IMPLEMENTASI *TASK PARALLEL LIBRARY* (TPL) DENGAN METODE *HUFFMAN CODE* UNTUK KOMPRESI TEKS

SKRIPSI

Sebagai salah satu syarat untuk memperoleh gelar
Sarjana dalam bidang Ilmu Komputer

Oleh:
HERU TRI PRASETYO
0510960030-96

PROGRAM STUDI ILMU KOMPUTER
JURUSAN MATEMATIKA
FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
UNIVERSITAS BRAWIJAYA
MALANG

2011

LEMBAR PENGESAHAN SKRIPSI

IMPLEMENTASI *TASK PARALLEL LIBRARY* (TPL)
DENGAN METODE *HUFFMAN CODE*
UNTUK KOMPRESI TEKS

oleh:
HERU TRI PRASETYO
0510960030-96

Setelah dipertahankan di depan Majelis Penguji
Pada tanggal 16 Agustus 2011
dan dinyatakan memenuhi syarat untuk memperoleh gelar
Sarjana Komputer dalam bidang Ilmu Komputer

Pembimbing I,	**Pembimbing II,**
Bayu Rahayudi, ST. MT	**Drs. Achmad Ridok, M. Kom**
NIP. 197407122006041001	**NIP. 196808251994031002**

Mengetahui,
Ketua Jurusan Matematika
Fakultas MIPA Universitas Brawijaya

Dr. Abdul Rouf Alghofari, M. Sc
NIP. 196709071992031001


UNIVERSITAS BRAWIJAYA

LEMBAR PERNYATAAN

Saya yang bertanda tangan di bawah ini :

Nama	**: Heru Tri Prasetyo**
NIM	**: 0510960030-96**
Jurusan	**: Matematika**
Program Studi	**: Ilmu Komputer**
Penulis tugas akhir berjudul	**: Implementasi *Task Parallel Library* (TPL) Dengan Metode *Huffman Code* Untuk Kompresi Teks**

Dengan ini menyatakan bahwa :

1. **Isi dari tugas akhir yang saya buat adalah benar-benar karya sendiri dan tidak menjiplak karya orang lain, selain nama-nama yang termaktub di isi dan tertulis di daftar pustaka dalam Tugas Akhir ini.**
2. **Apabila dikemudian hari ternyata Tugas Akhir yang saya tulis terbukti hasil jiplakan, maka saya akan bersedia menanggung segala resiko yang akan saya terima.**

Demikian pernyataan ini dibuat dengan segala kesadaran.

Malang, 16 Agustus 2011
Yang menyatakan,

Heru Tri Prasetyo
NIM. 0510960030

Implementasi *Task Parallel Library* (TPL)
Dengan Metode *Huffman Code*
Untuk Kompresi Teks

ABSTRAK

Kemajuan teknologi membuat peningkatan kinerja prosesor lebih efisien. Produsen prosesor mengatasi hal ini dengan membuat prosesor *multi core*. Metode *Huffman* adalah salah satu algoritma kompresi. Algoritma ini bekerja dengan cara, karakter yang sering dipakai dikodekan dengan rangkaian bit yang pendek, sedangkan karakter yang jarang dipakai dikodekan dengan rangkaian bit yang panjang. Pada penelitian ini, kompresi dengan metode *Huffman* dilakukan secara paralel menggunakan teknik *task parallel library*.

Task parallel library adalah suatu teknik dimana sistem kerjanya membagi tugas untuk semua prosessor yang ada di mesin. Pembagian tugas untuk prosessor berfungsi untuk meningkatkan kecepatan kompresi secara signifikan.

Untuk mengkompresi data secara paralel, data yang akan dikompresi dipartisi terlebih dahulu, yang nantinya akan dialokasikan ketiap-tiap *core* yang tersedia. Setelah terkompresi, data-data yang terpisah akan digabungkan untuk mendapatkan *file* utuh yang telah terkompresi. Begitu sebaliknya dengan proses dekompresi, yaitu *file* yang telah terkompresi akan dipartisi, yang nantinya akan digabungkan kembali menjadi *file* utuh yang telah terkompresi.

Berdasarkan percobaan yang telah dilakukan, menunjukan bahwa kompresi *file* dengan teknik *task parallel library* menunjukan prosentase waktu rata-rata lebih cepat sebesar 32.35% dibandingkan dengan kompresi metode *Huffman* tanpa *task parallel library*.

Implementation Task Parallel Library (TPL) With Huffman Code Method for Text Compression

ABSTRACT

Technological advances really improves the performance of processors efficiently. Processors manufacturers address this by making multi-core processors. Huffman Method is one kind of many compression algorithms. The algorithm works in a way that the characters which are often used are encoded with series of short bits, while the rarely used characters are encoded with series of long bits. In this study, Huffman compression method is performed in parallel using parallel task library.

Task Parallel Library is a technique where the system works by sharing responsibility for each processor in the machine. This is done to improve the compression speed significantly.

To compress the data in parallel, the data to be compressed should be partitioned before, which then will be allocated to each core available. Once compressed, the separated data will be combined to get an intact file. So contrary to the decompression process, the file that has been compressed will be partitioned first, which then merged back into a whole file again.

Based on experiments that have been conducted, it shows that the compressed file using parallel task library shows the percentage of average time faster by 32.35% compared to Huffman compression method without using task parallel library.

KATA PENGANTAR

Puji syukur penulis panjatkan ke hadirat Tuhan Yang Maha Esa yang telah melimpahkan segala berkat, anugerah, hikmat, serta kasih-Nya yang luar biasa sehingga penulis dapat mernyelesaikan proposal skripsi dengan judul "Implementasi *Task Parallel Library* (TPL) Dengan Metode *Huffman Code* Untuk Kompresi Teks".

Skripsi ini diajukan sebagai salah satu syarat untuk memperoleh gelar Sarjana Komputer di Fakultas MIPA, Jurusan Ilmu Komputer, Universitas Brawijaya Malang. Atas terselesaikannya skripsi ini penulis mengucapkan terima kasih pada :

1. Bayu Rahayudi, ST. MT., selaku pembimbing utama dalam penulisan skripsi ini.
2. Drs. Achmad. Ridok, M. Kom., selaku pembimbing pendamping dalam penulisan skripsi ini.
3. Drs. Marji, MT., selaku Ketua Program Studi Ilmu Komputer Jurusan Matematika FMIPA Universitas Brawijaya.
4. Dr. Abdul Rouf Alghofari, M. Sc., selaku Ketua Jurusan Matematika Fakultas MIPA Universitas Brawijaya.
5. Segenap bapak dan ibu dosen yang telah mendidik dan mengajarkan ilmunya kepada penulis selama menempuh pendidikan di Program Studi Ilmu Komputer Jurusan Matematika FMIPA Universitas Brawijaya.
6. Segenap staf dan karyawan di Jurusan Matematika FMIPA Universitas Brawijaya yang telah banyak membantu Penulis dalam pelaksanaan penyusunan tugas akhir ini.
7. Kedua orang tua yang tak pernah berhenti memberikan doa dan dukungannya.
8. Widia Nur Diana yang telah memberikan banyak bantuan, dorongan serta motivasi sehingga skripsi ini dapat terselesaikan.
9. Muhammad Zaenuri dan Vebnu Hartono yang selalu menjadi sahabat terbaik saat suka dan duka.
10. Rizky Zulkarnain, Martheen, Dani, Shintia, Yulian Budi Laksono, Danu, Muhammad Rizky, Rohmatullah Al Amin, Mas Candra, Mas Soni di Program Studi Ilmu Komputer Universitas Brawijaya yang telah banyak memberikan bantuannya demi kelancaran pelaksanaan penyusunan tugas akhir ini.

11. Semua pihak yang telah membantu terselesaikannya penyusunan skripsi ini yang tidak dapat disebutkan satu per satu.

Akhirnya hanya kepada Allah SWT kita kembalikan semua urusan dan semoga skripsi ini dapat bermanfaat bagi semua pihak, Penulis menyadari bahwa dalam penyusunan skripsi ini banyak kekurangan, untuk itu saran dan kritik yang membangun demi kesempurnaan penulisan selanjutnya sangat diharapkan. Semoga skripsi ini dapat bermanfaat untuk semua pihak.

Malang, 16 Agustus 2011

Penulis

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

DAFTAR SOURCECODE

BAB I
PENDAHULUAN

1.1 Latar Belakang

Seiring dengan perkembangan teknologi, maka komputer mengalami perkembangan baik dari segi bentuk, ukuran dan kemampuan. Sejak tahun 2000an perkembangan *processor* beralih dari era *single core* menjadi *multi core*. *Multi core* adalah suatu teknologi yang diaplikasikan dalam perangkat lunak yang berfungsi untuk mencapai perbaikan kinerja. Menurut Schauer, Bryan pada tahun (2005) *Multi core* adalah sebuah komponen tunggal dengan dua atau lebih prosessor (*core*) yang merupakan unit membaca atau mengeksekusi program.

Kompresi adalah suatu proses yang melakukan konversi sebuah input data *stream* (*stream* sumber atau data mentah asli) menjadi stream lainnya (*bitsream* hasil, atau *stream* yang telah terkompresi) yang berukuran lebih kecil (Salomon, D : 2002). Sedangkan menurut Pu pada tahun (2006) Kompresi adalah sebuah penyajian informasi ke dalam bentuk yang lebih sederhana.

Salah satu metode yang popular digunakan dalam kompresi data adalah metode *Huffman*. Metode ini mempunyai prinsip dasar yang cukup sederhana. Algoritma *Huffman* mempunyai prinsip pengkodean, dimana setiap karakter diberi kode biner. Karakter yang sering dipakai, dikodekan dengan rangkaian bit yang pendek, sedangkan karakter yang jarang dipakai, dikodekan dengan rangkaian bit yang lebih panjang.

Task parallel library adalah suatu metode dimana sistem kerjanya membagi tugas untuk semua prosessor yang ada di mesin. Pembagian tugas untuk prosessor berfungsi untuk meningkatkan kecepatan kompresi secara signifikan. Selain itu, berguna untuk memanfaatkan sumber daya mesin yang tersedia secara efisien untuk kepentingan pengguna.

Sebelumnya penelitian mengenai kompresi pernah dilakukan oleh Laksono pada tahun 2010 dengan judul, Implementasi Transformasi Star Pada Kompresi Teks Menggunakan Algoritma *Huffman*. Namun pada penelitian tersebut tidak memanfaatkan *Multi core* dalam rancangan *Task Parallel Libray* pada sistem yang dibuat. Sedangkan penelitian menggunakan *task parallel library* sebelumnya pernah dilakukan oleh Bharath, K A pada tahun 2010 dengan judul

Parallel fast compression unleashing the power of multi-core machines using the .NET Task Parallel Library (TPL) dan dalam penelitian tersebut menunjukkan bahwa dengan TPL didapatkan waktu yang lebih cepat dibandingkan dengan metode tanpa TPL. Tetapi pada penelitian tersebut tidak menggunakan algoritma *Huffman* sebagai metode kompresinya dan hanya dilakukan pada satu *file*.

Untuk mengatasi permasalahan rendahnya kecepatan kompresi pada kompresi *Huffman* tanpa teknik *Task Parallel Library*. Maka dalam penelitian ini diterapkan teknik *Task Parallel Library* pada metode *Huffman* dengan melakukan proses kompresi pada beberapa *file*.

Berdasarkan latar belakang yang telah dipaparkan, maka pada skripsi ini diberikan judul "**IMPLEMENTASI *TASK PARALLEL LIBRARY* (TPL) DENGAN METODE *HUFFMAN CODE* UNTUK KOMPRESI TEKS**".

1.2 Rumusan Masalah

Berdasarkan latar belakang yang telah dijelaskan maka rumusan masalah yang dikerjakan adalah :

1. Bagaimana mengimplementasikan *task parallel library* pada kompresi *file* menggunakan metode *Huffman.*
2. Berapa besar kecepatan kompresi metode *Huffman* menggunakan *task parallel library* dibandingkan dengan metode *Huffman* tanpa *task parallel library*.

1.3 Batasan Masalah

Berdasarkan permasalahan diatas, berikut ini diberikan batasan untuk menghindari melebarnya masalah yang akan diselesaikan:

1. Penelitian proses kompresi metode *Huffman* menggunakan *task parallel library* dilakukan pada satu perangkat komputer.
2. *File* yang dikompresi adalah *file* dengan format .doc, .txt dan .pdf.
3. Pengkompresian dilakukan pada beberapa *file* atau folder yang berisi lebih dari satu *file*.

1.4 Tujuan

Tujuan yang ingin dicapai dalam penelitian dan penulisan skripsi ini yaitu membandingkan serta mengetahui kecepatan proses kompresi metode *Huffman* menggunakan *task parallel library* dengan metode *Huffman* tanpa *task parallel library* dalam menyelesaikan masalah kompresi *file*.

