112.581	80.373.285
Homo_sapiens.GRCh38.88.chromosome.19.dat	103.644	58.617.616
Homo_sapiens.GRCh38.88.chromosome.20.dat	93.016	64.444.167
Homo_sapiens.GRCh38.88.chromosome.21.dat	67.878	46.709.983
Homo_sapiens.GRCh38.88.chromosome.22.dat	77.049	50.818.468
Homo_sapiens.GRCh38.88.chromosome.X.dat	213.977	156.040.895
Homo_sapiens.GRCh38.88.chromosome.Y.dat	69.038	541.06.423
<b>Total</b>	<b>4.028.943</b>	<b>3.085.148.840</b>

Data pada Tabel 4.1 menunjukkan satu contoh sekuens DNA manusia yang terdiri dari 24 kromosom (kromosom 1-22, X dan Y) beserta dengan ukurannya dalam KB dan jumlah pasang basa yang ada di dalamnya. Data tersebut adalah data yang akan digunakan dalam penelitian ini. Terlihat juga pada Tabel 4.1 bahwa file sekuens dengan ukuran terbesar adalah file kromosom 1 dengan ukuran file sebesar 356.015 KB dengan jumlah 248.956.422 pasang basa di dalamnya. Total ukuran file dari satu genom manusia tersebut adalah sebesar 4.028.943 KB dengan total 3.085.148.840 pasang basa di dalamnya.

#### 4.1.1 Mengunduh Data dari Ensembl

Data yang diambil untuk penelitian ini adalah data yang berasal dari FTP Ensembl yang dapat diunduh pada laman [ftp://ftp.ensembl.org/pub/release-88/fasta/homo\\_sapiens/dna/](ftp://ftp.ensembl.org/pub/release-88/fasta/homo_sapiens/dna/).



**Gambar 4. 1 Spesies yang tersedia pada publikasi nomor 88 FTP Ensembl**

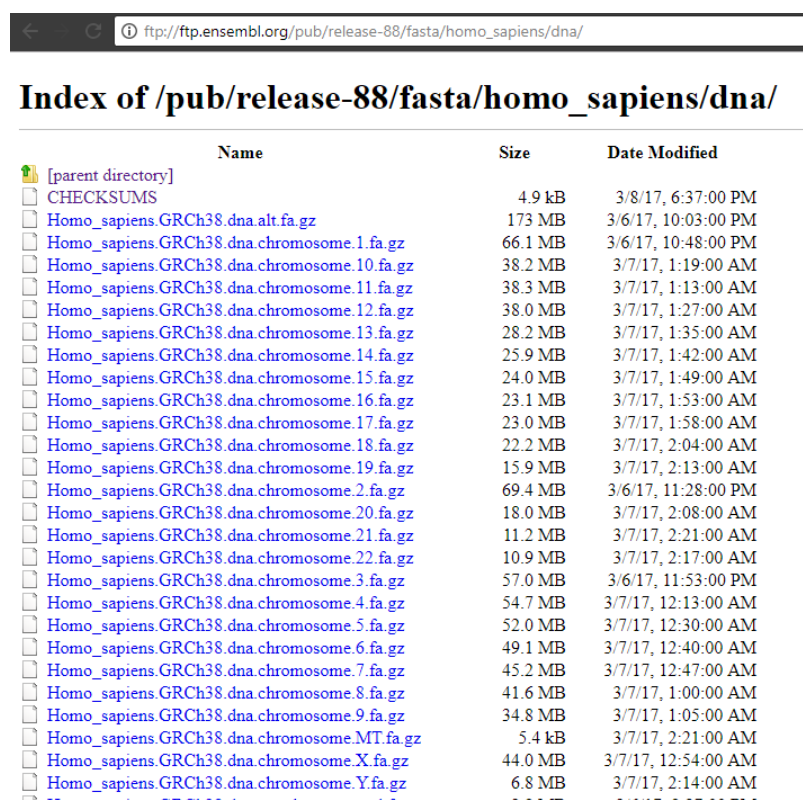
Pada Gambar 4.1 terlihat sebagian data fasta spesies yang tersedia pada publikasi nomor 88 di FTP Ensembl. Karena data yang dibutuhkan dalam penelitian ini adalah data sekuens DNA, maka tipe data yang dipilih adalah fasta. Untuk mencari file DNA pada suatu spesies dari halaman yang diperlihatkan Gambar 4.1 dapat mengikuti alur yang dijabarkan berikut.

```

<species> -- data
            |-- fasta
            |
            |   |-- cdna      [fasta cDNA gene/genscan dumps]
            |   |-- dna       [fasta DNA for assembly]
            |   |-- pep        [fasta peptide gene/genscan
dumps]
            |
            |-- flatfiles
            |
            |   |-- embl       [EMBL format contig dumps]
            |   |-- genbank    [Genbank format contig dumps]
            |
            |-- mysql          [text dumps for Mysql databases]

```

Dari penjabaran di atas dapat dilihat jika ingin mencari file sekuens DNA maka dapat masuk ke folder *Homo sapiens* pada tipe fasta dan masuk ke folder DNA. Di dalam folder DNA terdapat beberapa file kromosom secara terpisah dalam format .fa.gz yang dapat diunduh seperti yang terlihat pada Gambar 4.2.



Name	Size	Date Modified
[parent directory]		
CHECKSUMS	4.9 kB	3/8/17, 6:37:00 PM
Homo_sapiens.GRCh38.dna.alt.fa.gz	173 MB	3/6/17, 10:03:00 PM
Homo_sapiens.GRCh38.dna.chromosome.1.fa.gz	66.1 MB	3/6/17, 10:48:00 PM
Homo_sapiens.GRCh38.dna.chromosome.10.fa.gz	38.2 MB	3/7/17, 1:19:00 AM
Homo_sapiens.GRCh38.dna.chromosome.11.fa.gz	38.3 MB	3/7/17, 1:13:00 AM
Homo_sapiens.GRCh38.dna.chromosome.12.fa.gz	38.0 MB	3/7/17, 1:27:00 AM
Homo_sapiens.GRCh38.dna.chromosome.13.fa.gz	28.2 MB	3/7/17, 1:35:00 AM
Homo_sapiens.GRCh38.dna.chromosome.14.fa.gz	25.9 MB	3/7/17, 1:42:00 AM
Homo_sapiens.GRCh38.dna.chromosome.15.fa.gz	24.0 MB	3/7/17, 1:49:00 AM
Homo_sapiens.GRCh38.dna.chromosome.16.fa.gz	23.1 MB	3/7/17, 1:53:00 AM
Homo_sapiens.GRCh38.dna.chromosome.17.fa.gz	23.0 MB	3/7/17, 1:58:00 AM
Homo_sapiens.GRCh38.dna.chromosome.18.fa.gz	22.2 MB	3/7/17, 2:04:00 AM
Homo_sapiens.GRCh38.dna.chromosome.19.fa.gz	15.9 MB	3/7/17, 2:13:00 AM
Homo_sapiens.GRCh38.dna.chromosome.2.fa.gz	69.4 MB	3/6/17, 11:28:00 PM
Homo_sapiens.GRCh38.dna.chromosome.20.fa.gz	18.0 MB	3/7/17, 2:08:00 AM
Homo_sapiens.GRCh38.dna.chromosome.21.fa.gz	11.2 MB	3/7/17, 2:21:00 AM
Homo_sapiens.GRCh38.dna.chromosome.22.fa.gz	10.9 MB	3/7/17, 2:17:00 AM
Homo_sapiens.GRCh38.dna.chromosome.3.fa.gz	57.0 MB	3/6/17, 11:53:00 PM
Homo_sapiens.GRCh38.dna.chromosome.4.fa.gz	54.7 MB	3/7/17, 12:13:00 AM
Homo_sapiens.GRCh38.dna.chromosome.5.fa.gz	52.0 MB	3/7/17, 12:30:00 AM
Homo_sapiens.GRCh38.dna.chromosome.6.fa.gz	49.1 MB	3/7/17, 12:40:00 AM
Homo_sapiens.GRCh38.dna.chromosome.7.fa.gz	45.2 MB	3/7/17, 12:47:00 AM
Homo_sapiens.GRCh38.dna.chromosome.8.fa.gz	41.6 MB	3/7/17, 1:00:00 AM
Homo_sapiens.GRCh38.dna.chromosome.9.fa.gz	34.8 MB	3/7/17, 1:05:00 AM
Homo_sapiens.GRCh38.dna.chromosome.MT.fa.gz	5.4 kB	3/7/17, 2:21:00 AM
Homo_sapiens.GRCh38.dna.chromosome.X.fa.gz	44.0 MB	3/7/17, 12:54:00 AM
Homo_sapiens.GRCh38.dna.chromosome.Y.fa.gz	6.8 MB	3/7/17, 2:14:00 AM

**Gambar 4. 2 Folder DNA Manusia pada Publikasi Nomor 88 FTP Ensembl**

Setelah diunduh file tersebut dapat diekstraksi menjadi file yang siap dipakai dalam format .dat yang ukuran filenya dapat menjadi lima hingga tujuh kali lipat dari ukuran file saat masih berformat .gz sebelumnya.



#### 4.1.2 Pengertian Format File

File yang diunduh dari FTP Ensembl seperti yang diperlihatkan pada Gambar 4.2 memiliki format file .gz dengan salah satu contohnya adalah “Homo\_sapiens.GRCh38.88.chromosome.1.dat” yang di mana untuk format nama filenya adalah <species>.<assembly>.<sequence type>.<id type>.<id>.fa.gz dengan penjelasan sebagai berikut.

1. <species>  
Nama spesies organisme
2. <assembly>  
Nama pembangun kumpulan sekuens
3. <sequence type>  
Jenis file DNA (murni atau sudah dimodifikasi dengan *RepeatMarker Tool*).  
Jenisnya adlah ‘dna’, ‘dna\_rm’ dan ‘dna\_sm’
4. <id type>  
Sistem koordinat di Ensembl. Di antaranya adalah ‘chromosome’ dan ‘nonchromosomal’
5. <id>  
Identitas sekuens. Jika <id type> adalah ‘chromosome’ dan <id> adalah ‘1’ itu berarti file untuk kromosom 1
6. fa  
Menunjukkan bahwa file tersebut adalah file berbasis FASTA
7. gz  
Format *zip* untuk efisiensi ukuran file

Jika sudah diekstraksi, file akan berformat .dat dengan ukuran yang lebih besar lima hingga tujuh kali dari format file sebelumnya. Format .dat sendiri adalah format file umum yang dapat berupa file apapun bergantung pada kebutuhannya. Format file ini juga biasa digunakan untuk video pada VCD. Namun pada umumnya file dengan format .dat adalah file berisi teks atau biner. Dalam kasus file yang sedang dibahas pada penelitian ini, file .dat adalah file yang berisi teks.

#### 4.1.3 Penjelasan Isi File

Isi dari file .dat tersebut tersusun atas beberapa keterangan serta sekuens DNA sebagai data utama. Contoh potongan dari isi file tersebut adalah sebagai berikut.

```

LOCUS      21 46709983 bp DNA HTG 6-MAR-2017
DEFINITION Homo sapiens chromosome 21 GRCh38 full sequence 1..46709983
reannotated via
            Ensembl
ACCESSION  chromosome:GRCh38:21:1:46709983:1
VERSION    21GRCh38
KEYWORDS   .
SOURCE     human
ORGANISM   Homo sapiens
            Eukaryota; Opisthokonta; Metazoa; Eumetazoa; Bilateria;
            Deuterostomia; Chordata; Craniata; Vertebrata; Gnathostomata;
            Teleostomi; Euteleostomi; Sarcopterygii; Dipnotetrapodomorpha;
            Tetrapoda; Amniota; Mammalia; Theria; Eutheria; Boreoeutheria;
            Euarchontoglires; Primates; Haplorrhini; Simiiformes; Catarrhini;
            Hominoidea; Hominidae.
COMMENT    This sequence was annotated by Ensembl. Please visit the Ensembl or
EnsemblGenomes
            web site, http://www.ensembl.org/ or http://www.ensemblgenomes.org/
for more
            information.
COMMENT    All feature locations are relative to the first (5') base of the
sequence in this
            file. The sequence presented is always the forward strand of the
assembly.
            Features that lie outside of the sequence contained in this file
have clonal
            location coordinates in the format: <clone
accession>.<version>:<start>..<end>
COMMENT    The /gene indicates a unique id for a gene,
/note="transcript_id=..." a unique id
            for a transcript, /protein_id a unique id for a peptide and
note="exon_id=..." a
            unique id for an exon. These ids are maintained wherever possible
between
            versions.
COMMENT    All the exons and transcripts in Ensembl are confirmed by similarity
to either
            protein or cDNA sequences.
<.....>
BASE COUNT 11820664 a      8185244 c      8226381 g      11856330 t      6621364 n
ORIGIN
      1 CCTGATTTTG GCCACTAGGT GGAGTCTGGC TCTAGGGTTT CGAGGCCGCT GGTGTGGTGT
     61 GGCGGAGTCC GGGTTTGCCA CCGCTGCGCT CCATGAGCAG GTAGCAGCTG CAGCGGAGCT
    121 TTAGACCGAG GCTGGCAGGG CTGGCCCCAG ACGGCCTGAG GGTGAGGGAG TGCAGGGTCC
    181 TCCACCCCTA GTCCGCTCTT TCCTTTGCCC TTACCCAGAG CGGGTTGTGC GGGCTCTGGG
    241 CTCGTGTGCG GCGCTGGGCT CTGTGCAGCC GCCGAGATGG GGCTGAGCAG CGGATTTCTT
    301 CCCTGCTGCA GCTGGAGGAC GATTACCTGC ACTAGCCGCT GAGGCCGGCAT CTGGCCCTGG
    361 GTTACTGCAG CTGGTGACGC GGGCAGGGTC AGGGTTGGTT GCAGGTGGCA GTCGCTGCTA
    421 AACCCATTGC GAGCCTCAGG GTCACCAAGT TCACCGTCCT TTCATCATAG TATCTGATCT
    481 TTGCCCCGCG CCCAGAGTGC GGACTGGCCT GCGCTGGGGA CTGCATAGCT TCTGGGGGCC
    541 GGTGACGCCG AGTTTCACGT CCTCCTGCAG CTGCGTGGCC TAAGGTCTTA GGCGCCGCGG
<.....>
46709401 ACATTAGAAC AGCTTTCAGG TCTATCGTGA GAAAGGAAAT ATCTTCAAAT AAAAAGTAGA
46709461 CAGAAGCATT CTGATAAACT TGTTTGTGAA GTGTGATCTC AGCTAACAGA GGTGGATCTT
46709521 TCTTGTGATA GAGCAGTTCT GAAAAACACT TTGTTGAATC TGCAAGTGGA CATTTGGATA
46709581 GATTGAAGA TTTCGTTGGA AACGGGAATA TCTTCATATC AAATCTAGAC AGAAGCATTC
46709641 TCAGAAACGT CTTTGTGATG TTTCGATTCA ACTCATAGAG TTGAACATTC CGTTTCAGAG
46709701 AGCAGCTTTG AAGCACACTT TTTGTAGTAT GTGCAAGGGG ATATTGAGAG CGCTCCGAGG
46709761 CCTAAGGTGA AAAAGCAAAT ATCTTCCCAT AACCACATGA CAGAAACATT GTCAGAAACT
46709821 CCTTTATGAC GTATGCACTC ACCTAACAGA GAAGAACCTT CCTTTTGACA GAGCAGTTTT
46709881 GATACACTCT TTTTGTAGAA TCTGCAAGTG GATATTGGA TAGCTGTGAA GATTTGCTTG
46709941 GAAACGGGAA TATCTTCTTA TAAATCTAG ACAGAAGCAT TCT
//