1.5 Manfaat

Manfaat yang akan dicapai dari skripsi ini adalah dihasilkannya perangkat lunak yang menerapkan metode *Huffman* menggunakan *task parallel library* dan mampu membantu menyelesaikan permasalahan rendahnya kecepatan proses kompresi.

1.6 Sistematika Penulisan

Sistematika penulisan skripsi ini adalah sebagai berikut :

BAB I : PENDAHULUAN

Pada bab ini membahas mengenai latar belakang, rumusan masalah, batasan masalah, tujuan, manfaat dan sistematika penulisan skripsi.

BAB II : TINJAUAN PUSTAKA

Bab ini menjelaskan mengenai dasar teori yang terkait dengan topik penulisan tugas akhir yang diangkat menjadi acuan dasar dalam pembuatan sistem kompresi *file*.

BAB III : METODOLOGI DAN PERANCANGAN

Bab ini menjelaskan mengenai metode dan perancangan yang akan dilakukan dalam membangun tahapan-tahapan kompresi *file*.

BAB IV : IMPLEMENTASI DAN PEMBAHASAN

Bab ini menjelaskan hasil implementasi dari metode dan perancangan yang telah dijelaskan pada bab sebelumnya. Hal yang dijelaskan adalah: implementasi program, uji coba, dan analisa hasil kompresinya.

BAB V : PENUTUP

Bab ini berisi kesimpulan dari penelitian yang telah dilakukan, dan saran untuk pengembangan penelitian selanjutnya.


UNIVERSITAS BRAWIJAYA

BAB II
TINJAUAN PUSTAKA

Bab ini mencantumkan beberapa tinjauan pustaka dan referensi-referensi yang berkaitan dengan penelitian ini. Pada bab ini akan dijelaskan mengenai pengertian dan konsep dasar kompresi secara umum. Kemudian penjelasan mengenai kompresi menggunakan metode *Huffman* dengan menggunakan *task parallel library*. Selanjutnya akan diberikan rumus rasio kompresi untuk pengujian yang akan dilakukan.

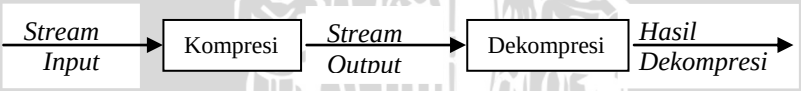
2.1 *Task Parallel Library*

Task Parallel Library adalah sekumpulan tipe *public* dan APIs di *system threading* dan *sytem threading* tasks *namespaces* di .net Framework versi 4.

Tujuan dari TPL adalah melakukan proses penambahan *parallelisasi* dan *concurrency* (proses berjalan bersamaan) untuk aplikasi, sehingga dapat memaksimalkan kinerja suatu proses.

2.2 Kompresi

Kompresi data adalah proses mengubah *stream* data masukan menjadi *stream* data keluaran agar menjadi lebih kecil, sedangkan proses pembalikan data yang sudah terkompres menjadi data semula disebut dengan dekompresi. Gambar 2.1 mengilustrasikan proses kompresi dan dekompresi data.



Gambar 2.1. Proses pengolahan data (Held, G. 1998)

Teknik kompresi data berdasarkan outputnya dibagi menjadi dua kategori, yaitu:

1. *Lossy Compression*

   Lossy Compression menyebabkan adanya perubahan data dibandingkan sebelum dilakukan proses kompresi. Sebagai gantinya *lossy compression* memberikan derajat kompresi lebih tinggi. Tipe ini cocok untuk kompresi *file* suara digital dan gambar digital. *File* suara dan gambar secara alamiah masih bisa digunakan walaupun tidak berada pada kondisi yang sama sebelum dilakukan kompresi.

2. *Lossless Compression*

   Sebaliknya *lossless compression* memiliki derajat kompresi yang lebih rendah tetapi dengan akurasi data yang terjaga antara sebelum dan sesudah proses kompresi. Kompresi ini cocok untuk basis data, dokumen atau spreadsheet. Pada *lossless compression* ini tidak diijinkan ada bit yang hilang dari data pada proses kompresi (Widhiartha, 2008).

Huffman menggunakan metode statik yang selalu menggunakan peta kode yang sama. Berdasarkan teknik pengkodean simbol yang digunakan, metode kompresi dapat dibagi dalam tiga kategori :

1. Metode *symbolwise*

   Dalam metode ini peluang kemunculan dari tiap simbol dalam *file*, input dihitung, lalu mengkodekan satu simbol dalam satu waktu, dimana simbol yang lebih sering muncul diberi kode lebih pendek dibandingkan simbol yang lebih jarang muncul. Metode ini diterapkan pada algoritma *Huffman*.
2. Metode *dictionary*

   Pada metode ini karakter dalam *file* input digantikan dengan indeks lokasi dari karakter tersebut dalam sebuah *dictionary*. Metode ini diterapkan pada algoritma LZW.
3. Metode *predictive*

   Metode ini menggunakan model FSA (*finite state automa*) untuk memprediksi distribusi probabilitas dari simbol-simbol selanjutnya. Metode ini diterapkan pada algoritma DMC (Callista, 2007).

2.3 Algoritma *Huffman*

Pada tahun 1951, David A. Huffman dalam kelas Informasi Teori di MIT diberikan pilihan untuk membuat sebuah *term paper* atau mengikuti ujian akhir. Pada saat itu pilihan *term paper* yang diberikan profesor Robert M. Fano adalah tentang menemukan kode biner yang paling efisien. Tidak dapat membuktikan kode apapun yang paling efisien, Huffman hampir menyerah dan mulai belajar untuk mengikuti ujian akhir saja, ketika ia menemukan ide untuk menggunakan pohon biner dengan pengurutan berdasarkan kekerapan dan berhasil membuktikan bahwa cara ini adalah yang paling efisien (Prabawa, 2008).

Ada dua jenis metode *huffman*, yaitu:

1. *Static Huffman Coding*
   Frekuensi karakter dari string yang akan dikompres dianalisa terlebih dahulu. Selanjutnya dibuat pohon huffman yang merupakan pohon biner dengan root awal yang diberi nilai 0 (sebelah kiri) dan 1 (sebelah kanan).
2. *Adaptive Huffman Coding*
   Metode SHC mengharuskan kita mengetahui terlebih dahulu frekuensi masing-masing karakter sebelum dilakukan proses pengkodean. Metode AHC merupakan pengembangan dari SHC dimana proses penghitungan frekuensi karakter dan pembuatan pohon Huffman dibuat secara dinamis pada saat membaca data.

Prinsip kode *Huffman* adalah mengganti karakter yang paling sering muncul di dalam data dengan kode yang lebih pendek, sedangkan karakter yang lebih jarang muncul dikodekan dengan kode yang lebih panjang. Algoritma *Huffman* menerapkan metode statik yaitu menggunakan peta kode yang selalu sama. Metode ini membutuhkan dua fase yaitu fase pertama untuk menghitung probabilitas kemunculan tiap karakter dan menentukan peta kodenya dan fase kedua untuk mengubah pesan menjadi kumpulan kode yang akan ditransmisikan (Callista, 2007).

2.3.1 Pembentukan Pohon *Huffman*

Kode *Huffman* pada dasarnya adalah himpunan yang berisi sekumpulan kode biner yang direpresentasikan dari pohon biner yang diberikan nilai atau label. Untuk cabang kiri pada pohon biner diberikan label 0, sedangkan pada cabang kanan diberikan label 1. Rangkaian bit yang terbentuk pada setiap lintasan dari akar ke daun merupakan pengkodean untuk karakter yang berpadanan. Pohon biner ini biasa disebut pohon *Huffman* (*Huffman tree*). Langkah-langkah pembentukan pohon *Huffman* menurut Cellista (2007) adalah sebagai berikut:

1. Baca semua karakter di dalam data untuk menghitung frekuensi kemunculan setiap karakter. Setiap karakter penyusun data dinyatakan sebagai pohon bersimpul tunggal. Dan setiap simpul ini di-*assign* dengan frekuensi kemunculan karakter tersebut.
2. Terapkan strategi *greedy* dengan menggabungkan dua buah pohon yang mempunyai frekuensi terkecil pada sebuah akar.

Akar mempunyai frekuensi yang merupakan jumlah dari frekuensi dua buah pohon penyusunnya.

3. Ulangi langkah dua sampai hanya tersisa satu buah pohon *Huffman*. Agar pemilihan dua pohon yang akan digabungkan berlangsung dengan cepat, maka semua pohon yang ada selalu terurut menaik berdasarkan frekuensi.
4. Baca kembali karakter-karakter di dalam data, kodekan setiap karakter dengan kode *Huffman* yang bersesuaian.

Misalnya, data dengan panjang 100 karakter dan disusun oleh huruf-huruf a, b, c, d, e, f dengan frekuensi kemunculan setiap huruf seperti ditunjukkan pada tabel 2.1 :

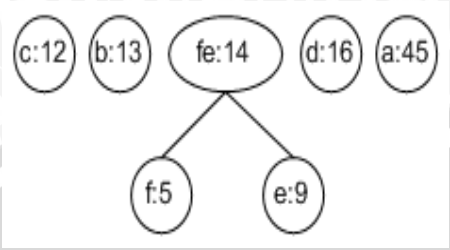
Tabel 2.1 Frekuensi kemunculan huruf

Karakter	Frekuensi
a	45
b	13
c	12
d	16
e	9
f	5

Langkah pertama menghitung kemunculan tiap karakter, kemudian tiap karakter dinyatakan sebagai pohon bersimpul tunggal seperti ditunjukkan pada gambar 2.2.

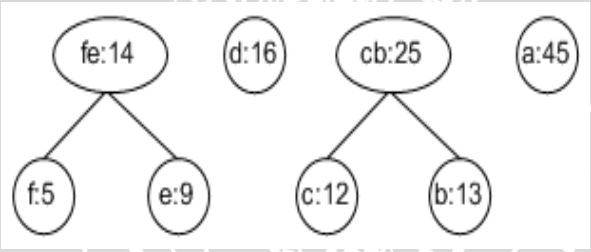
Gambar 2.2 Frekuensi kemunculan karakter

Pada langkah kedua diambil akar yang memiliki jumlah kemunculan terkecil, seperti ditunjukkan pada gambar 2.3. Dalam hal ini karakter yang memiliki nilai frekuensi terkecil adalah F dan E. Karakter F dan E kemudian dibentuk menjadi akar dari pohon yang baru.



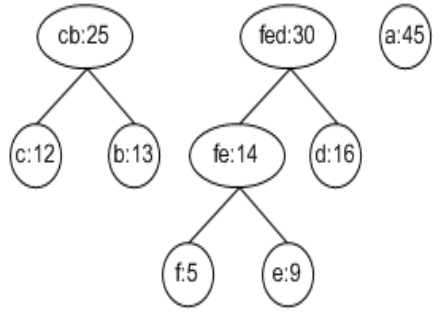
Gambar 2.3 Pembentukan Pohon *Huffman* Langkah Kedua

Pada langkah ketiga diambil lagi 2 akar dari pohon yang memiliki jumlah kemunculan terkecil, seperti ditunjukkan pada gambar 2.4. Dalam hal ini adalah akar C dan B. Akar C dan B digabung sehingga membentuk akar dari pohon yang baru dengan jumlah frekuensi sama dengan hasil penjumlahan frekuensi akar C dan B.



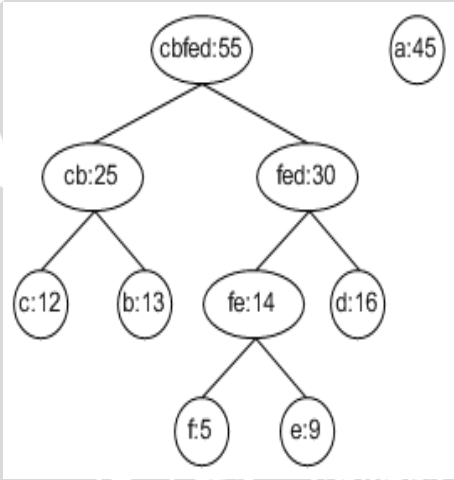
Gambar 2.4 Pembentukan Pohon *Huffman* Langkah Ketiga

Pada langkah keempat, dua akar terkecil pada langkah ketiga digabungkan kembali untuk membentuk pohon dengan akar yang baru dengan frekuensi yang merupakan penjumlahan dari frekuensi akar pembentuknya, seperti pada gambar 2.5. Akar terkecil yang diambil pada langkah ketiga adalah FE dan D.

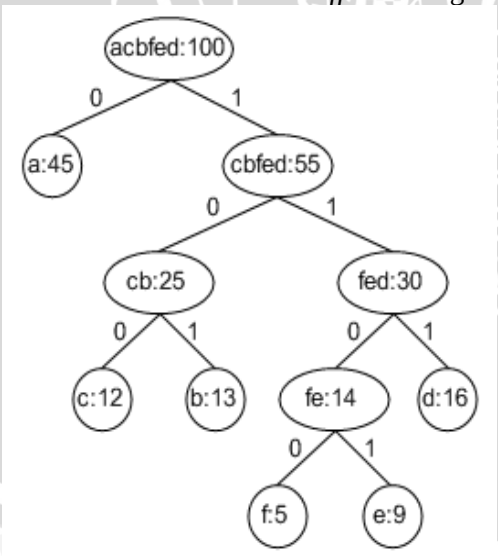


Gambar 2.5 Pembentukan Pohon *Huffman* Langkah Keempat

Pada langkah kelima akar FED dan CB digabungkan untuk membentuk pohon yang baru seperti pada gambar 2.6. Langkah-langkah sebelumnya dilanjutkan sampai semua akar yang ada telah digabung, sehingga membentuk satu pohon yang baru dengan akar yang memiliki frekuensi sama dengan jumlah frekuensi seluruh akar pertama kali, seperti pada gambar 2.7. Dalam hal ini seluruh akar pada frekuensi karkater.



Gambar 2.6 Pembentukan Pohon *Huffman* Langkah Kelima



Gambar 2.7 Pohon *Huffman*

2.3.2 Proses *Encoding*

Proses untuk melakukan pembentukan kode dari suatu data tertentu disebut *encoding*. Dalam hal ini, kode *Huffman* akan terbentuk sebagai suatu kode biner. Kode *Huffman* didapatkan dengan membaca setiap kode dari daun simbol tersebut hingga ke akarnya. Ketika suatu kode *Huffman* telah dibentuk, suatu data dapat dengan mudah di-*encode* dengan cara mengganti setiap simbol menggunakan kode yang telah dibentuk.

Langkah-langkah untuk meng-*encode* suatu string biner menurut Wardoyo (2005) adalah sebagai berikut:

1. Tentukan karakter yang akan di-*encode*
2. Baca dari akar, baca setiap bit yang ada pada cabang yang bersesuaian sampai ditemukan cabang dimana karakter itu berada.
3. Ulangi langkah 2 sampai seluruh karakter di-*encode*.

Ketika suatu kode *Huffman* telah dibentuk, suatu data dapat dengan mudah di-*encode* dengan mengganti setiap simbol menggunakan kode yang telah dibentuk. Sebagai contoh dapat dilihat tabel 2.2, yang merupakan hasil *encoding* untuk pohon *Huffman* pada contoh sebelumnya.

Tabel 2.2 *Huffman* hasil *encoding*

Simbol	Frekuensi	Peluang	Kode *Huffman*
a	45	45/100	0
b	13	13/100	101
c	12	12/100	100
d	16	16/100	111
e	9	9/100	1101
f	5	5/100	1100

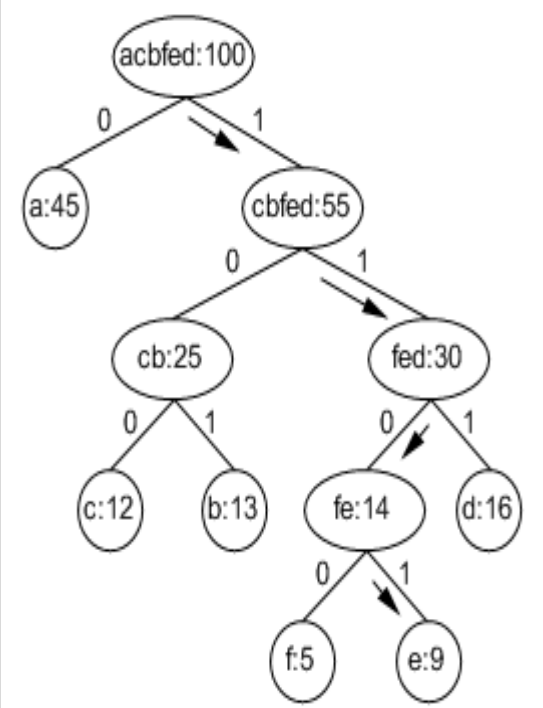
2.3.3 Proses *Decoding*

Decoding merupakan proses yang mengembalikan suatu data dari suatu kode tertentu. Proses *decoding* ini merupakan kebalikan dari proses *encoding*. *Decoding* dapat dilakukan dengan dua cara, yaitu cara pertama dengan menggunakan pohon *Huffman* dan cara kedua dengan menggunakan tabel kode *Huffman* (Ayuningtyas, 2007).

Langkah-langkah men-*decoding* suatu string biner dengan menggunakan pohon *Huffman* menurut Wardoyo (2005) adalah sebagai berikut:

1. Baca sebuah bit dari string biner.
2. Untuk setiap bit pada langkah 1, lakukan pembacaan pada cabang yang bersesuaian.
3. Ulangi langkah 1 dan 2 sampai bertemu *node*. Kodekan rangkaian yang telah dibaca dengan karakter di daun.
4. Ulangi dari langkah 1 sampai semua bit di dalam *string* habis.

Sebagai contoh akan dilakukan pen-*decoding*-an suatu string biner yang bernilai "1101". Setelah dilakukan penelusuran pada pohon *Huffman*, akan ditemukan bahwa string yang mempunyai kode *Huffman* "1101" adalah karakter E. Seperti ditunjukkan pada gambar 2.8.



Gambar 2.8 Proses *Decoding* menggunakan Pohon *Huffman*

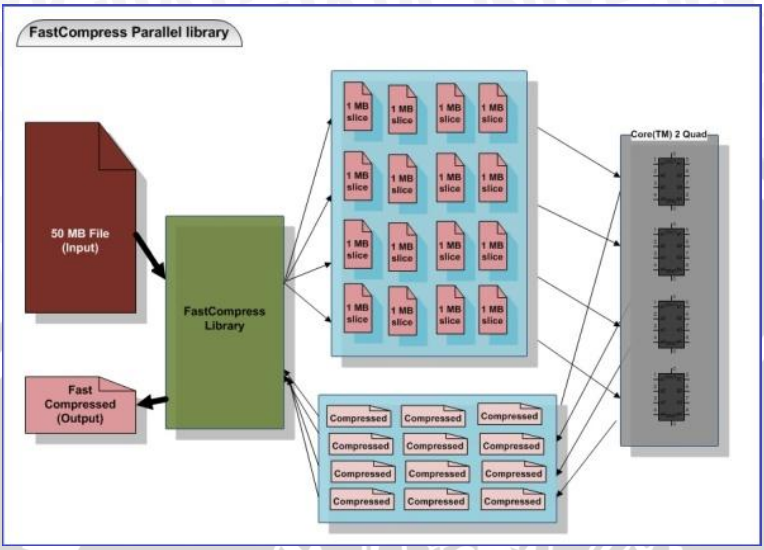
Cara kedua adalah menggunakan tabel kode *Huffman*. Sebagai contoh, akan digunakan kode *Huffman* pada tabel 2.2 untuk

merepresentasikan *string* "ACE". Dengan menggunakan tabel 2.2 *string* tersebut akan direpresentasikan menjadi rangkaian bit : 0 100 1101.Total jumlah bit yang dibutuhkan sebanyak 8 bit. Dari tabel 2.2 tampak bahwa kode untuk sebuah simbol atau karakter tidak boleh menjadi awalan dari kode simbol yang lain untuk menghindari ambiguitas dalam proses dekompresi atau *decoding*. Karena tiap kode *Huffman* yang dihasilkan unik, maka proses *decoding* dapat dilakukan dengan mudah. Contoh: saat membaca kode bit pertama dalam rangkaian bit "01001101", didapatkan bit "0", cek kode "0" pada tabel 2.2, Didapatkan bahwa bit "0" adalah pemetaan dari huruf A, kemudian baca bit selanjutnya sehingga menjadi "1", tidak terdapat kode "1" pada tabel. Kemudian baca bit selanjutnya yaitu "10", tidak terdapat kode "10" pada tabel, maka baca bit selanjutnya sehingga menjadi "100" dapat disimpulkan bahwa bit "100" merupakan pemetaan dari karakter "C". Kemudian baca kode bit selanjutnya, yaitu bit "1", tidak terdapat kode *Huffman* "1", lalu baca kode bit selanjutnya, sehingga menjadi "11". Tidak terdapat juga kode *Huffman* "11", lalu baca lagi kode bit selanjutnya, sehingga menjadi "110". Tidak terdapat juga kode "110", lalu baca lagi bit selanjutnya sehingga didapatkan rangkaian bit "1101" adalah pemetaan dari simbol "E".

2.4 Paralelisasi *Huffman* menggunakan *Task Pararel Library*

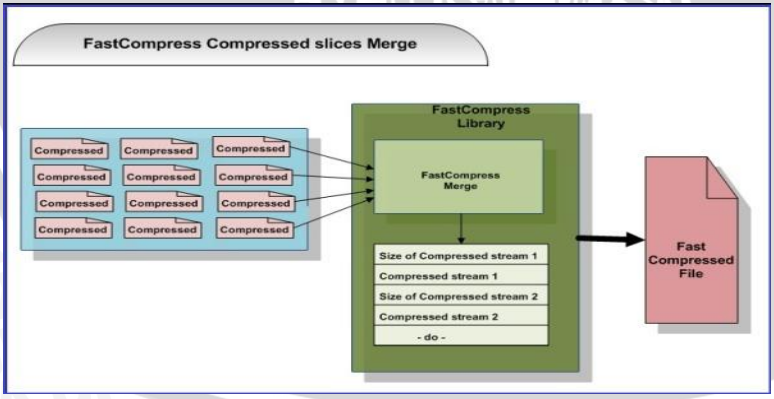
Paralelisasi *Hufman* menggunakan *Task pararel Library* di awali dengan melakukan partisi terhadap *file*. Setelah sebuah *file* terpartisi menjadi beberapa bagian maka selanjutnya partisi dari *file* tersebut dapat dikerjakan *oleh Task pararel Library*.

Partisi *file* ini nantinya dialokasikan agar dapat dikerjakan oleh setiap prosesor yang tersedia. *TPL* melakukan pekerjaan skala secara otomatis untuk jumlah prosesor di komputer pengguna. Jika mesin pengguna memiliki 8 prosesor kemungkinan besar partisi 8 pertama akan diambil untuk eksekusi *parallel* di 8 prosesor. Ilustrasi kompresi menggunakan TPL dapat dilihat gambar 2.9.



Gambar 2.9 Kompresi Menggunakan Paralel

Langkah selanjutnya adalah untuk menggabungkan partisi ke dalam sebuah *file* terkompresi tunggal. Proses penggabungan secara sederhana tidak akan dapat bekerja. Hal ini karena data partisi mungkin dengan ukuran yang berbeda, yaitu setiap partisi 1 MB tidak selalu dikompres menjadi masing-masing 10 KB, ukuran ini bisa bervariasi. Kemudian hasil kompresi partisi akan digabungkan untuk mendapatkan *file* asli. Seperti yang ditunjukkan pada gambar 2.10.



Gambar 2.10 Penggabungan Blok-Blok Kompresi

Fast Compress dicapai dengan membagi tugas untuk semua prosesor yang ada di mesin. Hal ini meningkatkan kecepatan kompresi secara signifikan. Disamping itu juga memanfaatkan sumber daya mesin yang tersedia secara efisien untuk kepentingan pengguna. Kinerja *Fast Compress* berbanding lurus dengan jumlah prosesor pada mesin. Dekompresi *Fast Compress* juga mempunyai kecepatan yang sama dengan proses kompresi. (K A, 2010).

2.5 Rasio Kompresi

Rasio kompresi merupakan perbandingan antara ukuran *file* setelah dikompresi dengan ukuran *file* asli (sebelum dikompresi). Persentase kompresi dapat dinyatakan dengan persamaan 2.1 (Salomon, 2004). Semakin besar nilai rasio kompresi, hasil kompresi semakin baik.

$$R = (1 - \left(\frac{K}{A}\right)) \times 100\% \qquad (2.1)$$

Dimana: A = ukuran *file* asli
K = ukuran *file* hasil kompresi
R = rasio kompresi

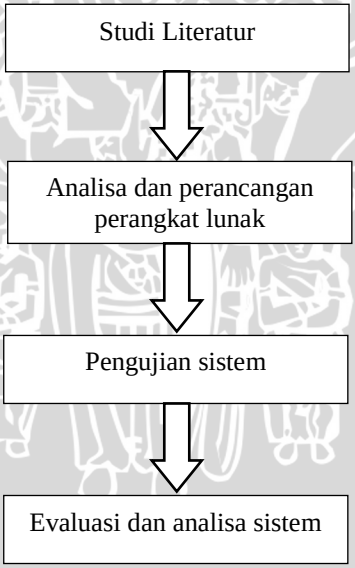
BAB III
METODOLOGI DAN PERANCANGAN

Pada bab ini akan dibahas mengenai metode dan tahap-tahap yang digunakan dalam pembuatan aplikasi yang mengimplementasikan kompresi *Huffman code* menggunakan *task parallel library*.