```

Dari contoh potongan data tersebut dapat dilihat bahwa pada baris pertama terdapat judul 'LOCUS' diikuti dengan deskripsi yang mana '21' bermakna kromosom 21, '46709983 bp' yang berarti pada file tersebut memiliki 46.709.983 pasang basa, '6-MAR-2017' tanggal publikasinya yaitu 6 Maret 2017. Selanjutnya

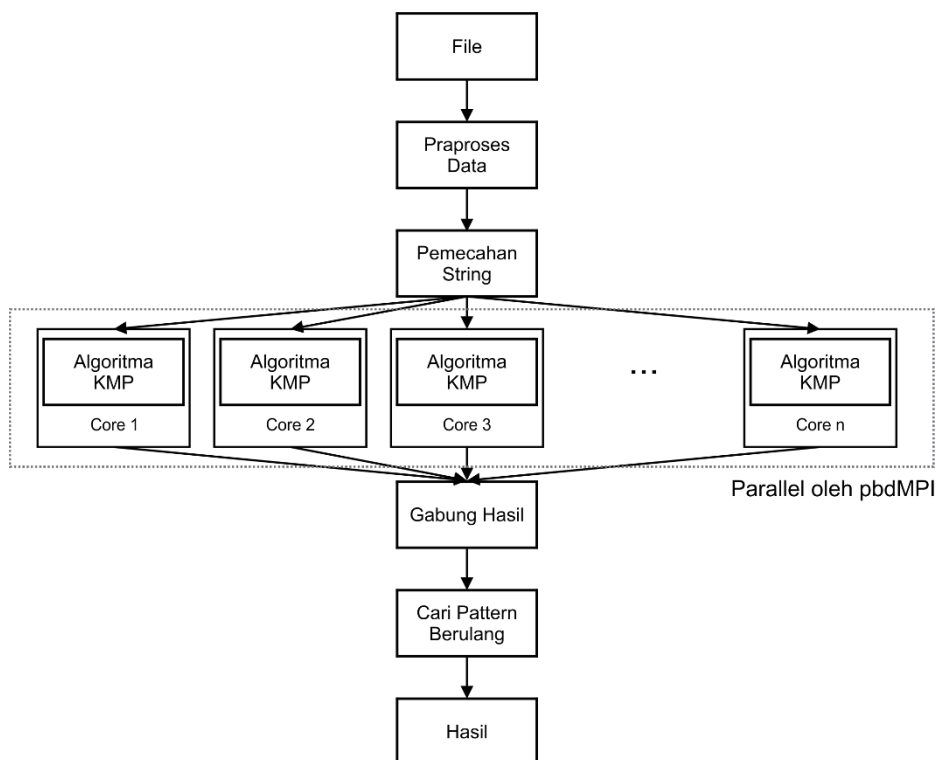
terdapat beberapa keterangan yang menjelaskan bahwa data tersebut adalah data dari *Homo sapiens* serta terdapat beberapa komentar pada judul 'COMMENT'.

Untuk sekuens DNA-nya sendiri terbagi menjadi enam kolom di mana setiap kolom berisi sepuluh karakter sekuens. Di sisi kiri sekuens terdapat indeks yang merujuk pada karakter pertama pada tiap baris di kolom pertama. Terlihat angka 1 berarti menunjukkan indeks ke-1. Jadi pembaca dapat mengetahui indeks karakter tertentu berdasarkan angka yang tertera disamping ditambahkan dengan urutan karakter tersebut dalam satu baris.

Letak dari sekuens DNA pada file tersebut adalah setelah kata 'ORIGIN' dan sebelum tanda '/' yang menjadi akhir pada setiap file serupa. Format tersebut memudahkan peneliti untuk melakukan praproses jika ingin mengambil sekuens DNA-nya saja seperti yang dilakukan pada penelitian kali ini.

## 4.2 Perancangan Model

Dalam implementasi algoritma Knuth-Morris-Pratt pada pbdMPI, penulis merancang sebuah model agar berjalannya konsep komputasi paralel seperti yang diperlihatkan pada Gambar 4.3.



**Gambar 4. 3 Perancangan model *parallel computing* algoritma KMP**

### 4.2.1 Praproses Data

Pertama, file data yang akan digunakan akan melalui tahap praproses untuk menghasilkan *string* yang bersih. Sebelumnya, isi dari file tersebut memiliki informasi-informasi data yang sangat banyak sedangkan dalam penelitian ini hanya akan membutuhkan sekuens yang ada di dalamnya. Sekuens yang dimaksud adalah data huruf-huruf pasang basa ‘A’, ‘C’, ‘G’, ‘T’ yang dimulai setelah kata ‘ORIGIN’ dan sebelum simbol ‘//’ pada isi file tersebut. Setelah itu juga dilakukan praproses untuk menghapus angka-angka dan spasi yang terdapat pada sekuens tersebut. Contoh dari data sekuens yang akan dipraproses adalah sebagai berikut.

```
<.....>
COMMENT      All the exons and transcripts in Ensembl are
confirmed by similarity to either protein or cDNA sequences.
ORIGIN
      1 CTACTGCTGC TACATCTGCT
     11 GTCGAT
//
```

Setelah dilakukan praproses seperti yang dijelaskan sebelumnya, maka hasil praproses dari data yang di atas akan menjadi *string* yang siap digunakan pada tahap selanjutnya, yaitu “CTACTGCTGCTACATCTGCTGTCGAT”. Begitu pula untuk data dalam jumlah besar akan menjadi *string* yang panjangnya dapat mencapai ratusan juta.

### 4.2.2 Pemotongan String Sekuens

Pemotongan atau pemecahan sekuens dilakukan dengan membagi isi dari *string* sebanyak *iterator* atau pemecah yang digunakan. Tiap pecahan akan berjumlah  $n$  karakter yang merupakan hasil bagi dari panjang *string* dibagi dengan *iterator*, sedangkan sisanya akan menjadi jumlah karakter untuk bagian potongan terakhir. Contoh untuk *string* yang berjumlah 26 karakter seperti contoh sebelumnya dengan jumlah *iterator* atau pemecah yang diatur dengan nilai 3, maka pada masing-masing pecahan ke-1 dan ke-2 akan mendapat 8 karakter sedangkan pada pecahan terakhir akan mendapat 10 karakter guna menggenapkan jumlah karakter pada *string* seperti pada ilustrasi Gambar 4.4.

String = 26 karakter

C	T	A	C	T	G	C	T	G	C
T	A	C	A	T	C	T	G	C	T
G	C	T	G	A	T				

Pemecah = 3



Potongan string ke-1

C	T	A	C	T	G	C	T
---	---	---	---	---	---	---	---

Potongan string ke-2

G	C	T	A	C	A	T	C
---	---	---	---	---	---	---	---

Potongan string ke-3

T	G	C	T	G	C	T	G	A	T
---	---	---	---	---	---	---	---	---	---

**Gambar 4. 4 Pemecahan string berdasarkan jumlah iterator**

Namun jika dilihat seksama pada Gambar 4.4 akan terjadi *miss* jika *pattern* yang dicari berada tepat pada pemotongan *string*. Contohnya jika *pattern* yang dicari adalah “AC” maka akan ditemukan pada potongan ke-1. Jika *pattern* yang dicari adalah “TGAT” maka akan ditemukan pada potongan ke-3. Lalu bagaimana jika *pattern* yang dicari adalah “TCT” atau “ATCT”? Tentu saja *pattern* tersebut tidak akan ditemukan pada setiap potongan karena tidak ada *pattern* yang dimaksud, padahal jika pencarian dilakukan pada *string* yang utuh tentu *pattern* tersebut akan ditemukan.

Oleh karena itu, penulis membuat sebuah model pemotongan agar tidak terjadi *miss* seperti yang disebutkan sebelumnya. Model ini dilakukan dalam pengambilan jumlah karakter *string* pada potongan pertama hingga potongan n-1 dengan n adalah nilai *iterator*. Rumus matematis untuk penentuan jumlah karakter yang akan diambil oleh potongan 1 hingga potongan n-1 adalah sebagai berikut.

$$n_{karakter} = \left\lfloor \frac{string}{iterator} \right\rfloor + (n_{pattern} - 1)$$

Sebagai contoh, jika pada Gambar 4.4 *pattern* yang akan dicari adalah “ATCTG” yang berjumlah 5 karakter maka setiap potongan sesuai Gambar 4.4 harus ditambah dengan 4 karakter pertama dari potongan selanjutnya seperti Gambar 4.5 berikut.

String = 26 karakter

C	T	A	C	T	G	C	T	G	C
T	A	C	A	T	C	T	G	C	T
G	C	T	G	A	T				

Pemecah = 3



Potongan string ke-1

C	T	A	C	T	G	C	T	G	C	T	A
---	---	---	---	---	---	---	---	---	---	---	---

Potongan string ke-2

G	C	T	A	C	A	T	C	T	G	C	T
---	---	---	---	---	---	---	---	---	---	---	---

Potongan string ke-3

T	G	C	T	G	C	T	G	A	T
---	---	---	---	---	---	---	---	---	---

**Gambar 4. 5 Penambahan jumlah karakter perpotongan string**

Dengan model pemotongan pada gambar 4.5, jika *pattern* yang akan dicari adalah “ATCTG” maka akan ditemukan pada potongan ke-2 tidak seperti saat model pemotongan pada Gambar 4.4. Contoh lainnya jika *pattern* yang dicari adalah “CTGCT” maka salah satunya akan ditemukan pada potongan ke-2. Dan jika *pattern* yang dicari digeser satu karakter menjadi “TGCTG” maka juga akan ditemukan pada potongan ke-3. Dengan model ini, pemotongan *string* tidak akan menyebabkan *miss pattern* yang terletak tepat pada titik pemotongan.

#### 4.2.3 Penggunaan Algoritma Knuth-Morris-Pratt

Setelah pemotongan *string* menjadi sejumlah bagian, maka dilakukanlah pencarian *pattern* pada *string* dengan menggunakan algoritma Knuth-Morris-Pratt yang sudah dijelaskan pada BAB II untuk setiap potongannya. Tahap ini dilakukan

secara paralel oleh setiap *cores* yang jumlahnya dapat ditentukan dengan memperhitungkan sumber daya yang ada. Dengan jumlah karakter pada tiap potongan *string* yang jauh lebih sedikit, maka setiap tahap pencarian *pattern* pada *string* juga akan berjalan lebih cepat.

Keluaran dari tahap ini adalah *index* dari setiap *pattern* yang ditemukan. Namun karena *string* yang digunakan pada setiap tahap ini seolah *string* yang baru dan lebih pendek, maka *index* dari *string* di setiap potongan berawal dari 1 seperti pada gambar 4.6 berikut.

String = 26 karakter

C	T	A	C	T	G	C	T	G	C
T	A	C	A	T	C	T	G	C	T
G	C	T	G	A	T				

= 4, 7, 16, 19, 22

Pemecah = 3



Potongan string ke-1

C	T	A	C	T	G	C	T	G	C	T	A
---	---	---	---	---	---	---	---	---	---	---	---

= 4, 7

Potongan string ke-2

G	C	T	A	C	A	T	C	T	G	C	T
---	---	---	---	---	---	---	---	---	---	---	---

= 8

Potongan string ke-3

T	G	C	T	G	C	T	G	A	T
---	---	---	---	---	---	---	---	---	---

= 3, 6

**Gambar 4. 6 Keluaran fungsi KMP string utuh dan potongan string**

Contoh pada gambar 4.6 dapat terlihat pencarian *pattern* “CTG” pada *string* “CTACTGCTGCTACATCTGCTGTCGAT” yang mengeluarkan hasil 4, 7, 16, 19, 22 sebagai *index pattern* “CTG” yang ditemukan pada *string* tersebut. Namun jika kita bekerja dengan panjang *string* yang sangat besar maka kita harus melakukannya secara paralel dengan didahului oleh pemotongan seperti yang sebelumnya telah dijabarkan. Tapi jika dilakukan paralel dengan pemotongan seperti sebelumnya, terlihat pada Gambar 4.6 dengan contoh sederhana setiap

potongan akan memberikan hasil *index* berdasarkan *index* potongan *string*, bukan keseluruhan *string*.

Oleh karena itu, penulis membuat sebuah model untuk mengubah *index* pada hasil potongan ke-2 hingga potongan ke-n dengan *index pattern* yang sebenarnya pada saat keadaan *string* utuh dengan membuat sebuah *adder*. Variabel tersebut akan ikut serta pada proses pencarian *pattern* pada *string* dengan menggunakan algoritma Knuth-Morris-Pratt dan digunakan sebagai penambah *index* yang dihasilkan.

Jumlah *adder* yang diperlukan adalah sebanyak potongan-1. Nilai *adder* sendiri adalah nilai jumlah karakter yang digunakan dalam perhitungan seperti yang diperlihatkan pada Gambar 4.4 dikalikan dengan urutan masing-masing potongan yang dapat dirumuskan menjadi seperti berikut.

$$adder[i] = \left\lfloor \frac{string}{iterator} \right\rfloor \times i$$

Sehingga dalam contoh kasus pada Gambar 4.6, terdapat dua *adder*, yaitu 8 untuk potongan ke-2 dan 16 untuk potongan ke-3. Dengan begitu, keluaran dari setiap bagian potongan tersebut seperti pada Gambar 4.7.

Potongan string ke-1

C	T	A	C	T	G	C	T	G	C	T	A
---	---	---	---	---	---	---	---	---	---	---	---

= 4, 7

Potongan string ke-2

G	C	T	A	C	A	T	C	T	G	C	T
---	---	---	---	---	---	---	---	---	---	---	---

= (8) + 8  
= 16

Potongan string ke-3

T	G	C	T	G	C	T	G	A	T
---	---	---	---	---	---	---	---	---	---

= (3, 6) + 16  
= 19, 22

**Gambar 4. 7 Hasil keluaran dari tiap potongan setelah pemberian adder**

#### 4.2.4 Penggabungan Hasil

Setelah keluaran dari setiap proses pencarian *pattern* pada *string* dengan menggunakan algoritma Knuth-Morris-Pratt sudah menghasilkan *index* yang benar, maka langkah selanjutnya adalah menggabungkan hasil dari setiap potongan yang



dikerjakan oleh setiap *core* menjadi sebuah *list* yang pada langkah selanjutnya akan dicari pada *index* berapa perulangan *pattern* secara berurutan terpanjang terjadi.