Penelitian dilakukan dengan tahapan-tahapan sebagai berikut:

1. Mempelajari literatur yang terkait dengan kompresi *file*, *task parallel library*, dan algoritma Huffman.
2. Melakukan perancangan sistem kompresi *file* dan mengimplementasikan menjadi sebuah perangkat lunak.
3. Melakukan uji sistem dengan melakukan kompresi pada *file* yang digunakan sebagai data dalam penelitian ini.
4. Mengevaluasi tingkat keberhasilan sistem dan melakukan analisa terhadap kompresi yang telah dilakukan.

Langkah-langkah yang digunakan dapat dilihat pada gambar 3.1.



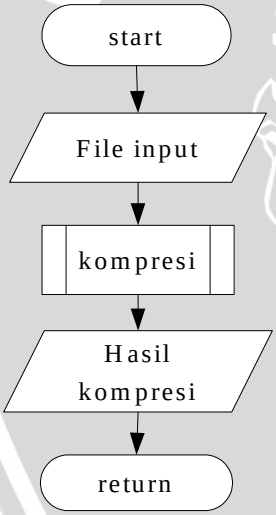
Gambar 3.1 Langkah-langkah penelitian

3.1 Data yang Digunakan

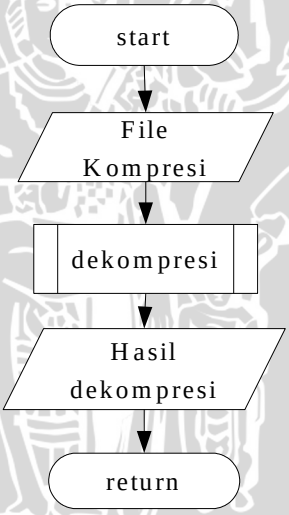
Data yang digunakan dalam penelitian ini adalah data dengan berbagai ukuran mulai dari yang kecil sampai data berukuran besar. Data ini digunakan pada uji coba proses kompresi sebagai *file* yang dilakukan proses kompresi. Terdapat tiga *file* dokumen dengan ukuran yang bervariasi.

3.2 Rancangan Sistem

Rancangan sistem kompresi *Huffman* seperti yang ditunjukkan pada Gambar 3.2 dan Gambar 3.3 menjelaskan mengenai alur kompresi *Huffman*. Alur kerja diawali memasukkan *file* kemudian akan dikompresi yang ditunjukkan pada Gambar 3.2 *Flowchart* Sistem Kompresi. Untuk mengembalikan *file* ke bentuk semula perlu adanya proses dekompresi yang ditunjukkan pada Gambar 3.3 *Flowchart* Sistem Dekompresi



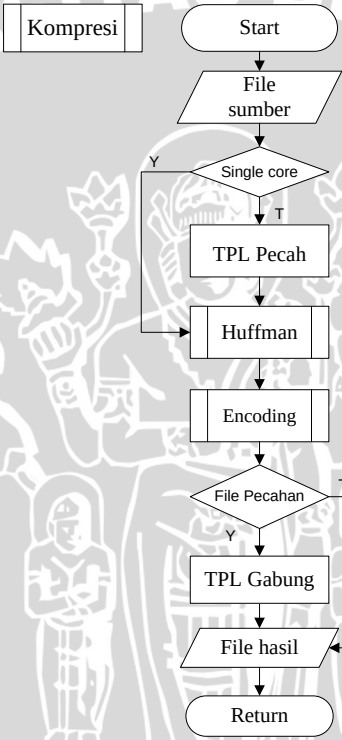
Gambar 3.2 *Flowchart* Sistem Kompresi



Gambar 3.3 *Flowchart* Sistem Dekompresi

3.3 Rancangan Proses Kompresi

Dalam penelitian ini data berupa *file* yang dilakukan kompresi akan melalui beberapa tahap. Terdapat empat proses utama yang terdapat pada sistem ini, yaitu proses *task parallel library* pisah dan *task parallel library* gabung, serta proses kompresi dan dekompresi. Tahapan proses kompresi dapat dilihat pada gambar 3.4. Berikut ini akan dijelaskan secara lebih rinci untuk masing-masing proses yang dijalankan oleh sistem ini.



Gambar 3.4 *Flowchart* Proses Kompresi

3.3.1 *Task Parallel Library*

Sebelum dilakukan proses kompresi *file* sumber akan di pecah menjadi beberapa bagian dengan menggunakan teknik TPL Pecah, *file* yang sudah terpecah akan dikompresi menggunakan metode *Huffman* kemudian digabung kembali dengan menggunakan teknik TPL Gabung.

3.3.1.1 TPL Pecah

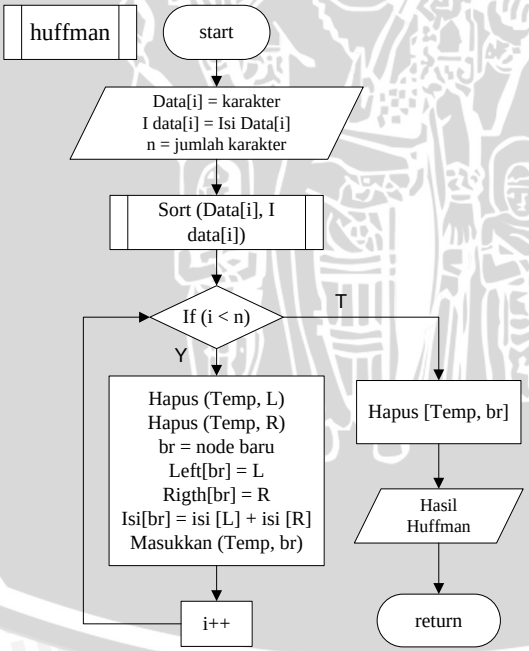
Teknik ini bekerja dengan cara memecah *file* sumber menjadi beberapa bagian yang ditentukan oleh *user*. Besar kecilnya ukuran *file* tersebut di cari berapa ukuran yang paling optimal. Tiap bagian memiliki sebuah nilai index yang spesifik, untuk menandai urutan pecahan *file*. Bagian-bagian ini akan didistribusikan pada masing-masing prosesor agar tiap prosesor dapan bekerja secara independen.

3.3.1.2 TPL Gabung

Teknik ini digunakan untuk menggabungkan *file* sumber yang sudah terpecah dan terkompresi, agar menjadi *file* kompresi tunggal. Penggabungan dilakukan dengan menggunakan indek yang telah dibuat pada proses TPL Pecah.

3.3.2 Kompresi *Hufman*

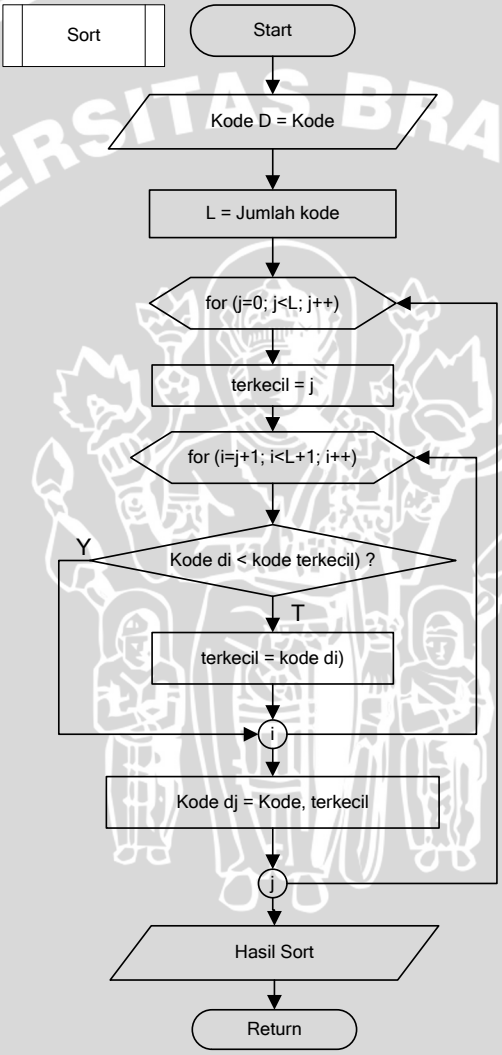
Metode kompresi yang digunakan dalam penelitian ini adalah metode *Huffman* Statik, seperti yang telah dijelaskan pada subbab 2.2. *Flowchart* Kompresi *Huffman* ditunjukkan pada gambar 3.5.



Gambar 3.5 *Flowchart* Kompresi *Huffman*

3.3.2.1 Proses *Sorting*

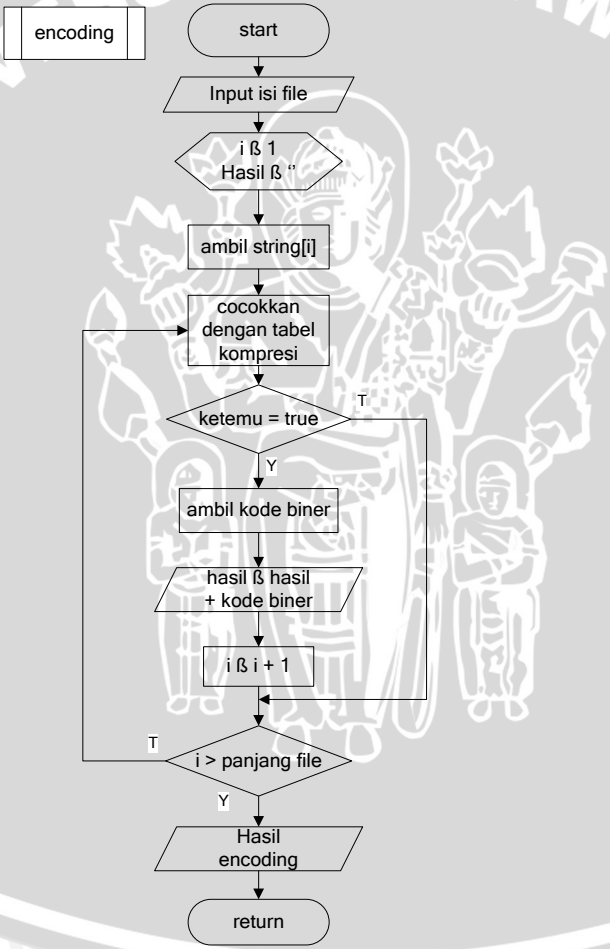
Flowchart sorting digunakan untuk mengurutkan karakter dari frekuensi kecil ke besar, proses ini bertujuan membuat *node* dalam kompresi *Huffman*. *Flowchart* proses *sorting* ditunjukkan pada gambar 3.6.



Gambar 3.6 *Flowchart* proses *Sorting*

3.3.3 Proses *Encoding*

Proses *encoding* bertujuan untuk menyusun *string* biner dari karakter dalam *file* yang akan dikompresi. *String* biner didapat dari karakter yang akan digunakan pada proses kompresi menggunakan metode *Huffman*. Setiap karakter akan dibandingkan dengan kode yang ada pada tabel kompresi, kemudian karakter tersebut akan digantikan oleh kode yang terdapat pada tabel jika terjadi kecocokan pada saat dilakukan perbandingan. *Flowchart* proses *encoding* ditunjukkan pada gambar 3.7.