#### 4.2.5 Pencarian *Pattern* Berulang

Pada tahap akhir ini akan dicari *pattern* yang muncul bersampingan seperti permasalahan penelitian ini untuk mencari *pattern* berulang penyebab penyakit genetik. Dalam pencarian ini akan ditelusuri *index pattern* yang saling berdekatan. Jika tidak ada yang berdekatan sama sekali, maka hasil dari tahap ini akan mengeluarkan 0. Jika memiliki satu rangkaian perulangan berapa pun jumlahnya, maka *index-index pattern* itu lah yang akan dijadikan sebagai hasil tahap ini. Jika terdapat lebih dari satu, maka tahap ini akan mengeluarkan hasil *index-index pattern* dengan jumlah perulangan *pattern* terbanyak.

Contoh dalam kasus Gambar 4.7, tahap ini akan mengeluarkan hasil 16, 19, 22 sebagai *index* di mana ditemukannya “CTG” dengan perulangan terpanjang, yaitu sebanyak tiga kali perulangan *pattern* “CTG” atau “CTGCTGCTG”.

### 4.3 Pengembangan Perangkat Lunak

Pada tahap pengembangan perangkat lunak, dilakukan implementasi algoritma Knuth-Morris-Pratt pada R *Package High-Performance Computing* pbdMPI dengan model yang telah dirancang pada subbab 4.2 Perancangan Model. Pengembangan perangkat lunak akan dijelaskan ke dalam beberapa subsubbab berdasarkan metode *waterfall*.

#### 4.3.1 Analisa

Program Deteksi *Genomic Repeats* yang akan dikembangkan adalah program untuk mendeteksi perulangan *pattern* tertentu dalam sekuens DNA manusia yang mana jika jumlahnya memenuhi kondisi tertentu akan mengakibatkan penyakit genetik tertentu. Program ini dijalankan secara paralel dengan menggunakan *package* dari bahasa pemrograman R, pbdMPI dengan algoritma pencarian yang digunakan untuk mencari *pattern* dalam *string* adalah algoritma Knuth-Morris-Pratt.

Data yang akan digunakan sebagai objek program ini adalah data yang telah dijelaskan pada subbab 4.1 Pengumpulan Data serta alat yang digunakan adalah alat yang telah disebutkan pada subbab 3.2 Alat dan Bahan Penelitian.

#### 4.3.2 Desain

Program ini dibuat berdasarkan *Command Line Interface* (CLI) yang dapat digunakan pengguna pada *terminal/prompt* yang tersedia. Fungsi-fungsi yang ada akan dibuat menyesuaikan dengan perancangan model yang telah dibangun pada subbab 4.2. Perancang Model. Dalam program ini pun memiliki batasan-batasan tertentu di antaranya adalah sebagai berikut.

1. Untuk menentukan file data yang akan digunakan, *pattern* yang akan dicari dan nilai *iterator* yang akan digunakan harus ditulis dalam kode program.
2. Data yang dapat digunakan dalam program ini adalah data dengan format standar NCBI/Ensembl karena harus ada penyesuaian dalam hal praproses data.
3. Untuk sementara program ini baru hanya diuji untuk sistem operasi Windows mengingat penyesuaian dalam penggunaan MPI.
4. Program ini menggunakan bahasa pemrograman R dengan *package* pbdMPI yang mengharuskan pengguna telah meng-*install package* pbdMPI dalam bahasa R-nya serta telah meng-*install* Microsoft MPI.
5. Karena menggunakan algoritma Knuth-Morris-Pratt, minimal jumlah karakter *pattern* yang akan dicari adalah sebanyak dua.
6. Karena data yang digunakan berukuran besar, maka disarankan komputer yang digunakan untuk melakukan eksperimen harus memiliki spesifikasi kelas menengah.
7. Karena program ini sebatas eksekusi kode program yang telah dibuat, maka tampilan dari program ini hanya sebatas pada *terminal/prompt*.

#### 4.3.3 Implementasi

Program ini ditulis menggunakan bahasa pemrograman R dengan telah mempersiapkan bahasa R yang ter-*install package* pbdMPI dan Microsoft MPI.

Kode program dimulai dengan *load package* pbdMPI, inisiasi dan deklarasi variabel global untuk komputer pekerja seperti berikut.

```
#Load Package pbdMPI
suppressMessages(library(pbdMPI, quietly = TRUE))

#Init and declare global variable about comm size and comm rank
init()
.comm.size <- comm.size()
.comm.rank <- comm.rank()
```

#### 4.3.3.1 Masukan Pengguna

Jika pbdMPI sudah siap, maka langkah berikutnya adalah menerima inputan dari pengguna terkait nama file, *pattern* yang ingin dicari dan juga nilai *iterator* yang akan digunakan untuk komputasi seperti berikut.

```
#Set the data directory
#Example: setwd("C:/Document/Code")
setwd("F:/Kuliah/Semester 8/DATA/HUMAN/88")

#Set 'file' variable with the file data
#Example: file <- "data.txt"
file <- "Homo_sapiens.GRCh38.88.chromosome.22.dat"

#Set the pattern
#Example: pattern <- "gat"
pattern <- "ttaggg"
pattern <- toupper(pattern)

#Set the splitter number for the iterator
#Example: iterator <- 100
iterator <- 100
```

Pengguna diminta untuk memasukkan nama file pada variabel ‘file’, *pattern* yang akan dicari pada variabel ‘pattern’ dan nilai *iterator* pada variabel ‘iterator’ dengan terlebih dahulu set direktori file yang akan digunakan. *Pattern* yang dimasukkan akan diubah menjadi huruf kapital guna menyesuaikan dengan data yang digunakan.

#### 4.3.3.2 Penanganan Kesalahan

Penulis juga membuat penanganan kesalahan untuk program seperti kondisi jika file yang dimaksud tidak ada, set nilai *iterator* yang terlalu kecil atau tidak valid seperti berikut.

```

if(file.exists(file) && nchar(pattern) >= 2 &&
is.numeric(iterator) == TRUE && iterator >= 1){
<AKSI>
}else if(nchar(pattern) >= 2 && is.numeric(iterator) == TRUE &&
iterator >= 1){
  comm.cat("ERROR! File is not exists!")
}else if(is.numeric(iterator) == TRUE && iterator >= 1){
  comm.cat("ERROR! Pattern length is less than 2!")
}else if(iterator >= 1){
  comm.cat("ERROR! Set of iterator is not a number!")
}else{
  comm.cat("ERROR! Set of iterator is less than 1!")
}

```

#### 4.3.3.3 Praproses Data

Seperti model yang telah dirancang, data yang digunakan akan melalui tahap praproses untuk mendapatkan *string* yang bersih dan siap digunakan. Untuk melakukan praproses penulis membuat sebuah fungsi seperti berikut.

```

PreProcessing <- function(file){

  find_start <- unlist(gregexpr("ORIGIN", readChar(file,
file.info(file)$size)))
  find_end <- unlist(gregexpr("\n/", readChar(file,
file.info(file)$size)))

  string <- gsub(" |\r|\n|ORIGIN|[0-9]", "",
substr(readChar(file, file.info(file)$size), find_start,
find_end))

  return(string)
}

string <- PreProcessing(file)

```

Praproses dilakukan dengan mengambil sekuens dengan patokan sesudah kata ‘ORIGIN’ dan sebelum ‘//’. Penulis membuat sebuah penandaan indeks pada batas sekuen tersebut lalu menggunakan `gsub` untuk memasukannya pada variabel ‘string’ dengan terlebih dahulu dilakukan pembersihan karakter-karakter lain selain yang dibutuhkan termasuk spasi. Fungsi tersebut melempar *string* pada variabel ‘string’ yang akan digunakan dalam komputasi.

#### 4.3.3.4 Pembentukan *Prefix*

Setelah mendapatkan *string* yang bersih dan siap digunakan, langkah selanjutnya adalah membentuk *prefix* dari *pattern* yang telah dimasukkan oleh

pengguna. *Prefix* ini digunakan dalam pencarian menggunakan algoritma Knuth-Morris-Pratt seperti yang dijelaskan pada subbab 2.4 Algoritma Knuth-Morris-Pratt. Pada tahap ini penulis membuat sebuah fungsi yang mengimplementasikan algoritma pembentukan *prefix* seperti berikut.

```
#Function for get the prefix from the pattern
KMP_Prefix <- function(pattern) {

  #declare variable
  pattern <- unlist(strsplit(pattern, ""))
  n_pattern <- length(pattern)
  prefix <- c(0)
  a <- 0

  #pattern making
  for(b in 2:n_pattern){
    while(a > 0 && pattern[a+1] != pattern[b]){
      a <- prefix[a]
    }
    if(pattern[a+1] == pattern[b]){
      a <- a+1
    }
    prefix[b] <- a
  }

  #return the result
  return(prefix)
}

prefix <- KMP_Prefix(pattern)
```

Masukan dari pengguna adalah *pattern* yang mana fungsi yang dibuat akan menghasilkan *prefix* berupa *vector* yang berisi angka. Contoh jika *pattern* adalah ‘CCG’ maka fungsi di atas akan menghasilkan ‘0 1 0’ yang akan digunakan pada fungsi pencarian utama menggunakan algoritma Knuth-Morris-Pratt.

#### 4.3.3.5 Pemotongan *String*

Selanjutnya adalah implemtasi pemotongan *string* dari model yang telah dirancang sebelumnya. Pada tahap ini penulis juga membuat fungsi yang siap dipanggil untuk dilakukannya komputasi secara paralel mengingat pada fungsi tersebutlah pemotongan *string* menjadi beberapa bagian yang akan dijadikan masukan tiap komputasi dibuat. Potongan kode program untuk tahap ini adalah sebagai berikut.

```
#Variable that used for the parallel computation
lengthstring <- nchar(string)
```

```

split <- floor(lengthstring/iterator)
splitter <- c()
for(i in 1:(iterator-1)){
  splitter[i] <- split * i
}
iteratorhandle <- 0
splitplus <- nchar(pattern) - 1

#Computation
Parallel <- function(v){

  if(v == 1){
    KMP(substring(string, 1, splitter[v]+splitplus), pattern,
prefix, 0)
  }else if(v == iterator){
    KMP(substring(string, splitter[v-1]+1, lengthstring),
pattern, prefix, splitter[v-1])
  }else{
    KMP(substring(string, splitter[v-1]+1,
splitter[v]+splitplus), pattern, prefix, splitter[v-1])
  }

}

#Computation for handled condition
ParallelHandle <- function(v){

  if(v == 1){
    KMP(substring(string, 1, splitter[v]+splitplus), pattern,
prefix, 0)
  }else if(v == iteratorhandle){
    KMP(substring(string, splitter[v-1]+1, lengthstring),
pattern, prefix, splitter[v-1])
  }else{
    KMP(substring(string, splitter[v-1]+1,
splitter[v]+splitplus), pattern, prefix, splitter[v-1])
  }

}

```

Penulis membuat fungsi ‘Parallel’ yang didalamnya terdapat aksi eksekusi pencarian menggunakan algoritma Knuth-Morris-Pratt dengan masukan adalah *pattern* yang akan dicari, *prefix* dan *string* yang telah dipotong sesuai model yang telah dirancang sebelumnya. Penulis juga membuat fungsi ‘ParallelHandle’ untuk penanganan jika pengguna memberikan nilai *iterator* terlalu kecil berbanding dengan panjang *string* yang digunakan. Kondisi penggunaan fungsi tersebut akan dijelaskan pada tahap berikutnya.

#### 4.3.3.6 Eksekusi *Parallel Computing*

Tahap ini adalah eksekusi komputasi secara paralel (*parallel computing*) akan dilakukan. Untuk nilai *iterator* yang terlalu kecil, penulis membuat komputasi

dilakukan dengan nilai *iterator* terkecil agar setiap potongan komputasi hanya mengerjakan potongan *string* dengan maksimal panjangnya sebanyak lima juta karakter. Berikut adalah kode program untuk eksekusi *parallel computing*.

```
#Instruction for computation
if(split < 5000000){
  time <- system.time(pbdresult <- pbdLapply(1:iterator,
Parallel))
}else{
  iteratorhandle <- floor(lengthstring/5000000)
  time <- system.time(pbdresult <- pbdLapply(1:iteratorhandle,
ParallelHandle))
  comm.cat("WARNING: The iterator is too small for the string.
This program sets the iterator automatically by:",
iteratorhandle, "\n")
}
```

Dalam kode program di atas terlihat pemanggilan fungsi ‘Parallel’ dan ‘ParallelHandle’ yang di mana di dalamnya terdapat pemanggilan fungsi utama yaitu pencarian menggunakan algoritma Knuth-Morris-Pratt.

#### 4.3.3.7 Pencarian dengan Algoritma Knuth-Morris-Pratt

Masukan dari fungsi ini adalah potongan *string*, *pattern* dan *prefix* yang akan menghasilkan *list* berupa *index* ditemukannya *pattern* pada *string*. Fungsi ini dijalankan secara paralel pada setiap *core* yang mengerjakannya. Implementasi dari fungsi ini adalah sebagai berikut.

```
#Function to search the pattern in the string
KMP <- function(string, pattern, prefix, adder){

  #inisiasi variabel
  string <- unlist(strsplit(string, ""))
  pattern <- unlist(strsplit(pattern, ""))
  prefix <- prefix
  adder <- adder
  n_string <- length(string)
  n_pattern <- length(pattern)
  index <- c()
  total <- 0
  i <- 0

  #Perulangan sesuai dengan jumlah string
  for(j in 1:n_string){
    while(i > 0 && pattern[i+1] != string[j]){
      i <- prefix[i]
    }
    if(pattern[i+1] == string[j]){
      i <- i+1
    }
  }
```

```

    if(i == n_pattern){
      index <- c(index, j-n_pattern+1)
      total <- total+1
      i <- prefix[i]
    }
  }
  return(index + adder)
}

```

Masukan dari fungsi ini adalah potongan *string*, *pattern* dan *prefix* yang akan menghasilkan *list* berupa *index* ditemukannya *pattern* pada *string*. Fungsi ini dijalankan secara paralel pada setiap *core* yang mengerjakannya. Implementasi dari fungsi ini adalah sebagai berikut.

#### 4.3.3.8 Penggabungan Hasil

Setelah mendapatkan hasil dari setiap potongan komputasi yang dijalankan secara paralel, tahap selanjutnya adalah menggabungkan hasil dari tiap potongan komputasi tersebut yang dalam bentuk *list* menjadi satu *vector* yang dimasukkan dalam suatu variabel seperti berikut.

```

pbdresultgather <- as.vector(unlist(pbdresult), mode = "integer")

```

#### 4.3.3.9 Pencarian *Pattern* Berulang

Langkah berikutnya adalah mencari *pattern* yang berulang dilihat dari *index* dan panjang *pattern* yang digunakan. Fungsi ini akan melempar hasil perulangan *pattern* paling panjang seperti yang telah dijelaskan pada subsubsubbab 4.2.5 Pencarian *Pattern* Berulang. Berikut adalah impelementasinya pada bahasa R.

```

#Function for checking the repeating pattern
Con <- function(output, pattern){

  max <- c()
  temp <- c()

  for(i in 1:length(output)){

    if(is.na(output[i+1]) == FALSE && (output[i] +
nchar(pattern)) == output[i+1]){
      if(length(temp) == 0){
        temp <- c(output[i], output[i+1])
      }else{
        temp <- c(temp, output[i+1])
      }
    }else{

```



```

        if(length(max) < length(temp)){
            max <- c(temp)
        }
        temp <- c()
    }
    return(max)
}
time2 <- system.time(continuous <- Con(pbdresultgather,
pattern))

```

#### 4.3.3.10 Pencetakan Hasil

Setelah melakukan semua tahapan, akhirnya program akan mencetak hasil pada *console* yaitu berupa keterangan nama tugas pencarian *pattern* tertentu pada file tertentu, jumlah pasang basa yang terdapat pada *string*, jumlah *pattern* yang ditemukan di dalam *string*, nilai perulangan *pattern* terpanjang dan *index* dari perulangan *pattern* terpanjang dan waktu untuk dilakukannya komputasi seperti yang diimplementasikan pada potongan kode program berikut.

```

comm.cat("\nTask:\nSearching", pattern, "in", file, "\n\nNumber
of base pair:\n", lengthstring, "\n\nNumber of pattern in
string:\n", length(pbdresultgather), "\n\nLongest repeating
pattern:\n", length(continuous), "\n\nIndex longest repeating
pattern:\n", continuous, "\n\nTime for computation:\n")

comm.print(time + time2)

```

#### 4.3.4 Pengujian Program

Pada tahap pengujian, penulis menggunakan metode *black box* untuk memastikan bahwa semua fungsi dan *error handling* telah berjalan dengan baik. Pengujian dengan metode *black box* dilakukan oleh penulis dengan memperhatikan setiap fungsi dan *error handling* yang telah dibuat. Hasil pengujian terhadap sistem dapat dilihat pada Tabel 4.3.

Pengujian yang dilakukan penulis adalah tentang beberapa hal yang mungkin akan dilakukan pengguna yang akan menyebabkan pemrosesan tidak berjalan dengan baik. Contohnya seperti kesalahan penulisan nama file yang akan digunakan atau belum melakukan pengaturan direktori aktif pada kode program sehingga nama file yang dituju tidak tersedia. Maka dari itu program akan mengeluarkan informasi bahwa file yang dituju tidak eksis.

Tabel 4. 2 Pengujian *Error Handling* pada Program

No	Item Uji	Hasil yang Diharapkan	Hasil Nyata	Hasil Pengujian
1	Eksistensi File	Error bahwa nama file yang dituju tidak ada	Mengeluarkan "ERROR! File is not exists!"	Sesuai
2	Minimal nilai <i>iterator</i>	Melakukan proses dengan <i>iterator</i> minimal jika pengguna set nilai <i>iterator</i> terlalu kecil	Mengeluarkan "WARNING: The iterator is too small for the string. This program sets the iterator automatically by: (jumlah iterator minimal)"	Sesuai
3	Minimal panjang <i>pattern</i>	Error bahwa minimal panjang <i>pattern</i> adalah 2.	Mengeluarkan "ERROR! Pattern length is less than 2!"	Sesuai
4	Validasi nilai <i>iterator</i>	Error jika pengguna set nilai <i>iterator</i> bukan dengan angka	Mengeluarkan "ERROR! Set of iterator is not a number!"	Sesuai

Dengan jumlah data yang besar, penentuan nilai *iterator* bisa saja terlalu kecil yang menyebabkan tiap potongan *string* tetap memiliki ukuran yang besar. Contohnya jika panjang karakter pada *string* adalah sebesar seratus juta namun pengguna hanya mengatur nilai *iterator* hanya dua, maka tiap potongan yang akan dilakukan pemrosesan secara paralel tetap menanggung *string* dengan panjang karakter sebesar lima puluh juta yang mana jumlah tersebut masih merupakan jumlah yang sangat besar.

Oleh karena itu, penulis membuat sebuah *handling* untuk masalah yang disebutkan sebelumnya. Penulis membuat sebuah kondisi di mana maksimal panjang karakter dari *string* di setiap potongan adalah sebesar lima juta karakter. Dengan begitu, jika panjang karakter *string* yang akan diproses adalah sebesar seratus juta sedangkan pengguna mengatur nilai *iterator* dengan angka yang terlalu kecil seperti dua, maka program otomatis akan tetap menjalankan proses dengan maksimal panjang karakter tiap potongan *string* sebesar lima juta.

Itu berarti pada kasus yang sebelumnya disebutkan, program ini otomatis akan dieksekusi dengan nilai *iterator* sebesar dua puluh. Jadi, pada kasus tersebut jika pengguna mengatur nilai *iterator* dengan kurang dari dua puluh, program akan otomatis menjalankan pemrosesan dengan nilai *iterator* dua puluh berikut dengan peringatan untuk memberitahu pengguna bahwa nilai *iterator* yang digunakan dalam pemrosesan tidaklah menggunakan nilai yang diatur oleh pengguna.

Begitu pula dalam hal *pattern* yang akan dicari di mana panjang karakter *pattern* harus memiliki panjang minimal dua karakter dikarenakan program ini menggunakan algoritma Knuth-Morris-Pratt. Jika tidak sesuai, seperti sebelumnya program akan mengeluarkan informasi tentang batasan program yang dimaksud.

#### 4.4 Rancangan Skenario Eksperimen

Dalam melakukan eksperimen, penulis terlebih dahulu melakukan perancangan skenario untuk eksperimen. Penulis ingin melakukan dua eksperimen. Pertama adalah mencari *pattern* pada *string* menggunakan algoritma Knuth-Morris-Pratt dengan *stand alone*, kedua mencari *pattern* pada *string* menggunakan algoritma Knuth-Morris-Pratt secara paralel dengan program yang dibuat dengan beberapa pergantian variabel *iterator* dan jumlah *core* yang digunakan.

*Pattern* yang akan digunakan adalah ‘CCG’, *pattern* yang jika ditemukan berulang sebanyak 200-900 kali maka dapat disimpulkan manusia tersebut memiliki penyakit *Fagile XE Syndrome* yang pada normalnya *pattern* ‘CGG’ hanya berulang 4-39 kali. Selanjutnya ada *pattern* ‘CAG’ yang merupakan penyebab terjadinya penyakit yang termasuk kategori *polyglutamine*. Perbedaan pada *pattern* ‘CCG’ dan ‘CAG’ tidak hanya terletak pada perbedaan satu karakter di tengah, tapi juga memiliki perbedaan pada *prefix* yang dihasilkan. *Prefix* untuk ‘CCG’ adalah ‘0 1 0’, sedangkan *prefix* untuk ‘CAG’ adalah ‘0 0 0’. Peneliti ingin mengetahui perbedaan kecepatan yang ditimbulkan oleh *prefix*. Selanjutnya ada *pattern* ‘TTAGGG’ yaitu sebuah *telomere* atau bagian paling ujung dari DNA linear. Pencarian ‘TTAGGG’ dimaksudkan untuk melihat pengaruh panjang *pattern* terhadap perbedaan kecepatan komputasi yang dilakukan. Selain itu pemilihan ‘TTAGGG’ juga karena dipastikan ada di setiap sekuens DNA manusia.

Data yang digunakan pada penelitian ini hanya tiga file dengan pertimbangan perbedaan setiap file yang dipakai adalah kurang lebih 100 MB untuk mengetahui pengaruh ukuran file pada kecepatan komputasi yang dilakukan. Data yang digunakan adalah file kromosom 1 (356.015 KB), file kromosom 4 (257.483 KB) dan file kromosom 14 (152.755 KB).

#### 4.4.1 Skenario *Stand Alone*

Pada skenario pertama, penulis ingin melakukan eksperimen pencarian *pattern* pada *string* secara *stand alone* terhadap tiga file dengan *pattern* ‘CCG’, ‘CAG’ dan ‘TTAGGG’ seperti yang dapat terlihat pada Tabel 4.3.

**Tabel 4. 3 Skenario Eksperimen *Stand Alone***

No	<i>Pattern</i>	File
1	CCG	Homo_sapiens.GRCh38.88.chromosome.1.dat
2	CCG	Homo_sapiens.GRCh38.88.chromosome.4.dat
3	CCG	Homo_sapiens.GRCh38.88.chromosome.14.dat
4	CAG	Homo_sapiens.GRCh38.88.chromosome.1.dat
5	CAG	Homo_sapiens.GRCh38.88.chromosome.4.dat
6	CAG	Homo_sapiens.GRCh38.88.chromosome.14.dat
7	TTAGGG	Homo_sapiens.GRCh38.88.chromosome.1.dat
8	TTAGGG	Homo_sapiens.GRCh38.88.chromosome.4.dat
9	TTAGGG	Homo_sapiens.GRCh38.88.chromosome.14.dat

Tabel 4.3 menunjukkan eksperimen akan melakukan pencarian *pattern* ‘CCG’, ‘CAG’ dan ‘TTAGGG’ pada file kromosom 1, file kromosom 4, dan file kromosom 14 secara *stand alone*.

#### 4.4.2 Skenario *Parallel Computing*

Pada skenario kedua, penulis ingin melakukan 108 kali percobaan berkenaan dengan pencarian tiga *pattern* pada tiga file dengan empat nilai *iterator* berbeda yang dikerjakan oleh tiga kondisi jumlah *cores*: 2 *cores*, 4 *cores* dan 8 *cores* seperti yang terlihat pada Tabel 4.4.

Total penulis melakukan 114 kali eksperimen yang terbagi menjadi dua skenario (*stand alone* dan *parallel computing*) yang mana hasil dari kedua eksperimen tersebut akan menjadi bahan pembahasan tentang akurasi dan perbandingan kecepatan akurasi berdasarkan variabel nilai *iterator* dan jumlah *cores* yang berbeda-beda.

**Tabel 4. 4 Skenario Eksperimen *Parallel Computing***

<b>No</b>	<b>Pattern</b>	<b>File</b>	<b>Iterator</b>	<b>Core</b>
1	CCG	Kromosom 1	50	2
2	CCG	Kromosom 1	50	4
3	CCG	Kromosom 1	50	8
4	CCG	Kromosom 1	100	2
5	CCG	Kromosom 1	100	4
6	CCG	Kromosom 1	100	8
7	CCG	Kromosom 1	500	2
8	CCG	Kromosom 1	500	4
9	CCG	Kromosom 1	500	8
10	CCG	Kromosom 1	2.500	2
11	CCG	Kromosom 1	2.500	4
12	CCG	Kromosom 1	2.500	8
13	CCG	Kromosom 4	50	2
14	CCG	Kromosom 4	50	4
15	CCG	Kromosom 4	50	8
16	CCG	Kromosom 4	100	2
17	CCG	Kromosom 4	100	4
18	CCG	Kromosom 4	100	8
19	CCG	Kromosom 4	500	2
20	CCG	Kromosom 4	500	4
21	CCG	Kromosom 4	500	8
22	CCG	Kromosom 4	2.500	2
23	CCG	Kromosom 4	2.500	4
24	CCG	Kromosom 4	2.500	8
25	CCG	Kromosom 14	50	2
26	CCG	Kromosom 14	50	4
27	CCG	Kromosom 14	50	8
28	CCG	Kromosom 14	100	2
29	CCG	Kromosom 14	100	4
30	CCG	Kromosom 14	100	8
...	...	...	...	...
101	TTAGGG	Kromosom 14	100	4
102	TTAGGG	Kromosom 14	100	8
103	TTAGGG	Kromosom 14	500	2
104	TTAGGG	Kromosom 14	500	4
105	TTAGGG	Kromosom 14	500	8
106	TTAGGG	Kromosom 14	2.500	2
107	TTAGGG	Kromosom 14	2.500	4
108	TTAGGG	Kromosom 14	2.500	8

Pada Tabel 4.4 dapat dilihat *pattern* yang dicari pada file yang terletak di kolom ‘file’ dengan *iterator* adalah jumlah pemecahan *string* dan *core* adalah berapa banyak *core* yang digunakan dalam satu kali komputasi. Untuk isi Tabel 4.4 yang lengkap dapat dilihat pada Lampiran 2.

#### 4.5 Hasil Eksperimen

Setelah melakukan dua skenario eksperimen, penulis mendapatkan hasil yang akan dipaparkan pada subsubbab berikut.

##### 4.5.1 Hasil Eksperimen *Stand Alone*

Hasil dari eksperimen *stand alone* dapat dilihat pada Tabel 4.5. Kolom ‘Total Pattern’ menunjukkan jumlah *pattern* yang terdapat pada file. Sedangkan ‘Longest Pattern Repeats’ menunjukkan jumlah perulangan terpanjang dari *pattern* yang ada pada file diikuti dengan ‘Index Pattern Repeats’ adalah *index* dari *pattern* yang dimaksud pada ‘Longest Pattern Repeats’. Kolom ‘waktu’ menunjukkan jumlah waktu untuk menyelesaikan tiap komputasi dalam satuan detik.

**Tabel 4. 5 Hasil Eksperimen *Stand Alone***

No	Pattern	File	Total Pattern	Longest Pattern Repeats	Index Pattern Repeats	Waktu (dtk)
1	CCG	Kromosom 1	669.612	12	10694585 10694588 10694591 10694594 10694597 10694600 10694603 10694606 10694609 10694612 10694615 10694618	1821,48
2	CCG	Kromosom 4	386.822	10	576233 576236 576239 576242 576245 576248 576251 576254 576257 576260	667,84
3	CCG	Kromosom 14	243.060	9	101761689 101761692 101761695 101761698 101761701 101761704 101761707 101761710 101761713	340,31
4	CAG	Kromosom 1	4.852.390	12	209432294 209432297 209432300 209432303 209432306 209432309 209432312 209432315 209432318 209432321 209432324 209432327	81.439,60
5	CAG	Kromosom 4	3.456.386	19	3074877 3074880 3074883 3074886 3074889 3074892 3074895 3074898 3074901 3074904 3074907 3074910 3074913 3074916 3074919 3074922 3074925 3074928 3074931	42.836,94
6	CAG	Kromosom 14	1847242	11	38441512 38441515 38441518 38441521 38441524 38441527 38441530 38441533 38441536 38441539 38441542	11.931,92
7	TTAGGG	Kromosom 1	43.719	10	248946280 248946286 248946292 248946298 248946304 248946310 248946316 248946322 248946328 248946334	344,02
8	TTAGGG	Kromosom 4	35.503	11	190122944 190122950 190122956 190122962 190122968 190122974 190122980 190122986 190122992 190122998 190123004	302,90
9	TTAGGG	Kromosom 14	17.018	4	18572932 18572938 18572944 18572950	207,00

#### 4.5.2 Hasil Eksperimen *Parallel Computing*

Hasil dari eksperimen *parallel computing* dapat dilihat pada Tabel 4.6 (untuk lengkapnya dapat melihat Lampiran 3). Kolom *iterator* menunjukkan nilai *iterator* atau jumlah pemotongan yang dilakukan pada *string* yang terdapat pada file. Sedangkan kolom *core* menunjukkan berapa *core* yang digunakan pada saat komputasi itu dilakukan. Kolom '*Total Pattern*' menunjukkan jumlah *pattern* yang terdapat pada file. Sedangkan '*Longest Pattern Repeats*' menunjukkan jumlah perulangan terpanjang dari *pattern* yang ada pada file diikuti dengan '*Index Pattern Repeats*' adalah *index* dari *pattern* yang dimaksud pada '*Longest Pattern Repeats*'. Kolom 'waktu' menunjukkan jumlah waktu untuk menyelesaikan tiap komputasi dalam satuan detik.