Gambar 3.7 *Flowchart* proses *Encoding*

3.3.4 Dekompresi

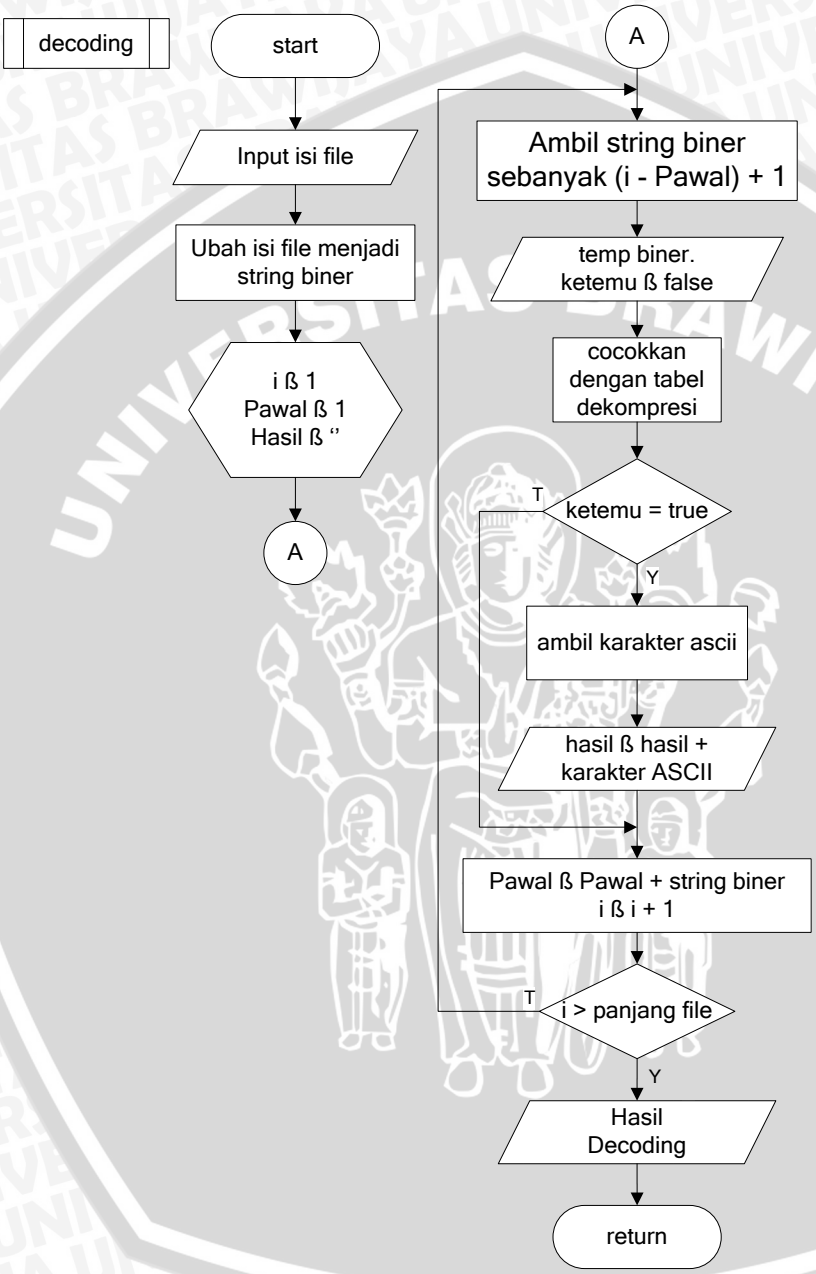
Proses dekompresi adalah proses pengembalian *file* yang telah dikompresi ke *file* asli sebelum dikompresi. Langkah-langkah yang dilakukan dalam proses dekompresi akan dijelaskan pada subbab berikut. *Flowchart* dekompresi ditunjukkan pada gambar 3.8.

Gambar 3.8 *Flowchart* Dekompresi

3.3.4.1 Proses *Decoding*

Proses *decoding* bertujuan untuk menyusun kembali sebuah karakter dari string biner pada *file* yang telah terkompresi. Proses ini dilakukan dengan cara membaca bagian ketujuh pada *file* terkompresi yang berisi kumpulan karakter ASCII. Kumpulan karakter tersebut akan dikonversi kembali ke bentuk biner. Setelah dilakukan konversi, string biner hasil konversi akan dipotong pada bagian paling belakang. Pemotongan dilakukan dengan cara menghilangkan bagian 0 sejumlah bilangan yang dibaca pada bagian enam *header file*. Bagian ini merupakan bagian yang sengaja ditambahkan untuk proses konversi dari bilangan biner ke ASCII sehingga tidak diperlukan pada saat *decoding file*.

Proses selanjutnya adalah membandingkan *string* biner yang tersisa dengan tabel dekompresi yang telah dibentuk. Pembandingan dilakukan per bilangan biner, jika tidak ditemukan akan dilanjutkan ke bilangan biner berikutnya. Proses dilanjutkan hingga terdapat kecocokan. Kemudian karakter pada tabel dekompresi akan menggantikan bilangan biner yang bersesuaian. *Flowchart* proses *decoding* ditunjukkan pada gambar 3.9.



Gambar 3.9 *Flowchart* proses *Decoding*

3.5 Contoh Perhitungan Manual

Sebagai contoh, terdapat *file* teks yang akan dikompresi dengan panjang 10 karakter yaitu “matematika”. Dalam kode ASCII kalimat tersebut membutuhkan representasi 10 x 8 = 80 bit (10 byte) dengan rincian sebagai berikut.

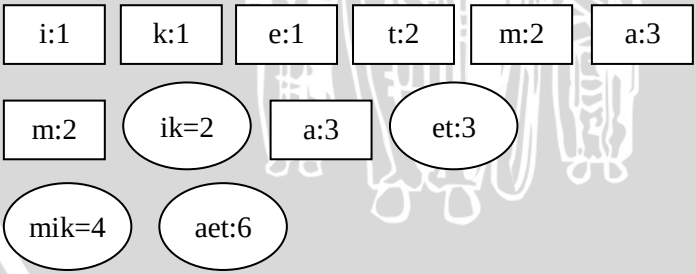
01101101	01100001	01110100	01100101	01101101
m	a	t	e	m
01100001	01110100	01101001	01101011	01100001
a	t	i	k	a

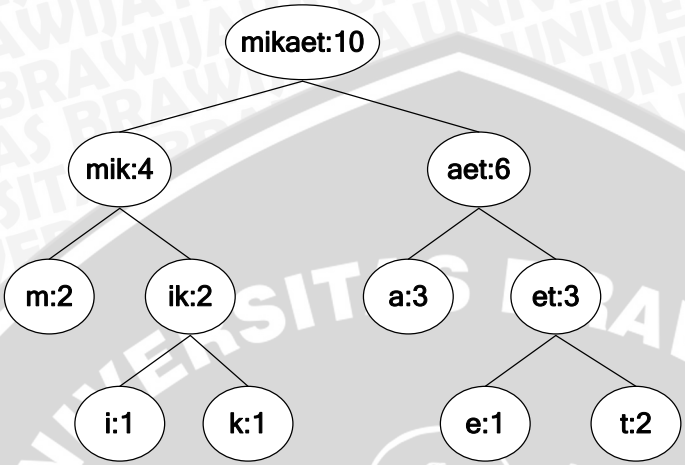
Akan dilakukan kompresi terhadap kalimat tersebut. Pertama, kompresi akan dilakukan menggunakan metode *Huffman*. Langkah awal yang dilakukan adalah menghitung frekuensi kemunculan tiap karakter. Frekuensi kemunculan tiap karakter dimasukkan pada tabel kompresi awal seperti ditunjukkan pada tabel 3.1.

Tabel 3.1 Kompresi awal

Karakter	Frekuensi
m	2
a	3
t	2
e	1
i	1
k	1

Langkah-langkah pembentukan pohon *Huffman* ditunjukkan pada gambar 3.10.





Gambar 3.10 Langkah-langkah pembentukan pohon *Huffman*

Dari pohon *Huffman* yang terdapat pada gambar 3.7 diperoleh kode *Huffman* untuk ditambahkan pada tabel kompresi seperti yang ditunjukkan pada tabel 3.2.

Dari kode *Huffman* yang didapatkan dari tabel 3.2, kalimat "matematika" direpresentasikan dengan rangkaian bit "**0010111110 001011101001110"**.

Dengan kompresi menggunakan metode *Huffman* ini, dibutuhkan 26 bit untuk menyimpan hasil kompresi kalimat "matematika".

Tabel 3.2 Kompresi yang sudah dilengkapi kode *Huffman*

Karakter	frekuensi	Kode huffman
m	2	00
i	3	010
k	2	011
a	1	10
e	1	110
t	2	111

Untuk contoh yang menggunakan *multi core* prosesnya sama, yang membedakan untuk *multi core* yaitu, *file* dipecah untuk didistibusikan kesejumlah *core* yang tersedia.

Struktur Data dari pohon *Huffman* ditunjukkan pada Tabel 3.3 Struktur Data pohon *Huffman*.

Tabel 3.3 Struktur Data pohon *Huffman*

List<Cabang> pohonHuffman
class Cabang
public ID
public ParentID
public Pertengahan
public Kiri
public Kanan
public NilaiByte
public NilaiBit
public JumlahBit

Setelah didapatkan sturktur data dari pohon *Huffman* seperti yang ditunjukkan pada Tabel 3.3 Struktur Data pohon *Huffman,* selanjutnya dibuat struktur *file* dari proses kompresi menggunakan *Huffman code*.

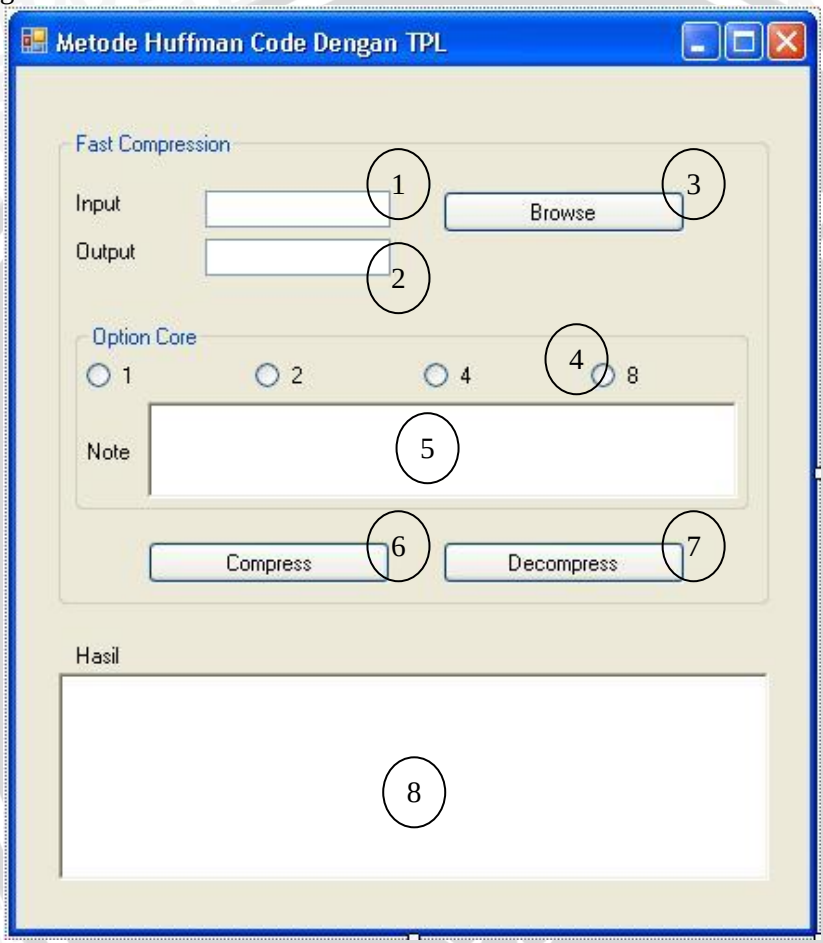
Struktur *file* yang digunakan terdiri dari `UkuranAsli` yaitu ukuran *file* yang akan diproses, `UkuranKompresi` yaitu ukuran *file* yang telah diproses, fungsi `Kompresi` yaitu fungsi yang melakukan proses kompresi, fungsi `Dekompresi` yaitu fungsi yang melakukan proses `dekompresi`, dan `Simpan` berfungsi untuk menyimpan *file* hasil kompresi. Keseluruhan fungsi ditunjukkan pada Tabel 3.4 Struktur *File*.

Tabel 3.4 Struktur *File*

public MultiFile
public UkuranAsli
public UkuranKompresi
public void Kompresi(string lokasiFile)
public void Dekompresi(string namaFile)
public void Simpan(string lokasiSimpan)

3.6 Rancangan *User Interface*

Rancangan *user interface* untuk aplikasi kompresi *file* menggunakan metode *Huffman* dengan teknik TPL ditunjukkan pada gambar 3.11.



Gambar 3.11 Rancangan *user interface* aplikasi

Adapun bagian-bagian dari *user interface* tersebut adalah :

1. *Textbox* untuk menampilkan lokasi *file* input yang akan di kompresi.
2. *Textbox* untuk menampilkan lokasi *file* output yang telah kompresi.
3. *Button* untuk mangambil *file* input yang akan dikompresi.

4. *RadioButton* untuk memilih berapa *core* yang digunakan untuk mengkompresi.
5. *RichTextBox* untuk menampilkan prosesor yang digunakan.
6. *Button* untuk melakukan kompresi.
7. *Button* untuk melakukan dekompresi.
8. *RichTextBox* untuk menampilkan waktu kompresi.

3.7 Rancangan Ujicoba dan Evaluasi Hasil

Ujicoba terhadap *file* yang dilakukan proses kompresi dan dekompresi diperlukan untuk mengetahui apakah aplikasi *Task Parallel Library* dan metode *Huffman* ini mampu melakukan kompresi dan dekompresi dengan efektif.

3.7.1 Rancangan Ujicoba

Pengujian yang akan dilakukan adalah menginputkan *file* dokumen pada aplikasi. Jumlah *file* yang dinputkan sebanyak 3 buah dengan *file type* yang berbeda. Ukuran dari ketiga *file* tersebut dibuat bervariasi. Hal ini bertujuan untuk mengetahui bagaimana pengaruh kecepatan terhadap hasil kompresi.

3.7.2 Analisa dan Evaluasi Hasil

Output yang dihasilkan oleh proses kompresi akan dianalisa dengan memperhatikan ukuran *file* hasil kompresi, waktu yang diperlukan untuk melakukan kompresi. Sedangkan *output* yang dihasilkan oleh proses dekompresi akan dianalisa dengan memperhatikan kesesuaian antara *file* hasil dekompresi dengan *file* sumber dan waktu yang diperlukan untuk melakukan proses dekompresi. Rancangan tabel untuk hasil ujicoba akan ditunjukkan pada tabel 3.5 dan 3.6.

Tabel 3.5 Rancangan hasil uji coba proses kompresi

No	Ukuran		Waktu (ms)		Jumlah Core
	File	File setelah kompresi	Tidak Menggunakan TPL	Menggunakan TPL	

Tabel 3.6 Rancangan hasil ujicoba proses dekompresi

No	Ukuran		Waktu (ms)		Jumlah Core
	File	File setelah dikompresi	Tidak Menggunakan TPL	Menggunakan TPL	

BAB IV
IMPLEMENTASI DAN PEMBAHASAN

Pada bab ini menjelaskan mengenai implementasi sistem dan pembahasan terhadap hasil uji coba. Setelah dilakukan uji coba terhadap sistem selanjutnya dapat dilakukan evaluasi dan analisa terhadap sistem yang telah dibuat.

4.1 Lingkungan Implementasi

Lingkungan implementasi meliputi lingkungan perangkat keras serta lingkungan perangkat lunak.

4.1.1 Lingkungan Perangkat Keras

Perangkat keras yang digunakan dalam pengembangan sistem ini adalah :

1. Prosesor Intel Core 2 Duo 2.4 GHz.
2. RAM 4 Gb DDR 3.
3. *Harddisk* 350 GB.

4.1.2 Lingkungan Perangkat Lunak

Perangkat lunak yang digunakan dalam pengembangan sistem pengkategori berita ini adalah :

1. Sistem operasi Microsoft Windows 7.
2. Microsoft Visual C# 2010 Express Edition.

4.2 Implementasi Program

Berdasarkan analisa dan rancangan sistem pada bab 3, maka akan dilakukan implementasi dari perancangan tersebut.

4.2.1 Implementasi Kompresi

Pada *Sourcecode* 4.1 Kompresi *Huffman*, `var path` berfungsi untuk mengambil lokasi *file* sementara, kemudian `var TreeOptimal` memanggil fungsi `TulisTree` yang berguna untuk membuat tabel frekuensi dan menulis *tree* kedalam *cache file* sementara. `TulisDataTerkompresi` berfungsi menerjemahkan *byte* kedalam blok-blok yang sudah terkompresi menggunakan *tree* yang dibentuk sebelumnya.

```
public byte[] Kompresi()
      {
            var path = Path.GetTempFileName();
```

```
            var cacheOutput = new FileStream(path,
                FileMode.Create);
            var TreeOptimal = TulisTree(cacheOutput);
            TulisDataTerkompresi(ref cacheOutput,
                TreeOptimal);
            cacheOutput.Flush(true);
            cacheOutput.Close();
            cacheOutput.Dispose();
            var hasil = File.ReadAllBytes(path);
            File.Delete(path);
            return hasil;
        }
```

Sourcecode 4.1 Kompresi *Huffman*

Dalam membentuk *tree*, maka yang pertama dilakukan adalah membuat fungsi menyusun pohon dari tabel frekuensi disebut dengan fungsi `BangunPohon()` yang dijelaskan pada *Sourcecode* 4.2 Bangun Pohon. Setelah didapatkan pembentukan pohon, maka selanjutnya membentuk *tree* dari tiap cabang pada pohon tersebut disebut dengan `SusunTreeDariTiapCabang()` yang dijelasan pada *Sourcecode* 4.3 SusunTreeDariTiapCabang.

```
    private bool BangunPohon()
       {
            int iParentIndex = 0;
            var TreeOptimal = pohonHuffman.Where(x =>
                x.Frekuensi > 0).ToList();
            TreeOptimal = TreeOptimal.OrderBy(x =>
                x.Frekuensi).ToList();
            var TreeSusun = new List<Cabang>(TreeOptimal);
            while (TreeSusun.Count > 1)
            {
                TreeSusun = TreeSusun.OrderBy(x =>
                    x.Frekuensi).ToList();
                var orangTua = new Cabang() { Frekuensi =
                    TreeSusun[0].Frekuensi +
                    TreeSusun[1].Frekuensi, Pertengahan =
                    true };
                pohonHuffman.Add(orangTua);
                iParentIndex =
                    pohonHuffman.FindIndex(delegate(Cabang
                    L1) { return L1.Equals(TreeSusun[0]); });
                pohonHuffman[iParentIndex].Kiri = true;
                pohonHuffman[iParentIndex].ParentID =
                    orangTua.ID;
                iParentIndex =
                    pohonHuffman.FindIndex(delegate(Cabang
                    L1) { return L1.Equals(TreeSusun[1]); });
                pohonHuffman[iParentIndex].Kanan = true;
                pohonHuffman[iParentIndex].ParentID =
                    orangTua.ID;
                TreeOptimal = new List<Cabang>(pohonHuffman);
```

```
            TreeOptimal.RemoveAll(delegate(Cabang leaf) {
                return leaf.Frekuensi == 0; });
            TreeOptimal.Sort(delegate(Cabang L1, Cabang
                L2) { return L1.Frekuensi.CompareTo
               (L2.Frekuensi); });
            TreeSusun = new List<Cabang>(TreeOptimal);
            TreeSusun.RemoveAll(delegate(Cabang leaf) {
               return leaf.ParentID != new Guid(); });
        }
        return true;
    }
```

Sourcecode 4.2 Bangun Pohon

Menyusun *tree* dari nilai-nilai tiap cabang. Cabang ParentNode berfungsi untuk menemukan *parent* berikutnya yang akan diproses untuk dibuat cabang baru.

```
private bool SusunTreeDariTiapCabang()
    {
        StringBuilder sbOutput = new StringBuilder();
        int iBinaryValue = 0;
        int iBitCount = 0;
        int iLeadingZeros = 0;
        foreach (Cabang Node in pohonHuffman)
        {
            if (!Node.Pertengahan && Node.Frekuensi != 0)
            {
                iBinaryValue = 0;
                iBitCount = 0;
                iLeadingZeros = 0;
                if (Node.Kiri || Node.Kanan)
                {
                    if (Node.Kiri) iBitCount++;
                    else if (Node.Kanan)
                    {
                        iBinaryValue ^= ((int)1 <<
                        iBitCount);
                        iBitCount++;
                    }
        Cabang ParentNode =
            pohonHuffman.Find(delegate(Cabang leaf) {
            return leaf.ID == Node.ParentID; });
                    while (ParentNode.ParentID != new
                          Guid())
                    {
                        if (ParentNode.Kiri) iBitCount++;
                        else if (ParentNode.Kanan)
                        {
                            iBinaryValue ^= ((int)1 <<
                            iBitCount);
                            iBitCount++;
    ParentNode = pohonHuffman.Find(delegate(Cabang leaf) {
          return leaf.ID == ParentNode.ParentID; });
                    }
```

```
                if (iBinaryValue != 0) iLeadingZeros =
                    iBitCount - Convert.ToString
                    (iBinaryValue, 2).Length;
                else iLeadingZeros = iBitCount;
                Node.NilaiBit = iBinaryValue;
                Node.JumlahBit = iBitCount;
                if (Node.NilaiBit == 0)
                {
                    Node.NilaiBit = 1;
                    Node.JumlahBit++;
                }
            }
            return true;
```

Sourcecode 4.3 `SusunTreeDariTiapCabang`

Kemudian dilakukan pembentukan *tree* disebut dengan fungsi `TulisTree` yang dijelaskan pada *Sourcecode* 4.4 TulisTree.

```
    private List<Cabang> TulisTree(FileStream msEncodedOutput)
        {
            BangunPohon();
            SusunTreeDariTiapCabang();
            var TreeOptimal = pohonHuffman.Where(cabang =>
                !cabang.Pertengahan).ToList();
            long buffer;
            var jumlahByteTertulis = 0;
            MemoryStream awalanTree = new MemoryStream();
            foreach (Cabang l in TreeOptimal)
            {
                if (!l.Pertengahan & l.Frekuensi > 0)
                {
                    buffer = l.NilaiByte;
                    buffer <<= 8;
                    buffer ^= l.JumlahBit;
                    buffer <<= 48;
                    buffer ^= l.NilaiBit;
                            awalanTree.Write
                            BitConverter.GetBytes(buffer),0,8);
                    jumlahByteTertulis++;
                    buffer = 0;
                }
            }
                            msEncodedOutput.Write
                           (BitConverter.GetBytes
                           (cacheInput.Length), 0, 8);
          msEncodedOutput.WriteByte
          ((byte)(jumlahByteTertulis - 1));
            msEncodedOutput.Write(awalanTree.ToArray(), 0,
               Convert.ToInt16(awalanTree.Length));
        msEncodedOutput.Write(new byte[] { 66, 67, 68 },0, 3);
        return TreeOptimal;
```

Sourcecode 4.4 `TulisTree`

TulisDataTerkompresi berfungsi untuk memasukan *tree* ke dalam *stream*. `if (JumlahParalel>1)` bekerja paralel apabila jumlah *core* lebih dari satu, `else` dikerjakan secara serial.

```
private void TulisDataTerkompresi(ref FileStream
  cacheOutput, List<Cabang> TreeOptimal)
    {
        var ukuran = (int)cacheInput.Length;
        var data = new byte[ukuran];
        var sebelum = cacheInput.Position;
        cacheInput.Position = 0;
        cacheInput.Read(data, 0, ukuran);
        cacheInput.Position = sebelum;
        var sisa = data.Length % UkuranBlok;
        var jumlahBlok = data.Length / UkuranBlok + (sisa
            > 0 ? 1 : 0);
        var arrayTemporer = new MemoryStream[jumlahBlok];
        Parallel.For(0, jumlahBlok, new ParallelOptions()
            { MaxDegreeOfParallelism = jumlahParalel },
              blokKe =>
            {
                var posisiBlok = blokKe * UkuranBlok;
                arrayTemporer[blokKe] =
              TulisBlok(TreeOptimal, data, UkuranBlok,
               posisiBlok);
            });
        foreach (var temporer in arrayTemporer)
        {
            temporer.Position = 0;
            temporer.CopyTo(cacheOutput);
```

Sourcecode 4.5 `TulisDataTerkompresi`

`TulisBlok` berfungsi untuk menulis tiap-tiap blok yang akan dikompresi. `Tempporer.Write` berfungsi untuk memberi indek hasil kompresi pada tiap-tiap blok.

```
private static MemoryStream TulisBlok(List<Cabang>
    OptimizedTree, byte[] data, int ukuranBlok, int
    awalBlok)
    {
        var temporer = new MemoryStream();
        temporer.Write(BitConverter.GetBytes(0L), 0, 8);
        long buffer = 0;
        int jumlahBuffer = 0;
        foreach (byte b in
              data.Skip(awalBlok).Take(ukuranBlok))
        {
            Cabang CabangYangDipakai = OptimizedTree[b];
            jumlahBuffer += CabangYangDipakai.JumlahBit;
            if (buffer != 0)
            {
                buffer <<= CabangYangDipakai.JumlahBit;
```

```
                    buffer ^= CabangYangDipakai.NilaiBit;
                }
                else
                {
                    buffer = CabangYangDipakai.NilaiBit;
                }
                while (jumlahBuffer > 7)
                {
                    int iBufferOutput = (int)(buffer >>
                    (jumlahBuffer - 8));
                    temporer.WriteByte((byte)iBufferOutput);
                    jumlahBuffer = jumlahBuffer - 8;
                    iBufferOutput <<= jumlahBuffer;
                    buffer ^= iBufferOutput;
                }
            if (jumlahBuffer > 0)
            {
                buffer = buffer << (8 - jumlahBuffer);
                temporer.WriteByte((byte)buffer);
            }
            temporer.Position = 0;temporer.Write
                (BitConverter.GetBytes
                (temporer.Length), 0, 8);
            return temporer;
```