**Tabel 4. 6 Hasil Eksperimen *Parallel Computing***

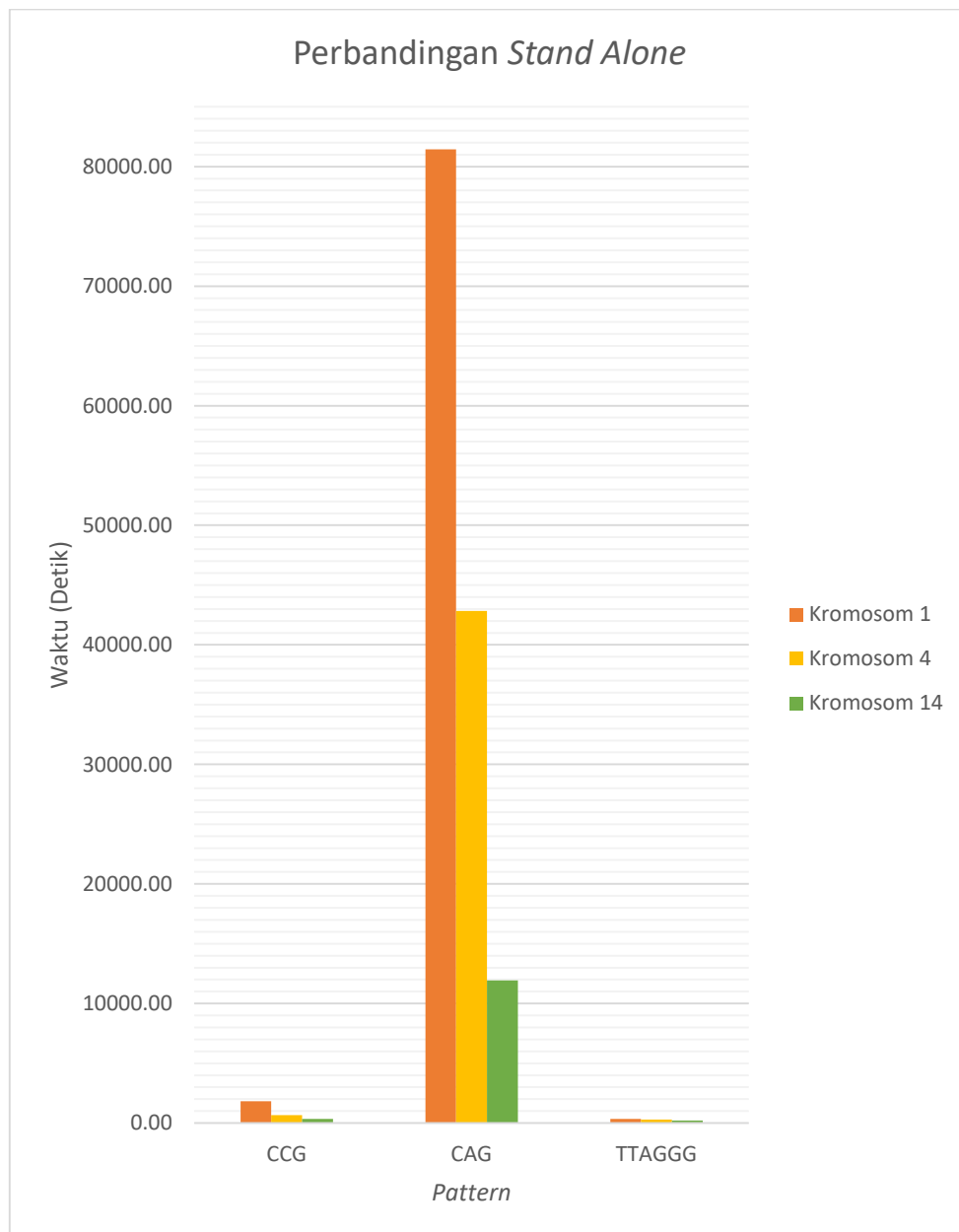
No	Pattern	File	Iterator	Core	Total Pattern	Longest Pattern Repeats	Index Pattern Repeats	Waktu
1	CCG	Kromosom 1	50	2	669.612	12	10694585 10694588 10694591 10694594 10694597 10694600 10694603 10694606 10694609 10694612 10694615 10694618	130,10
2	CCG	Kromosom 1	50	4	669.612	12	10694585 10694588 10694591 10694594 10694597 10694600 10694603 10694606 10694609 10694612 10694615 10694618	48,42
3	CCG	Kromosom 1	50	8	669.612	12	10694585 10694588 10694591 10694594 10694597 10694600 10694603 10694606 10694609 10694612 10694615 10694618	36,01
4	CCG	Kromosom 1	100	2	669.612	12	10694585 10694588 10694591 10694594 10694597 10694600 10694603 10694606 10694609 10694612 10694615 10694618	161,11
5	CCG	Kromosom 1	100	4	669.612	12	10694585 10694588 10694591 10694594 10694597 10694600 10694603 10694606 10694609 10694612 10694615 10694618	43,48
6	CCG	Kromosom 1	100	8	669.612	12	10694585 10694588 10694591 10694594 10694597 10694600 10694603 10694606 10694609 10694612 10694615 10694618	32,39
...	...	...	...	...	...	...	...	...
105	TTAGGG	Kromosom 14	500	8	17.018	4	18572932 18572938 18572944 18572958	12,57
106	TTAGGG	Kromosom 14	2.500	2	17.018	4	18572932 18572938 18572944 18572959	105,49
107	TTAGGG	Kromosom 14	2.500	4	17.018	4	18572932 18572938 18572944 18572960	26,78
108	TTAGGG	Kromosom 14	2.500	8	17.018	4	18572932 18572938 18572944 18572961	17,26

## 4.6 Pembahasan

Dari hasil eksperimen yang didapatkan, penulis dapat melakukan beberapa pembahasan yang akan dipaparkan pada subsubbab berikut.

### 4.6.1 Perbandingan Kecepatan di *Stand Alone*

Hasil eksperimen *stand alone* dapat memperlihatkan perbandingan waktu komputasi yang dibedakan oleh panjang *string* atau ukuran file seperti yang terlihat pada Gambar 4.8.



Gambar 4. 8 Perbandingan kecepatan pada skenario *stand alone*



Dari Gambar 4.8 dapat dilihat jika semakin kecil ukuran file atau panjang *string* maka akan semakin cepat pula proses komputasi yang dilakukan. Terlihat juga perbedaan kecepatan komputasi dari *pattern* yang dicari. Untuk *pattern* ‘CCG’ yang memiliki *prefix* ‘0 1 0’ membutuhkan waktu sebanyak 1.821,48 detik atau sekitar 30 menit untuk menemukan 669.612 *pattern* ‘CCG’ dalam file kromosom 1. Sedangkan untuk *pattern* ‘TTAGGG’ yang memiliki *prefix* ‘0 1 0 0 0 0’ membutuhkan waktu 344,02 detik atau sekitar lima menit untuk menemukan 43.719 *pattern* ‘TTAGGG’ pada file yang sama. Sedangkan hasil yang sangat signifikan ketika melakukan pencarian ‘CAG’ yang memiliki *prefix* ‘0 0 0’ yang membutuhkan waktu 81.439,6 detik atau sekitar 22,6 jam untuk menemukan 4.852.390 *pattern* ‘CAG’ dalam file kromosom 1.

Terdapat dua kemungkinan yang membedakan antara pencarian *pattern* pada file yang sama terhadap waktu komputasi. Pertama adalah pengaruh *prefix* karena pergeseran pencarian dalam algoritma Knuth-Morris-Pratt bergantung pada *prefix* atau karena pencarian *pattern* ‘CAG’ lebih banyak ditemukan (sekitar empat puluh kali lipat jumlah *pattern* ‘CCG’ dan 230 kali lipat *pattern* ‘TTAGGG’ pada file kromosom 1) yang akan menyebabkan komputasi lebih lama karena adanya aksi penyimpanan *index pattern* yang ditemukan.

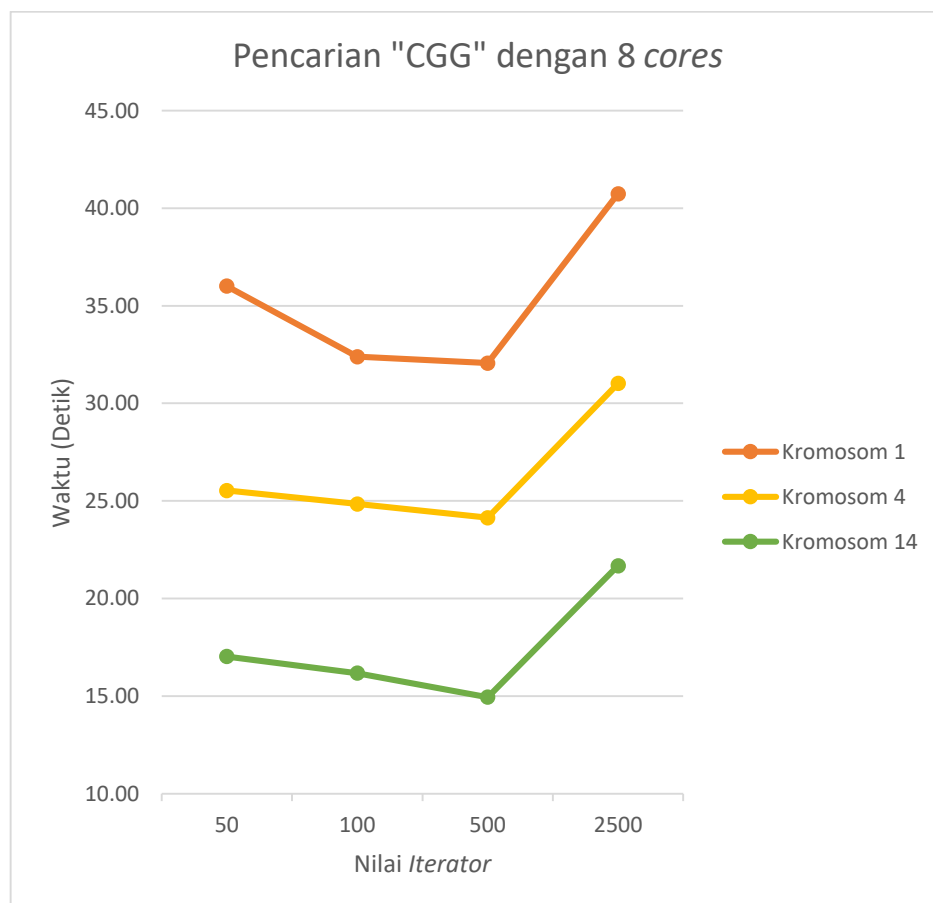
Namun penulis menyimpulkan banyaknya *pattern* pada file lebih dominan menjadi penyebab terjadinya perbedaan kecepatan. Hal ini dapat dilihat dari perbedaan jumlah *pattern* ‘CAG’ pada ketiga file tersebut memiliki jarak sekitar satu juta-dua juta *pattern* sedangkan perbedaan jumlah *pattern* ‘CCG’ pada ketiga file tersebut memiliki jarak sekitar 200.000-300.000 *pattern*. Selain itu *pattern* ‘TTAGGG’ pada ketiga file tersebut memiliki perbedaan jumlah *pattern* yang ditemukan hanya sekitar 1.000-2.000 *pattern*. Hal ini diperkuat dengan hasil pada pencarian *pattern* ‘TTAGGG’ yang tidak terlalu signifikan dalam waktu penyelesaian komputasi.

#### **4.6.2 Perbandingan Kecepatan Berdasarkan Nilai *Iterator***

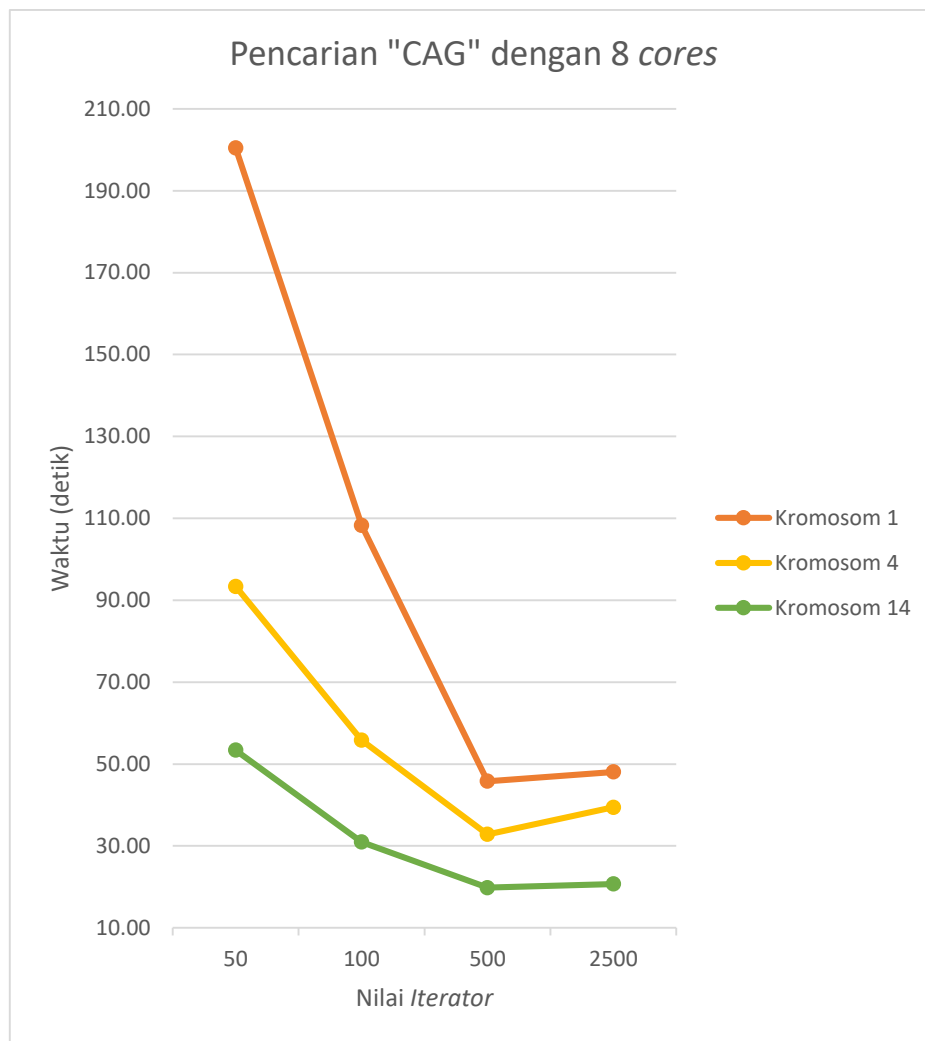
Hasil eksperimen *parallel computing* dapat dipecah menjadi beberapa sudut pandang. Salah satunya adalah melihat kecepatan yang disebabkan oleh perbedaan nilai *iterator* yang digunakan untuk dilakukannya komputasi. Dengan

jumlah *core* yang sama (8 *cores*) maka perbandingan berdasarkan nilai *iterator* dapat dianalisa.