Sourcecode 4.6 `TulisBlok`

4.2.2 Implementasi Proses Dekompresi

Proses ini membaca tree dari *file* yang terkompresi, kemudian fungsi `var offsets = CariOffset` berguna untuk memberikan index pada tiap-tiap blok yang telah terkompresi.

```
   public byte[] Dekompresi(byte[] bInput)
        {

            List<Cabang> Tree = new List<Cabang>(255);
            long buffer = 0;
            long panjangStream = 0;
            int jumlahEntryTree = 0;
            int posisiAkhirDataTree = 0;
            byte[] bDecodedOutput = BacaTree(bInput, Tree,
ref buffer, ref panjangStream, ref jumlahEntryTree, ref
posisiAkhirDataTree);
            var key = Tree.ToDictionary(x => x, x =>
                x.NilaiBit * 100000 + x.JumlahBit);
            Tree = key.Values.Distinct().Select(x =>
                key.First(y => y.Value == x).Key).ToList();
            var DTree = Tree.ToDictionary(x => x.NilaiBit *
                100000 + x.JumlahBit, x => x.NilaiByte);
            IndexBitValue = Tree.Select(x =>
                x.NilaiBit).ToList();
            buffer = 0;
            var offsets = CariOffset(bInput,
                posisiAkhirDataTree).ToList();
```

```
            if (JumlahParalel > 1)
            {
                Parallel.For(0, offsets.Count, new
                  ParallelOptions()
                  { MaxDegreeOfParallelism = JumlahParalel },
                    i =>
                    {
                        DekompresiBlok((int)offsets[i],
                        bInput, DTree, posisiAkhirDataTree,
                        UkuranBlok * i, bDecodedOutput);
                    });
            }
            else
            {
                for (var i = 0; i < offsets.Count; i++)
                {
                    DekompresiBlok((int)offsets[i], bInput,
                    DTree, posisiAkhirDataTree, UkuranBlok *
                    i, bDecodedOutput);
                }}
            return bDecodedOutput;
```

Sourcecode 4.8 `Dekompresi`

`BacaTree` di sini menyiapkan 256 cabang yang kemudian diisi dengan nillai-nilai tiap cabang dari *tree* yang sudah dikompresi.

```
  private static byte[] BacaTree(byte[] bufferInput,
List<Cabang> Tree, ref long buffer, ref long panjangData, ref
int jumlahTree, ref int posisiAkhirDataTree)
        {
            for (int i = 0; i < 256; i++)
            {
                Tree.Add(new Cabang());
            }
         panjangData = BitConverter.ToInt64(bufferInput, 0);
         byte[] bDecodedOutput = new byte[panjangData];
         jumlahTree = bufferInput[8];
         posisiAkhirDataTree = (((jumlahTree + 1) * 8) + 8);
            for (int i = 9; i <= posisiAkhirDataTree; i += 8)
            {
               buffer = BitConverter.ToInt64(bufferInput, i);
               var cabang = new Cabang();
               cabang.NilaiByte = (byte)(buffer >> 56);
               if (cabang.NilaiByte != 0) buffer ^=
                  (((Int64)cabang.NilaiByte) << 56);
                cabang.JumlahBit = (int)(buffer >> 48);
                buffer ^= (((Int64)cabang.JumlahBit) << 48);
                cabang.NilaiBit = (int)(buffer);
                Tree[cabang.NilaiByte] = cabang;
            }
            return bDecodedOutput;
```

Sourcecode 4.9 `BacaTree`

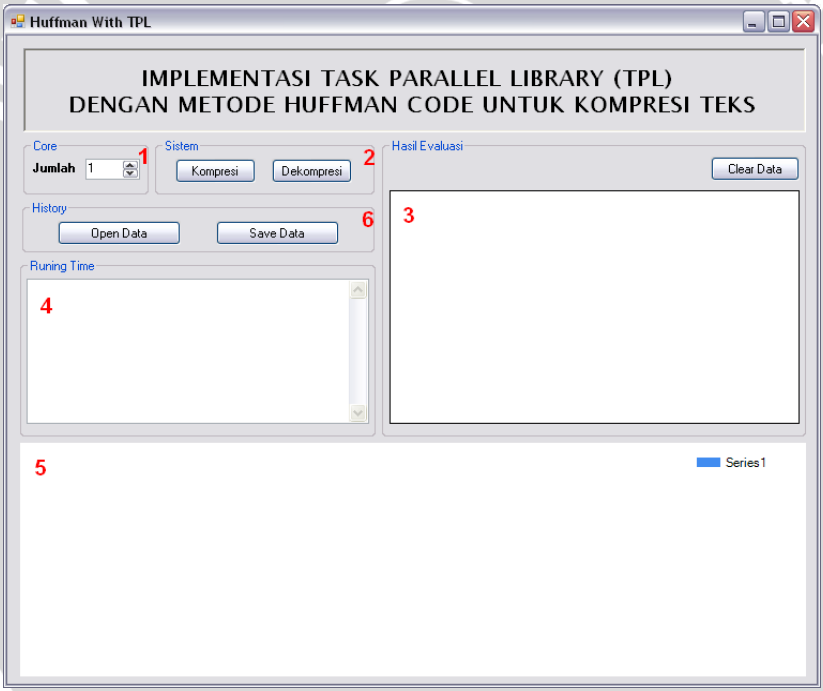
Menerjemahkan blok-blok yang sudah terkompresi kedata asli dengan tujuan agar blok-blok yang terpisah menjadi satu blok.

```
      private static void DekompresiBlok(int offsetInput,
byte[] bInput, Dictionary<int, byte> DTree, int
posisiAkhirTree, int offsetOutput, byte[] hasilDekompresi)
        {
            int iOutputBuffer = 0;
            int iInputBufferSize = 0;
            long iInputBuffer = 0;
            var posAwal = posisiAkhirTree + 4 + offsetInput;
            if (posAwal == bInput.Length)
            {
                return;
            }
            var ukuranBlok = BitConverter.ToInt64(bInput,
                posAwal);
            for (int i = (posAwal + 8); i < posAwal +
                ukuranBlok; i++)
            {
                iInputBufferSize += 8;
                if (iInputBuffer != 0)
                {
                    iInputBuffer <<= 8;
                    iInputBuffer ^= bInput[i];
                }
                else
                {
                    iInputBuffer = bInput[i];
                }
                for (int j = (iInputBufferSize - 1); j >= 0;
                    j--)
                {
                    iOutputBuffer = (int)(iInputBuffer >> j);
                    if (iOutputBuffer == 0) continue;
                    int iBitCount = iInputBufferSize - j;
                    var iOB = iOutputBuffer;
                    if (DTree.ContainsKey(iOB * 100000 +
                       iBitCount))
                    {
                        var kunci = DTree[iOB * 100000 +
                        iBitCount];
                        hasilDekompresi[offsetOutput] =
                          kunci;
                        iOutputBuffer <<= j;
                        iInputBuffer ^= iOutputBuffer;
                        iInputBufferSize = j;
                        offsetOutput++;
}
```

Sourcecode 4.10 `DekompresiBlok`

4.3 Implementasi Antarmuka

Sistem yang telah dibuat memiliki antar muka form utama yang dibagi menjadi enam buah bagian. Fungsi-fungsi tersebut dipergunakan dalam melakukan proses pengujian kompresi *file*. Dalam melakukan kompresi hal pertama dilakukan adalah memasukkan *file* yang akan dikompresi, selanjutnya pilih berapa jumlah *core* yang digunakan utuk proses kompresi. Apabila pengguna memilih jumlah *core* sebanyak satu, maka proses kompresi dilakukan tidak menggunakan teknik TPL. Sedangkan apabila pengguna memilih *core* lebih dari satu, maka proses kompresi dilakukan menggunakan tehnik TPL. Deskripsi ini ditunjukkan pada Gambar 4.1 Form Utama.

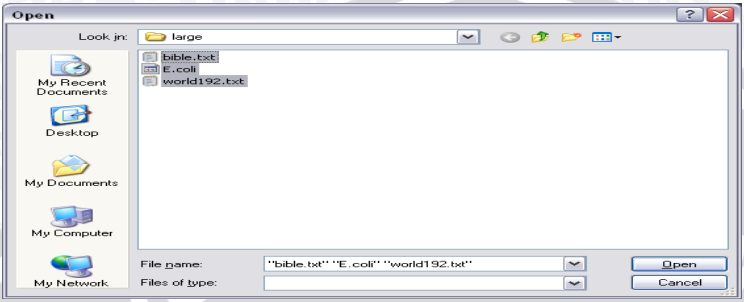


Gambar 4.1 Form Utama

4.3.1 Proses Pengambilan Dokumen

Tombol **Kompresi** digunakan untuk menentukan *file* mana yang akan dilakukan proses kompresi. Jika tombol **Kompresi** ditekan, maka akan muncul *dialog box* seperti yang ditampilkan pada

gambar 4.2. *User* akan melakukan pemilihan *file*, dan kemudian tekan tombol **Open**.



Gambar 4.2 Dialog Box Open

4.3.2 Proses Pemilihan *Core*

User diberi pilihan dengan adanya proses pilihan *core*. Dimana *user* memilih jumlah *core* yang akan digunakan untuk proses kompresi. Jumlah *core* yang dipilih oleh *user* tidak lebih dari *core* lebih dari jumlah *core* yang tersedia di mesin yang digunakan oleh *user*.

Gambar 4.3 Tampilan pilihan *core*

Proses pemilihan *core* oleh *user* ditunjukkan pada Gambar 4.3 Tampilan pilihan *core*.

4.3.3 Proses Kompresi

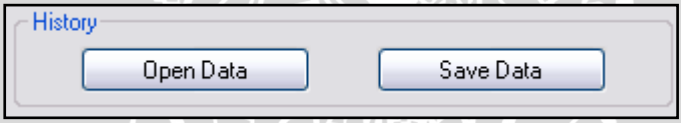
Setelah *user* telah menentukan jumlah *core* yang digunakan, maka proses yang selanjutnya dilakukan adalah melakukan kompresi terhadap *file* yang dipilih pada tahap Gambar 4.2 Dialog Box Open. Proses kompresi dilakukan dengan menekan tombol kompresi seperti yang ditunjukkan pada Gambar 4.4 Proses Kompresi dan proses Dekompresi.

Gambar 4.4 Proses Kompresi dan Proses Dekompresi

Fungsi dekrompresi dapat dijalankan apabila *user* telah menjalankan fungi sudah mengambil data yang sudah dikompresi dengan menekan tombol **Open Data** yang ditunjukkan pada gambar 4.5 Open Data dan Save Data.

4.3.4 Proses Pengambilan dan Penyimpanan Data

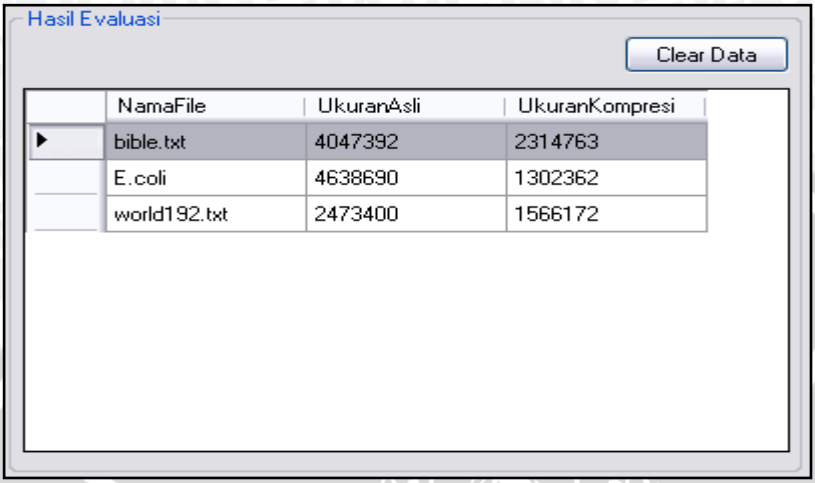
Proses Open Data menjelaskan mengenai pengambilan data yang telah terkompresi dan tersimpan di *drive*. Sedangkan proses Save Data menjelaskan mengenai proses penyimpanan data yang telah terkompresi dan selanjutnya melakuan penyimpanan pada data tersebut pada *drive*. Penjelasan mengenai proses ini ditunjukkan pada Gambar 4.5 Open Data dan Save Data.