Gambar 4.9 memperlihatkan grafik waktu komputasi pencarian *pattern* ‘CCG’ dengan empat kondisi nilai *iterator*: 50, 100, 500 dan 2.500. Terlihat untuk pencarian pada file kromosom 1 terjadi pengerjaan komputasi paling cepat pada saat *iterator* 500 dengan waktu 32,06 detik. Sedangkan waktu komputasi tertinggi adalah ketika nilai *iterator* adalah 2.500 selama 40,74 detik. Pencarian pada file kromosom 4 memiliki waktu komputasi paling rendah 24,14 detik pada *iterator* 500. Sedangkan waktu komputasi terendah untuk pencarian pada file kromosom 14 membutuhkan waktu 14,94 detik pada nilai *iterator* adalah 500. Kesimpulan untuk pencarian *pattern* ‘CCG’ adalah memiliki waktu komputasi paling rendah saat *iterator* adalah 500, sedangkan waktu komputasi paling tinggi adalah saat *iterator* adalah 2.500.

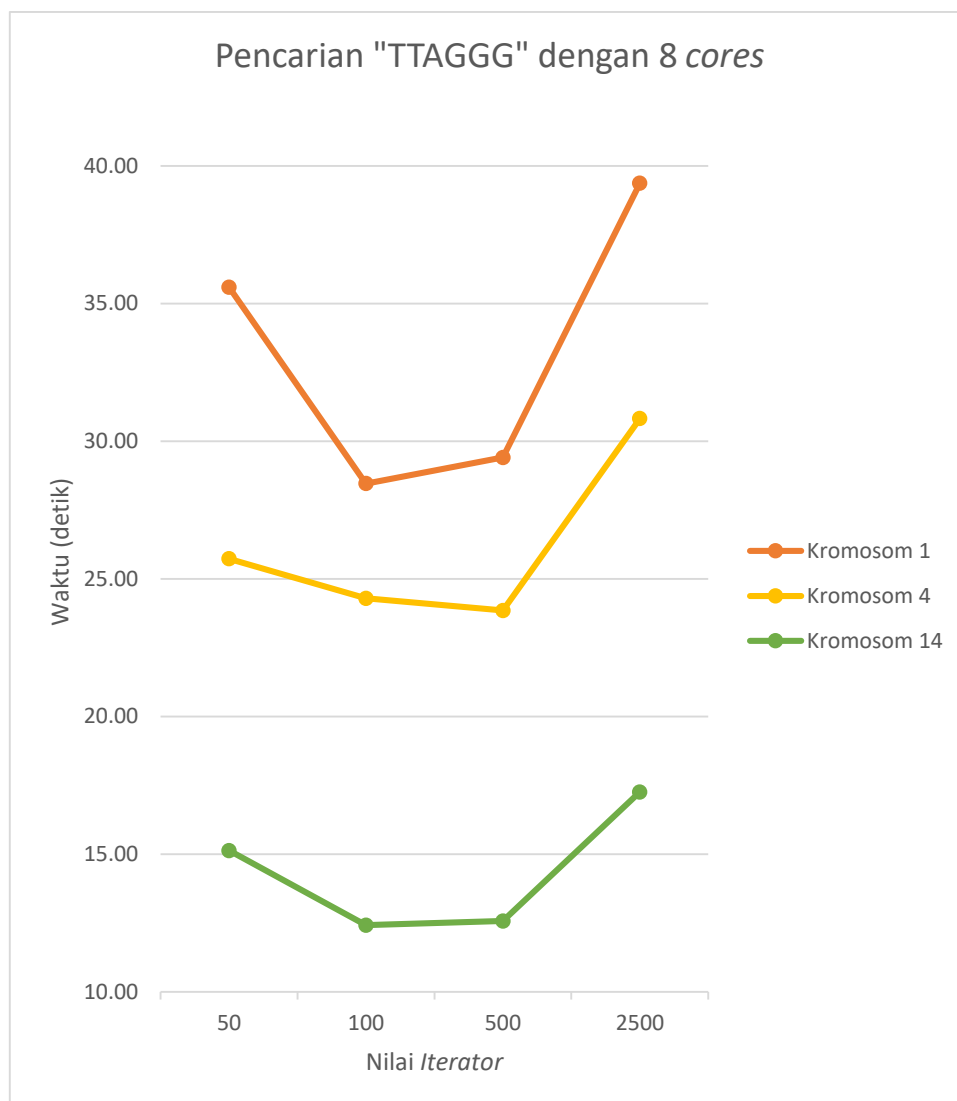


**Gambar 4. 9 Pencarian *pattern* ‘CCG’ dengan 8 *cores* pada file kromosom 1, kromosom 4 dan kromosom 14**



**Gambar 4. 10 Pencarian *pattern* ‘CAG’ dengan 8 *cores* pada file kromosom 1, kromosom 4 dan kromosom 14**

Gambar 4.10 memperlihatkan grafik waktu komputasi pencarian *pattern* ‘CAG’ dengan empat kondisi nilai *iterator*: 50, 100, 500 dan 2.500. Terlihat untuk pencarian pada file kromosom 1 terjadi pengerjaan komputasi paling cepat pada saat *iterator* 500 dengan waktu 45,80 detik. Sedangkan waktu komputasi tertinggi adalah ketika nilai *iterator* adalah 50 selama 200,46 detik. Pencarian pada file kromosom 4 memiliki waktu komputasi paling rendah 32,81 detik pada *iterator* 500. Sedangkan waktu komputasi terendah untuk pencarian pada file kromosom 14 membutuhkan waktu 19,81 detik pada nilai *iterator* adalah 500. Kesimpulan untuk pencarian *pattern* ‘CAG’ adalah masing-masing memiliki waktu komputasi paling rendah saat *iterator* adalah 500, sedangkan waktu komputasi paling tinggi adalah saat *iterator* adalah 50.



**Gambar 4. 11 Pencarian *pattern* ‘TTAGGG’ dengan 8 *cores* pada file kromosom 1, kromosom 4 dan kromosom 14**

Gambar 4.11 memperlihatkan grafik waktu komputasi pencarian *pattern* ‘TTAGGG’ dengan empat kondisi nilai *iterator*: 50, 100, 500 dan 2.500. Terlihat untuk pencarian pada file kromosom 1 terjadi pengerjaan komputasi paling cepat pada saat *iterator* 100 dengan waktu 28,46 detik. Sedangkan waktu komputasi tertinggi adalah ketika nilai *iterator* adalah 2.500 selama 39,37 detik. Pencarian pada file kromosom 4 memiliki waktu komputasi paling rendah 23,85 detik pada *iterator* 500. Sedangkan waktu komputasi terendah untuk pencarian pada file kromosom 14 membutuhkan waktu 12,42 detik pada nilai *iterator* adalah 100. Kesimpulan untuk pencarian *pattern* ‘TTAGGG’ adalah memiliki waktu komputasi paling rendah saat *iterator* adalah 100 untuk kromosom 1 dan kromosom 14 dan

500 untuk kromosom 4, sedangkan waktu komputasi paling tinggi adalah saat *iterator* adalah 2.500.

Melihat dari grafik yang ditunjukkan pada Gambar 4.9, Gambar 4.10 dan Gambar 4.11 dapat disimpulkan bahwa semakin besarnya nilai *iterator* tidak akan selalu sebanding dengan waktu komputasi yang lebih tinggi atau lebih rendah. Dapat dilihat bahwa waktu komputasi turun namun kembali naik pada *iterator* 2.500 ke atas. Setiap kondisi memiliki nilai *iterator* optimum berdasarkan panjang *string* dan *prefix*. Semakin tinggi nilai *iterator* maka semakin banyak jumlah pemotongan *string* yang mengakibatkan setiap *core* bekerja dengan *string* yang lebih pendek. Namun hal itu juga berbanding dengan jalur komunikasi yang semakin banyak untuk mengumpulkan hasil dari setiap komputasi yang dilakukan oleh tiap *core* secara paralel.

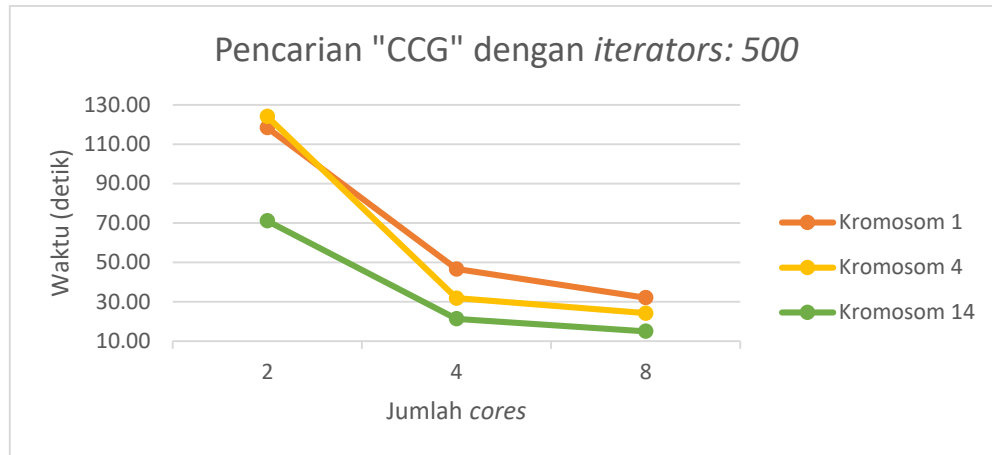
Pada kasus dan eksperimen penelitian ini, nilai *iterator* optimal adalah 500 yang menghasilkan waktu komputasi paling rendah pada tujuh dari sembilan hasil eksperimen.

#### **4.6.3 Perbandingan Kecepatan Berdasarkan Jumlah Core**

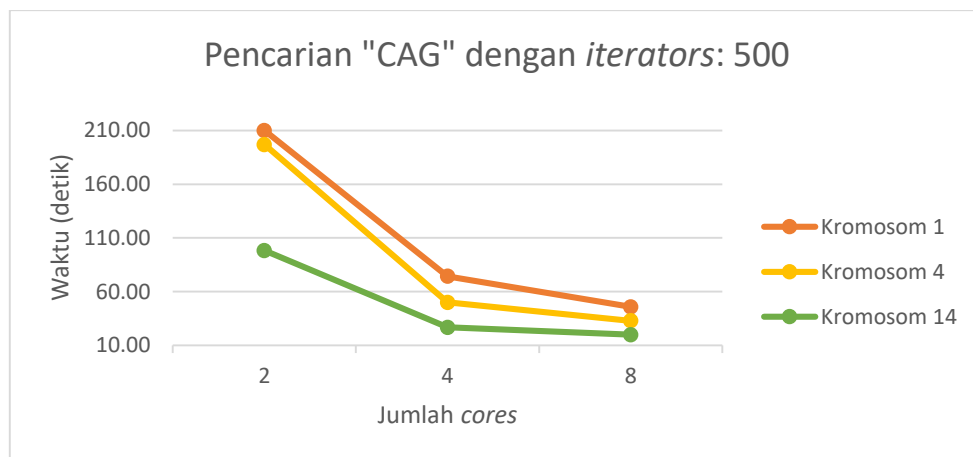
Hasil dari eksperimen *parallel computing* selanjutnya adalah melihat kecepatan yang disebabkan oleh perbedaan jumlah *core* yang digunakan untuk dilakukannya komputasi. Dengan jumlah *iterator* yang sama (500) maka perbandingan berdasarkan jumlah *core* dapat dianalisa.

Gambar 4.12 memperlihatkan grafik yang menunjukkan perbandingan waktu komputasi pencarian *pattern* ‘CCG’. Pada grafik tersebut memperlihatkan penurunan waktu komputasi dari 2 *cores* ke 4 *cores* lebih signifikan dari pada penurunan waktu komputasi dari 4 *cores* ke 8 *cores*. Begitu pula yang terjadi pada pencarian *pattern* ‘CAG’ yang dapat dilihat pada Gambar 4.13 dan *pattern* ‘TTAGGG’ yang dapat dilihat pada Gambar 4.14.

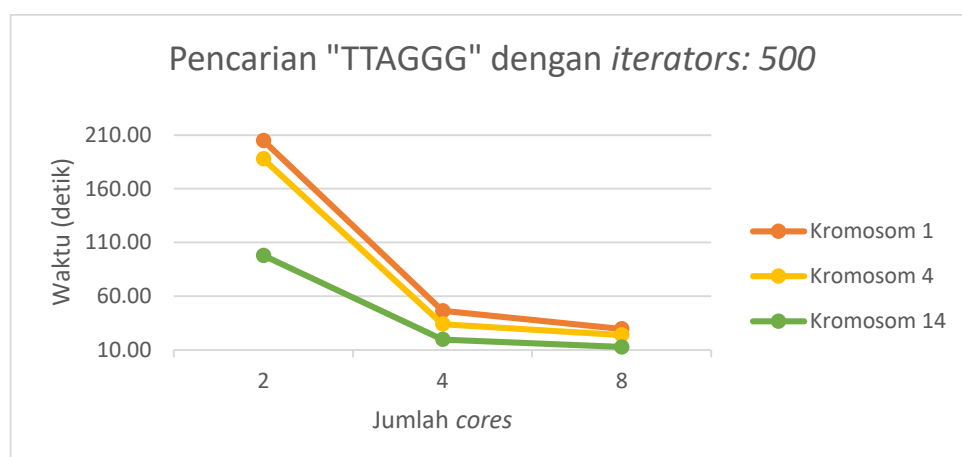
Penurunan percepatan waktu komputasi terhadap jumlah *core* terjadi karena terdapat pencarian *pattern* berulang yang dilakukan tidak secara paralel. Hal ini dapat menyebabkan penambahan jumlah *core* tidak akan mempengaruhi kecepatan komputasi karena telah berada pada kecepatan maksimum.



**Gambar 4. 12** Pencarian *pattern* ‘CCG’ dengan nilai *iterator 500* pada file kromosom 1, kromosom 4 dan kromsom 14



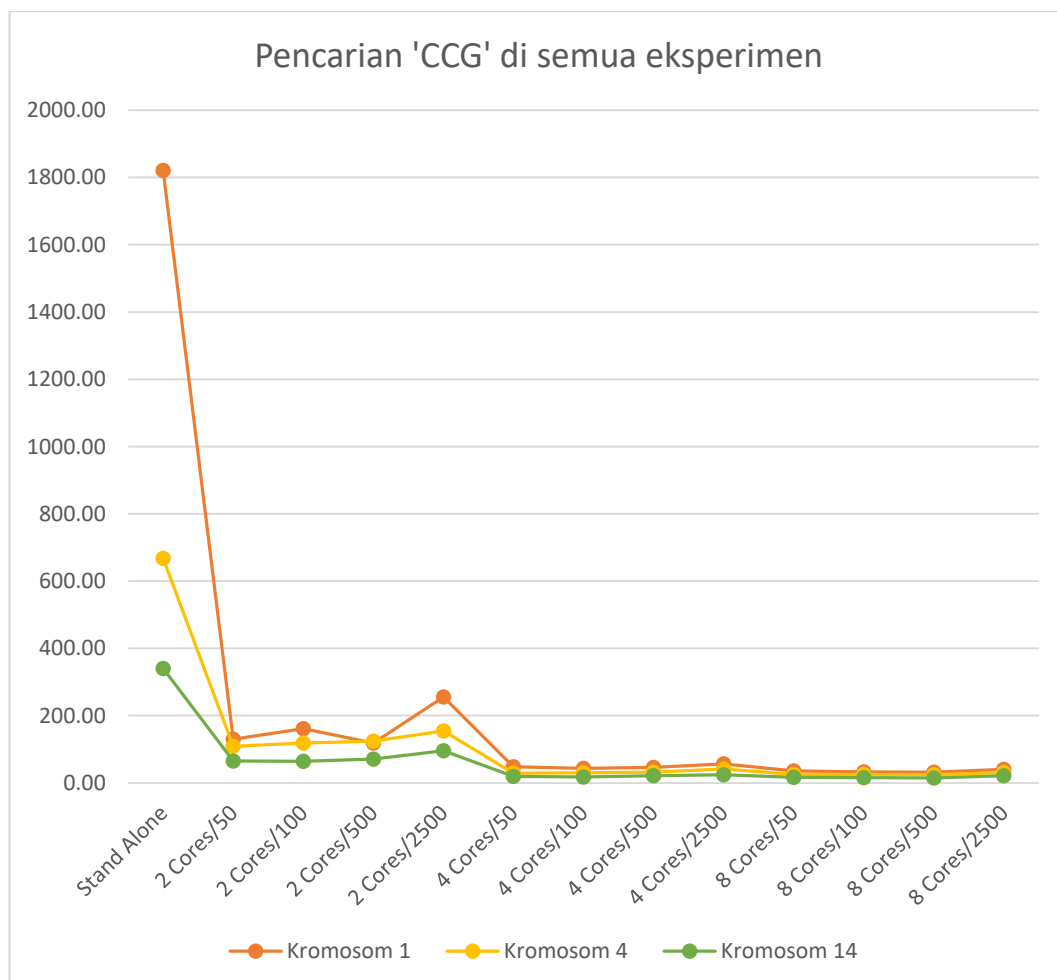
**Gambar 4. 13** Pencarian *pattern* ‘CAG’ dengan nilai *iterator 500* pada file kromosom 1, kromosom 4 dan kromsom 14



**Gambar 4. 14** Pencarian *pattern* ‘TTAGGG’ dengan nilai *iterator 500* pada file kromosom 1, kromosom 4 dan kromsom 14

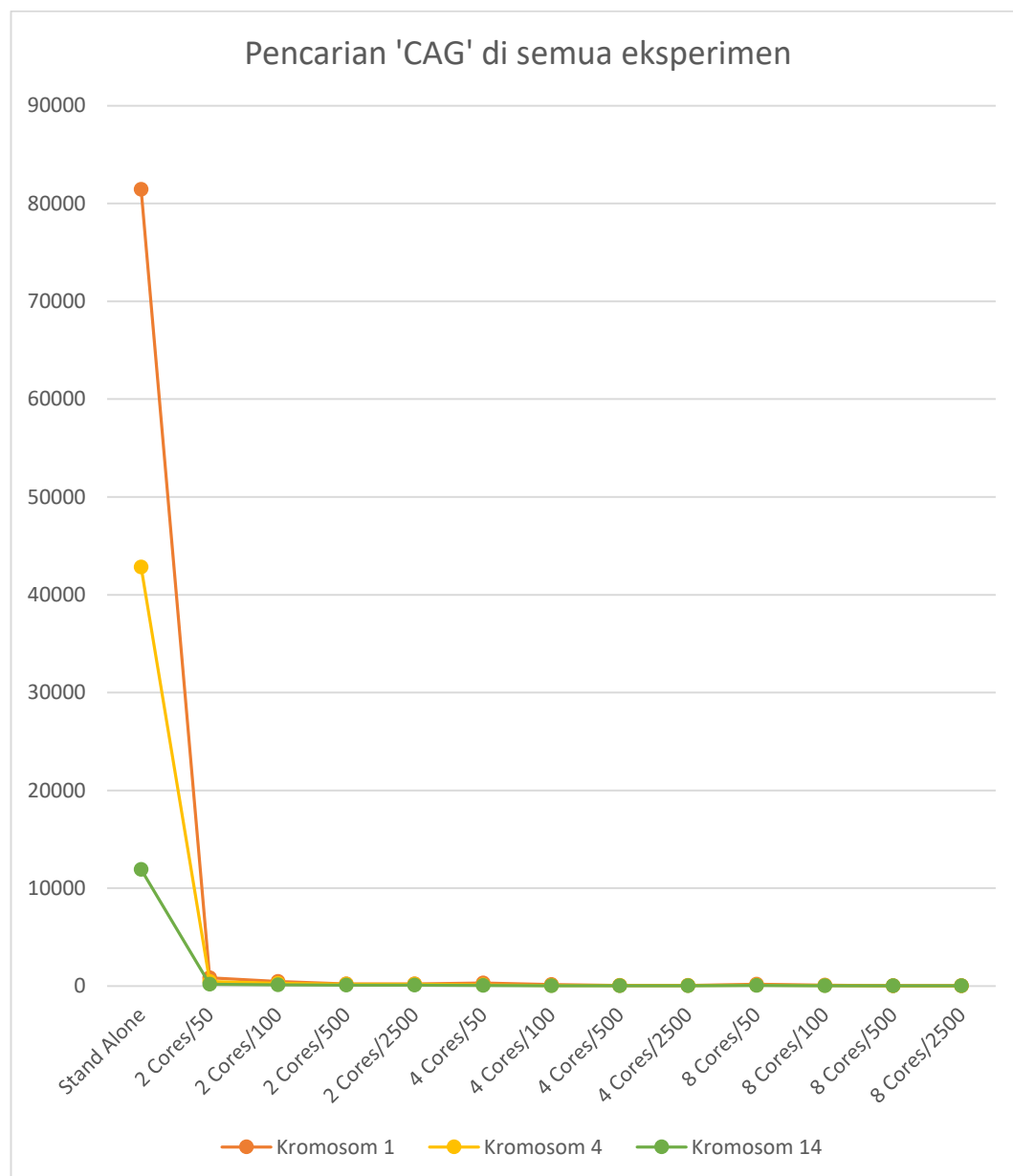
#### 4.6.4 Perbandingan Kecepatan *Stand Alone* dan *Parallel Computing*

Pada Gambar 4.15 diperlihatkan grafik perbandingan waktu pengerjaan antara komputasi yang dikerjakan secara *stand alone* dan *parallel computing* pada pencarian ‘CCG’ pada file kromosom 1, kromosom 4 dan kromosom 14. Terlihat pada grafik tersebut bahwa waktu komputasi dengan *stand alone* pada file kromosom 1 selama 1.821,48 detik memakan waktu lebih dari sepuluh kali lipat waktu komputasi secara paralel menggunakan 2 *core* rata-rata selama 166,21 detik. Sedangkan waktu komputasi *stand alone* tersebut memakan lebih dari 37 kali lipat waktu komputasi secara paralel menggunakan 4 *cores* dengan rata-rata selama 48,78 detik dan memakan lebih dari 50 kali lipat (35,30 detik) saat menggunakan 8 *cores*.



**Gambar 4. 15 Pencarian ‘CCG’ di semua eksperimen pada file kromosom 1, kromosom 4 dan kromosom 14**

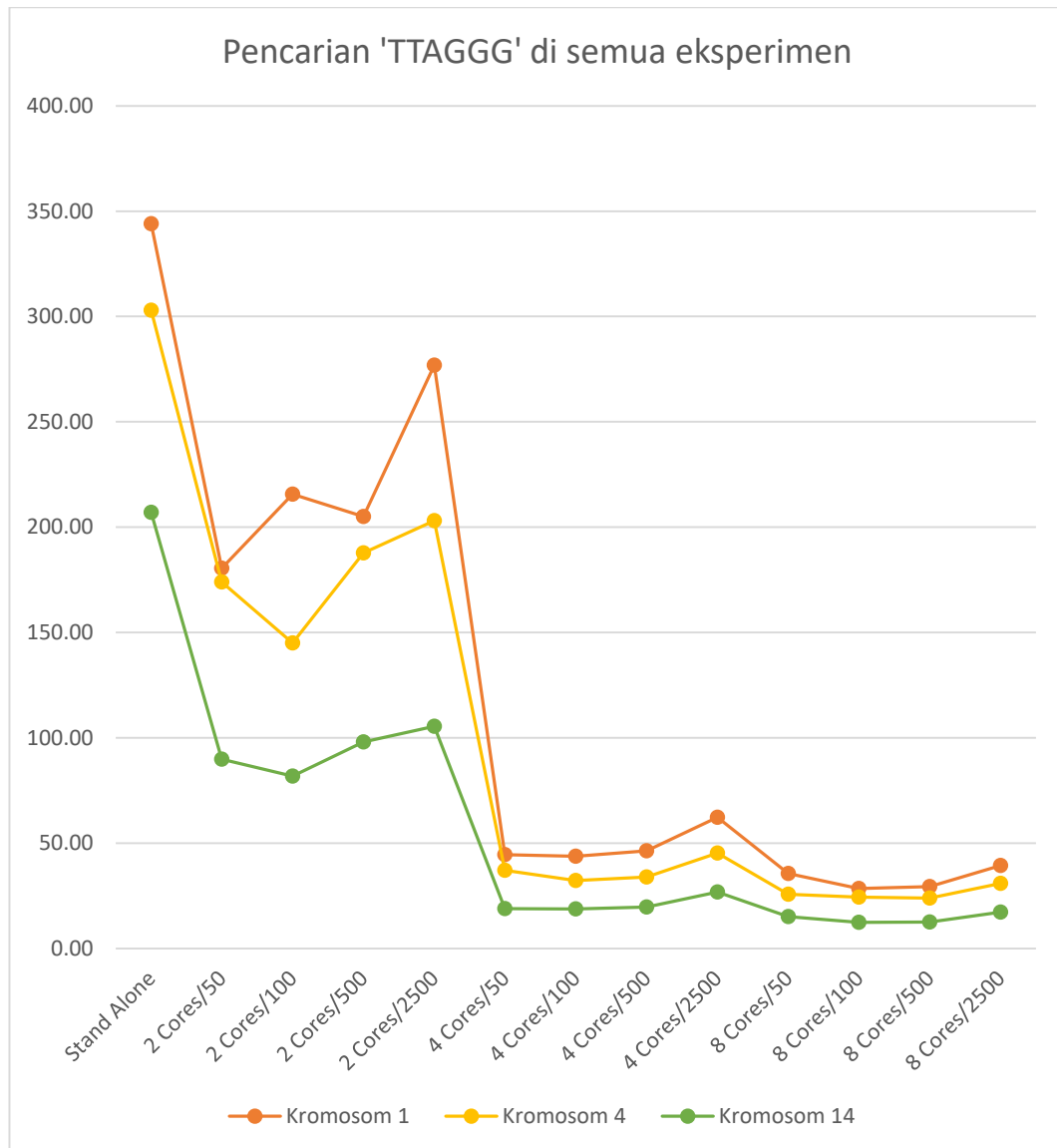
Berikutnya untuk pencarian ‘CAG’ yang dapat dilihat pada Gambar 4.16, terlihat perbandingan hasil dari eksperimen *stand alone* dengan semua hasil eksperimen *parallel computing* yang sangat signifikan. Pada file kromosom 1, pencarian menggunakan *stand alone* memakan waktu lebih dari 184 kali rata-rata pencarian dengan 2 *cores*, lebih dari 528 kali rata-rata pencarian dengan 4 *cores* dan lebih dari 809 kali lipat rata-rata pencarian menggunakan 8 *cores*. Selisih waktu komputasi *stand alone* dengan waktu komputasi terendah pada *parallel computing* adalah sebesar 81.393,80 detik atau sekitar 22,6 jam.



**Gambar 4. 16 Pencarian ‘CAG’ di semua eksperimen pada file kromosom 1, kromosom 4 dan kromosom 14**



Sedangkan untuk pencarian ‘TTAGGG’ dapat dilihat pada Gambar 4.17 di mana hasil eksperimen dari *stand alone* dan *parallel computing* tidak terlalu signifikan seperti pencarian dua *pattern* sebelumnya. Pencarian dengan *stand alone* hanya membutuhkan waktu sepuluh kali dari rata-rata pencarian menggunakan 8 *cores*. Selisih antara *stand alone* dan waktu komputasi terendah di *parallel computing* hanyalah sebesar 315,56 detik.



**Gambar 4. 17 Pencarian ‘TTAGGG’ di semua eksperimen pada file kromosom 1, kromosom 4 dan kromosom 14**

Penelitian sebelumnya (Cheng, Cheung dan Yiu, 2003) menggunakan *suffix* dari *patten*, tidak seperti pada penelitian ini yang menggunakan *prefix* dalam algoritma Knuth-Morris-Pratt itu sendiri. Penelitian tersebut menggunakan data

dari DNA Data Bank Japan (DDBJ) secara acak. Hasil dari penelitian tersebut dengan 2 komputer yang dijalankan secara paralel perlu menghabiskan waktu lebih dari 100 detik hanya untuk memproses sekuens DNA yang memiliki 1.000 pasang basa. Sedangkan dalam penelitian ini yang menggunakan 2 cores hanya membutuhkan waktu 86,99 detik untuk melakukan proses pencarian pada sekuens yang memiliki 107.043.718 pasang basa. Perbandingan yang sangat signifikan ini tentu tidak terlepas dari perbedaan spesifikasi sumber daya yang digunakan pada 14 tahun lalu. Namun ini membuktikan bahwa penelitian ini dapat memberikan perkembangan yang cukup baik di masa sekarang.

Selain itu penelitian sebelumnya (Al-Dabbagh, Barnouti, Naser, dan Ali, 2016) melakukan pencarian *pattern* pada sekuens DNA dan teks bahasa Inggris dengan menggunakan OpenMPI. Algoritma yang digunakan pada penelitian tersebut adalah algoritma pencarian *string* Boyer Moore. Dari penelitian tersebut mengungkapkan bahwa eksperimen dengan menggunakan data DNA memberikan hasil yang lebih stabil dibandingkan dengan teks bahasa Inggris.

Untuk rasio waktu komputasi dari *stand alone* dan *parallel computing* dengan 2, 4 dan 8 *cores* menunjukkan bahwa jumlah *core* tidak membuat komputasi lebih cepat secara linear. Hal itu berarti bahwa waktu komputasi 4 *cores* tidak konstan lebih cepat sekalian kali dari 2 *cores*. Namun jika dilihat dari rasio *stand alone* dan *parallel computing* yang dapat mencapai perbandingan waktu komputasi 1:809 menunjukkan kemajuan yang sangat signifikan dari penelitian sebelumnya yang hanya memiliki perbandingan sekitar 1:3 antara waktu komputasi *stand alone* dan *parallel computing*.

#### 4.6.5 Perbandingan Akurasi

Perbandingan akurasi pada penelitian ini berpatok pada hasil *parallel computing* yang dibandingkan dengan hasil *stand alone* agar memastikan hasil dari *parallel computing* cocok dengan *stand alone* karena ada kemungkinan terjadi perbedaan karena pemecahan. Hasil eksperimen *parallel computing* dan *stand alone* pada penelitian ini menunjukkan hasil yang sama 100% yang dapat dilihat pada Tabel 4.5 dan Tabel 4.6 (tabel yang lengkap tersedia dilampiran).

## BAB V

### KESIMPULAN DAN SARAN

#### 5.1 Kesimpulan

Setelah melakukan penelitian mengenai program deteksi *genomic repeats* dengan implementasi algoritma Knuth-Morris-Pratt pada R *package high-performance computing* pbdMPI, maka penulis mendapatkan beberapa kesimpulan yang berhubungan dengan tujuan penelitian. Berikut kesimpulan yang dapat penulis jabarkan.

1. Berhasil membuat model *parallel computing* untuk pencarian *pattern* pada *string* menggunakan algoritma Knuth-Morris-Pratt. Model tersebut memiliki konsep pemotongan *string* agar tidak terjadi *miss* ketika *pattern* yang dicari tepat berada pada pemotongan *string*. Pada model tersebut juga dikenalkan variabel *adder* sebagai penambah nilai *index* dari setiap hasil keluaran komputasi secara paralel agar sesuai dengan *index* pada *string* yang utuh.
2. Berhasil mengimplementasi program dengan model yang telah dirancang sebelumnya untuk meningkatkan kecepatan dan menjaga akurasi hasil yang baik. Program yang dibangun menggunakan bahasa pemrograman R dengan *package* pbdMPI untuk melakukan *parallel computing*. Program ini telah diuji untuk *error handling* dan akurasinya terhadap data yang lebih kecil.
3. Telah melakukan total 117 eksperimen yang terbagi ke dalam skenario *stand alone* sebanyak 9 eksperimen dan skenario *parallel computing* sebanyak 108 eksperimen. Eksperimen tersebut dilakukan pada satu komputer dengan spesifikasi yang sama. Eksperimen dengan waktu terlalu lama adalah pencarian ‘CAG’ pada file kromosom 1 secara *stand alone* yang menghabiskan waktu komputasi sebanyak 81.439,6 detik atau sekitar 22,6 jam.
4. Melakukan analisa hasil eksperimen dengan kesimpulan bahwa pemangkasan waktu pemrosesan tidak beriringan dengan jumlah *core* yang digunakan dalam hal rasio. Contoh dalam penggunaan 4 *cores* tidak berarti memiliki waktu pemrosesan 50% dari waktu yang dibutuhkan jika menggunakan 2 *cores*. Hal ini disebabkan dengan adanya proses pembacaan file, praproses

pembersihan *string* dan juga pencarian *pattern* yang berulang berdekatan. Selain itu nilai *iterator* yang optimal bergantung pada panjang karakter dari *string* mengingat akan terjadinya komunikasi yang banyak pula jika *iterator* diatur dengan nilai yang sangat besar. Selain sumber daya, panjang karakter *string*, *prefix pattern* dan nilai *iterator* ternyata jumlah *pattern* yang ditemukan pada *string* juga mempengaruhi waktu komputasi karena adanya proses penyimpanan *index* dan juga pencarian *pattern* berulang yang akan semakin lama jika *pattern* yang ditemukan pada *string* semakin banyak.

## 5.2 Saran

Dalam pelaksanaan penelitian, penulis menyadari bahwa masih banyak kekurangan yang dilakukan oleh penulis dalam penelitian ini. Oleh karena itu, penulis menyampaikan beberapa saran yang dapat dilakukan di kemudian hari. Penelitian selanjutnya dapat menghasilkan sebuah program yang jauh lebih baik dalam hal kecepatan. Berikut beberapa saran yang dapat penulis anjurkan.

1. Dalam input data seperti nama file, *pattern* dan nilai *iterator* dapat dimasukkan pengguna pada *terminal/prompt* karena kemungkinan akan adanya kesulitan pengguna dalam memasukkannya pada kode program.
2. Meneliti penggunaan sumber daya untuk eksekusi program agar dapat dilakukan pada sumber daya komputer seminimal mungkin.
3. Sebaiknya dapat membuat sebuah model untuk pembacaan file secara paralel juga mengingat tahap ini dilakukan secara *stand alone* pada penelitian ini.
4. Membuat beberapa fungsi praproses yang lebih beragam dan dinamis sehingga dapat digunakan pada data yang universal, tidak berpatok pada data dengan standar format NCBI/Ensembl.
5. Dapat membuat model atau formula untuk menentukan *nilai* *iterator* optimal.
6. Penulis berharap program ini dapat digunakan untuk menunjang perkembangan ilmu pengetahuan khususnya dalam bidang ilmu komputer dan biologi.

## DAFTAR PUSTAKA

- Albert, B. (2008). *Molecular Biology of The Cell*. New York: Garland Science.
- Al-Dabbagh, S. S., Barnouti, N. H., Naser, M. A., dan Ali, Z. G. (2016). Parallel Quick Search Algorithm for the Exact String Matching Problem Using OpenMP. *Journal of Computer and Communications*, 4, 1-11.
- Amir, Y., Awerbuch, B., Barak, A., dan R. Borgstrom, A. K. (2000). An Opportunity Cost Approach For Job Assignment In A Scalable Computing Cluster. *IEEE Transactions on Parallel and Distributed Systems*, 11(7), 760-768.
- Amir, Y., Awerbuch, B., Barak, A., Borgstrom, R., dan Keren, A. (2000). An Opportunity Cost Approach For Job Assignment In A Scalable Computing Cluster. *IEEE Transactions on Parallel and Distributed Systems*, 11(7), 760-768.
- Barth, G. (1981). An Alternative For The Implementation of The Knuth-Morris-Pratt Algorithm. *Information Processing Letters*, 13, 134-137.
- Calladine, C. R., Drew, H. R., Luisi, B. F., dan Travers, A. A. (2004). *Understanding DNA: The Molecule and How It Works (Third Edition)*. San Diego: Elsevier Academic Press.
- Campbell, N. A., dan Reece, J. B. (2008). *Biology: Eight Edition*. San Francisco: Pearson Benjamin Cummings.
- Chen, W.-C., Ostrouchov, G., Schmidt, D., Patel, P., dan Yu, H. (2012). pbdMPI: Programming with Big Data - Interface to MPI.
- Cheng, L.-L., Cheung, D. W., dan Yiu, S.-M. (2003). Approximate String Matching in DNA Sequences. *8th International Conference on Database Systems for Advanced Applications*. Kyoto, Japan: IEEE.
- Claude T. Ashley, J., dan Warre, S. T. (1995). Trinucleotide Repeat Expansion and Human Disease. *Annual Reviews*, 29, 703-728.
- dclone: Data Cloning in R. (2010). *The R Journal*, 2(2), 29-37.
- Dean, J., dan Ghemawat, S. (2010). MapReduce: A Flexible Data Processing Tool. *Communications of The ACM*, 53, 72-77.
- Edgar, R. C., dan Myers, E. W. (2005). PILER: Identification and Classification of Genomic Repeats. *Bioinformatics*, 21(1), 152-158.
- Foundation, N. S. (n.d.). *DNA Sequencing Process (Step 5)*. Retrieved from Nasional Science Foundation: [https://www.nsf.gov/news/mmg/mmg\\_disp.jsp?med\\_id=51711&from=](https://www.nsf.gov/news/mmg/mmg_disp.jsp?med_id=51711&from=)

- Franek, F., Jennings, C. G., dan Smyth, W. F. (2005). A Simple Fast Hybrid Pattern Matching Algorithm. *CPM*, 288-297.
- Houston, M. (2007, 10 4). *Folding@Home - GPGPU*. Retrieved from <http://graphics.stanford.edu/~mhouston/>
- Hussain, H., Malik, S. U., Hameed, A., Khan, S. U., Bickler, G., Min-Allah, N., . . . Li, H. (2013). A Survey On Resource Allocation In High Performance Distributed Computing Systems. *Parallel Computing*, 709–736.
- Irwin, D., Grit, L., dan Chas, J. (2004). Balancing Risk And Reward In A Market-based Task Service. *13th International Symposium on High Performance Distributed Computing (HPDC13)*, (pp. 160–169).
- Ishwaran, H., dan Kogalur, U. (2007). Random Survival Forests for R. *R News*, 7(2), 25-31.
- Iverson, dan Cheryl. (2007). *AMA Manual of Style*. Oxford, Oxfordshire: Oxford University Press.
- Kindhi, B. A., dan Sardjono, T. A. (2015). Pattern Matching Performance Comparison as Big Data Analysis Recommendations for Hepatitis C Virus (HCV) Sequence DNA. *2015 Third International Conference on Artificial Intelligence, Modelling and Simulation* (pp. 99-104). IEEE.
- Knaus, J. (2008). sfCluster/snowfall: Managing Parallel Execution of R Programs on a Compute Cluster. *useR!*
- Knuth, D. E., Morris, J. H., dan Pratt, V. R. (1977). Fast Pattern Matching In Strings. *SIAM Journal on Computing*, 6(2), 323-350.
- Kołodziej, J., Khan, S., Wang, L., Kisiel-Dorohinicki, M., Madani, S., Niewiadomska-Szynkiewicz, E., . . . Xu, C. (2012). Security, Energy, And Performance-aware Resource Allocation Mechanisms For Computational Grids. *Future Generation Computer Systems*.
- Liu, B., Li, J., Chen, C., Tan, W., Chen, Q., dan Zhou, M. (2015). Efficient Motif Discovery for Large-Scale Time Series in Healthcare. *IEEE Transactional on Industrial Informatics*, 11(3), 583-590.
- Lutz, R. E. (2007). Trinucleotide Repeat Disorders. *Seminars in Pediatric Neurology*, 14, 26-33.
- Nairn, J., dan Price, N. C. (2009). *Exploring Proteins: A Student's Guide to Experimental Skills and Methods*. Oxford, Oxfordshire: Oxford University Press.
- Orr, H. T., dan Zoghbi, H. Y. (2007). Trinucleotide Repeat Disorders. *Annual Review of Neuroscience*, 30, 575-621.

- Ostrouchov, G., Chen, W.-C., Schmidt, D., dan Patel, P. (2012). *Programming with Big Data in R*. Retrieved from <http://www.r-pbd.org>
- Pahadia, M., Srivastava, A., Srivastava, D., dan Patil, D. N. (2015). Genome Data Analysis using MapReduce Paradigm. *2015 Second International Conference on Advances in Computing and Communication Engineering* (pp. 556-559). IEEE.
- Parfrey, L. W., Lahr, D. J., dan Katz, L. A. (2008). The Dynamic Nature of Eukaryotic Genomes. *Molecular Biology and Evolution*, 25(4), 787-794.
- Pinel, F., Pecero, J., Khan, S., dan Bouvry, P. (2011). Energy-efficient Scheduling On Milliclusters With Performance Constraints. *ACM/IEEE International Conference on Green Computing and Communications (GreenCom)* (pp. 44-49). Chengdu, Sichuan, China: ACM/IEEE.
- Rossini, A., Tierney, L., dan Li, N. (2007). Simple Parallel Statistical Computing in R. *Journal of Computational and Graphical Statistics*, 16(2), 399-420.
- Rothman, S. S. (2002). *Lessons From The Living Cell: The Culture of Science and The Limits of Reductionism*. New York: McGraw-Hill.
- Sanger, F. (1977). *Nature*, 265(687).
- Schmidberger, M., Morgan, M., Eddelbuettel, D., Yu, H., Tierney, L., dan Mansmann, U. (2009). State of the Art in Parallel Computing with R. *Journal of Statistical Software*, 31(1), 1-27.
- Sommerville, I. (2011). *Software Engineering*. Addison-Wesley.
- Sukanto, R. A., dan Shalahuddin, M. (2011). *Modul Pembelajaran Rekayasa Perangkat Lunak (Terstruktur dan Beroientasi Objek)*. Bandung: Modula.
- The Huntington's Disease Collaborative Research Group. (1993). A Novel Gene Containing a Trinucleotide Repeat That Is Expanded and Unstable on Huntington's Disease Chromosomes. *Cell*, 72, 971-983.
- Turnpenny P, E. S. (2005). *Emery's Elements of Medical Genetics (12th ed.)*. London: Elsevier.
- U.S. National Library of Medicine. (n.d.). *What is DNA?* Retrieved Februari 8, 2017, from Genetics Home Reference: <https://ghr.nlm.nih.gov/primer/basics/dna>
- Ukkonen, E. (1985). Finding Approximate Patterns in Strings. *Journal of Algorithms*, 6, 132-137.
- University of Surrey. (2007, Juni 25). In Silico Cell For TB Drug Discovery. *Science Daily*.

- Valentini, G., Lassonde, W., Khan, S., Min-Allah, N., Madani, S., Li, J., . . . Bouvry, P. (2013). An Overview of Energy Efficiency Techniques In Cluster Computing Systems. *Cluster Computing*, 16(1), 3-15.
- Venter, J. C., Adams, M., Myers, E., Li, P., Mural, R., Sutton, G., . . . Skupski, M. (2001). The Sequence of the Human Genome. *Science*, 291(5507), 1304–1351.
- Vijayarani, D., dan Janani, M. (2017). String Matching Algorithms For Reteriving Information From Desktop – Comparative Analysis. *International Conference on Inventive Computation Technologies (ICICT)*. IEEE.
- Winkler, H. (1920). *Verbreitung und Ursache der Parthenogenesis im Pflanzen- und Tierreiche*. Jena: Verlag Fischer.
- Wyman, A. R., dan White, R. (1980). A Highly Polymorphic Locus in Human DNA. *Proc. Natl. Acad. Sci. U.S.A.*, 77(11), 6754-6758.
- Yangjun Chen, Y. W. (2016). On the Massive String Matching Problem. *2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)* (pp. 350-355). IEEE.
- Yu, H. (2002). Rmpi: Parallel Statistical Computing in R. *R News*, 2(2), 10-14.
- Zhang, L., Peng, Y., Liang, J., Liu, X., Yi, J., dan Wen, Z. (2015). An Improved String Matching Algorithm for HTTP Data Reduction. *2015 International Conference on Intelligent Information Hiding and Multimedia Signal Processing* (pp. 345-348). IEEE.