Gambar 4.5 Open Data dan Save Data

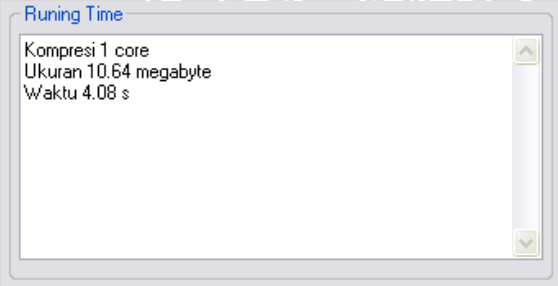
4.3.5 Proses Evaluasi Data

Setelah proses kompresi berhasil, maka ditampilkan hasil dari proses kompresi. Hasil yang ditampikan berisi informasi ukuran sebenarnya dan ukuran hasil kompresi dari *file* yang telah dikompresi. Penjelasan pada tahap ini ditunjukkan pada Gambar 4.6 Evaluasi Hasil Kompresi.



Gambar 4.6 Evaluasi Hasil Kompresi

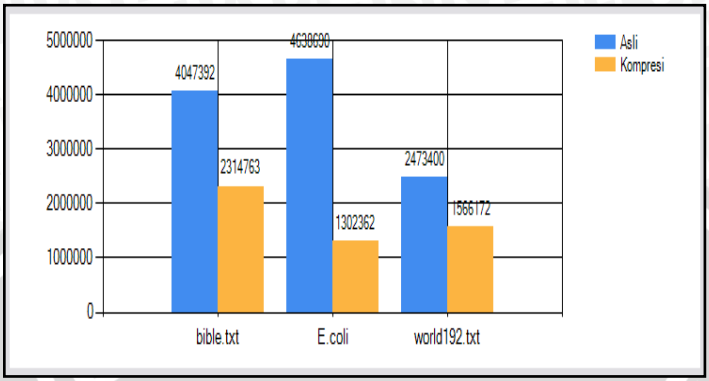
Selain itu juga akan ditampilkan informasi berapa besar *file* yang dikompresi dan berapa lama waktu yang diperlukan untuk melakukan proses kompresi. Proses ini ditampilkan pada Gambar 4.7 Informasi Ukuran *File* dan Waktu.



Gambar 4.7 Informasi Ukuran *File* dan Waktu

4.3.6 Diagram Evaluasi

Proses evaluasi yang ditunjukkan pada Gambar 4.7 Informasi Ukuran *File* dan Waktu kemudian untuk mempermudah pembacaan hasil kompresi, *user* diberikan tampilan *chart*, yang berfungsi menampilkan perbandingan hasil *file* yang sudah dikompresi dengan *file* asli. Proses ini digambarkan dalam bentuk sebuah diagram. seperti yang ditunjukkan pada Gambar 4.8 Diagram Hasil Evaluasi.



Gambar 4.8 Diagram Hasil Evaluasi

4.4 Implementasi Uji Coba

4.4.1 Skenario Evaluasi

Dalam melakukan pengujian sistem pada penelitian ini maka distribusi *file* yang digunakan terdiri dari 3 format *file*. Format *file* yang pergunaan antara lain: doc, .txt dan .pdf.

Evaluasi dilakukan untuk menguji kecepatan sistem kompresi metode *Huffman* menggunakan *task parallel library*. Distribusi *file* yang digunakan dalam penelitian ini ditunjukkan pada Tabel 4.1 Distribusi *File* Pengujian

Tabel 4.1 Distribusi *File* Pengujian

Format *file*	Jumlah
.doc	4
.txt	4
.pdf	4

Berdasarkan Tabel 4.1 Distribusi *File* Pengujian, diketahui bahwa jumlah dokumen percobaan berjumlah masing-masing 4 buah.

Pada penelitian ini dilakuan 2 macam percobaan, percobaan pertama kompresi metode *Huffman* tanpa *task parallel library* dan percobaan kedua kompresi metode *Huffman* dengan *task parallel library*.

4.5 Hasil Evaluasi

Pengujian dilakukan dengan jumlah *file* ujicoba yang berbeda yakni .doc, .txt dan .pdf yang nantinya akan menghasilkan ukuran dan waktu yang berbeda. Ada dua jenis ujicoba yaitu kompresi

huffman tanpa *task parallel library* dan kompresi *huffman* dengan *task parallel library*.

4.5.1 Hasil Kompresi Dengan *File* Uji .doc

Hasil uji coba terhadap sistem yang telah dibuat dilakukan untuk mengetahui kinerja dari sistem. Evaluasi dilakukan dengan cara membandingkan hasil kompresi metode *Huffman* tanpa *task parallel library* dan percobaan kedua menggunakan metode *Huffman* menggunakan *task parallel library*. Hasil kompresi metode *Huffman* tanpa *task parallel library* dapat dilihat pada tabel 4.2 Hasil Kompresi *Huffman* tanpa TPL *File* Uji .doc.

Tabel 4.2 Hasil Kompresi *Huffman* tanpa TPL *File* Uji .doc

Nama File	Ukuran File Sumber (mb)	Ukuran File Hasil (mb)	Core	Waktu Kompresi (s)	Rasio Kompresi
file1	21.74	13.60	1	4.50	37.44
file2	43.46	27.27	1	8.82	37.25
file3	86.90	55.02	1	17.95	36.68
file4	173.77	110.81	1	37.92	36.23

Berdasarkan hasil evaluasi pada Tabel 4.2 Hasil Kompresi *Huffman* tanpa TPL, menunjukkan bahwa rasio kompresi terendah dimiliki oleh file4 sebesar 36.23% dengan ukuran *file* 173.77mb dan besar *file* yang didapatkan setelah proses kompresi sebesar 110.81mb, waktu kompresi selama 37.92 detik. Sedangkan rasio kompresi tertinggi dimiliki oleh file1 sebesar 37.44% dengan ukuran *file* 21.74mb dan besar *file* yang didapatkan setelah proses kompresi sebesar 13.60mb dengan waktu kompresi selama 4.50 detik.

Tabel 4.3 Hasil Kompresi *Huffman* dengan TPL *File* Uji .doc

Nama File	Ukuran File Sumber (mb)	Ukuran File Hasil (mb)	Core	Waktu Kompresi (s)	Rasio Kompresi
file1	21.74	13.60	2	3.58	37.44
file2	43.46	27.27	2	5.26	37.25
file3	86.90	55.02	2	10.73	36.68
file4	173.77	110.81	2	20.89	36.23

Berdasarkan hasil evaluasi pada Tabel 4.3 Hasil Kompresi *Huffman* dengan TPL, menunjukkan bahwa rasio kompresi terendah dimiliki oleh file4 sebesar 36.23% dengan ukuran *file* 173.77 mb dan besar *file* yang didapatkan setelah proses kompresi sebesar 110.81 mb, waktu kompresi selama 20.89 detik. Sedangkan rasio kompresi tertinggi dimiliki oleh file1 sebesar 37.44% dengan ukuran *file* 21.74 mb dan besar *file* yang didapatkan setelah proses kompresi sebesar 13.60 mb dengan waktu kompresi selama 3.58 detik.

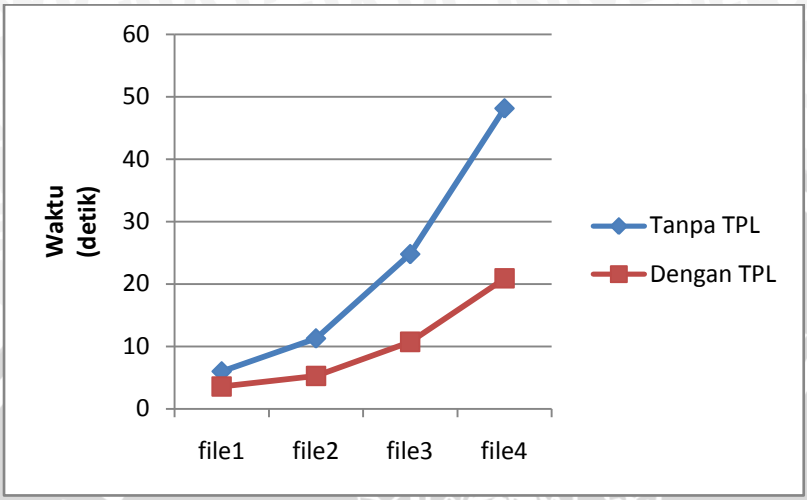
Berdasarkan hasil evaluasi yang telah dilakukan kompresi metode *Huffman* menggunakan *task parallel library* membuktikan bahwa, dengan *file* yang sama waktu kompresi yang didapatkan lebih cepat dibandingkan dengan kompresi metode *Huffman* tanpa *task parallel library*. Hasil ini ditunjukkan pada Tabel 4.4 Perbandingan Waktu Kompresi *File* Uji .doc.

Tabel 4.4 Perbandingan Waktu Kompresi *File* Uji .doc

Nama File	**Ukuran File Sumber (mb)**	**Waktu Kompresi (s)**	
		Tanpa TPL	**Dengan TPL**
file1	21.74	4.50	3.58
file2	43.46	8.82	5.26
file3	86.90	17.95	10.73
file4	173.77	37.92	20.89
Jumlah		**69.19**	**40.46**
Waktu Rata-rata		**17.29**	**10.11**

Berdasarkan hasil yang ditunjukkan Tabel 4.4 Perbandingan Waktu Kompresi menunjukkan bahwa prosentase waktu rata-rata kompresi metode *Huffman* menggunakan *task parallel library* lebih cepat sebesar **41.52%** dibandingkan dengan kompresi metode *Huffman* tanpa *task parallel library*.

Berdasarkan data yang ditunjukkan pada Tabel 4.4 Perbandingan Waktu Kompresi maka dapat dilihat pada Gambar Grafik 4.9 Selisih Waktu Kompresi *File* .doc.



Gambar 4.9 Grafik Selisih Waktu Kompresi *File* .doc

4.5.2 Hasil Kompresi Dengan *File* Uji .txt

Hasil uji coba terhadap sistem yang telah dibuat dilakukan untuk mengetahui kinerja dari sistem. Evaluasi dilakukan dengan cara membandingkan hasil kompresi metode *Huffman* tanpa *task parallel library* dan percobaan kedua menggunakan metode *Huffman* menggunakan *task parallel library*. Hasil kompresi metode *Huffman* tanpa *task parallel library* dapat dilihat pada tabel 4.5 Hasil Kompresi *Huffman* tanpa TPL *File* Uji .txt.

Tabel 4.5 Hasil Kompresi *Huffman* tanpa TPL *File* Uji .txt

Nama File	Ukuran File Sumber (mb)	Ukuran File Hasil (mb)	Core	Waktu Kompresi (s)	Rasio Kompresi
file1	14.66	9.34	1	3.65	36.28
file2	29.31	18.67	1	5.28	36.30
file3	58.63	37.34	1	10.79	36.31
file4	117.26	74.68	1	24.72	36.32

Berdasarkan hasil evaluasi pada Tabel 4.5 Hasil Kompresi *Huffman* tanpa TPL, menunjukkan bahwa rasio kompresi terendah dimiliki oleh file1 sebesar 36.28% dengan ukuran *file* 14.66 mb dan besar *file* yang didapatkan setelah proses kompresi sebesar 9.34 mb, waktu kompresi selama 2.65 detik. Sedangkan rasio kompresi

tertinggi dimiliki oleh file4 sebesar 36.32% dengan ukuran *file* 117.26 mb dan besar *file* yang didapatkan setelah proses kompresi sebesar 74.68 mb dengan waktu kompresi selama 24.72 detik.

Tabel 4.6 Hasil Kompresi *Huffman* dengan TPL *File* Uji .txt

Nama File	Ukuran File Sumber (mb)	Ukuran File Hasil (mb)	Core	Waktu Kompresi (s)	Rasio Kompresi
file1	14.66	9.34	2	3.59	36.28
file2	29.31	18.67	2	5.21	36.30
file3	58.63	37.34	2	10.69	36.31
file4	117.26	74.68	2	17.20	36.32

Berdasarkan hasil evaluasi pada Tabel 4.6 Hasil Kompresi *Huffman* dengan TPL, menunjukkan bahwa rasio kompresi terendah dimiliki oleh file1 sebesar 36.28% dengan ukuran *file* 14.66 mb dan besar *file* yang didapatkan setelah proses kompresi sebesar 9.34 mb, waktu kompresi selama 3.59 detik. Sedangkan rasio kompresi tertinggi dimiliki oleh file1 sebesar 36.32% dengan ukuran *file* 117.26 mb dan besar *file* yang didapatkan setelah proses kompresi sebesar 74.68 mb dengan waktu kompresi selama 17.20 detik.

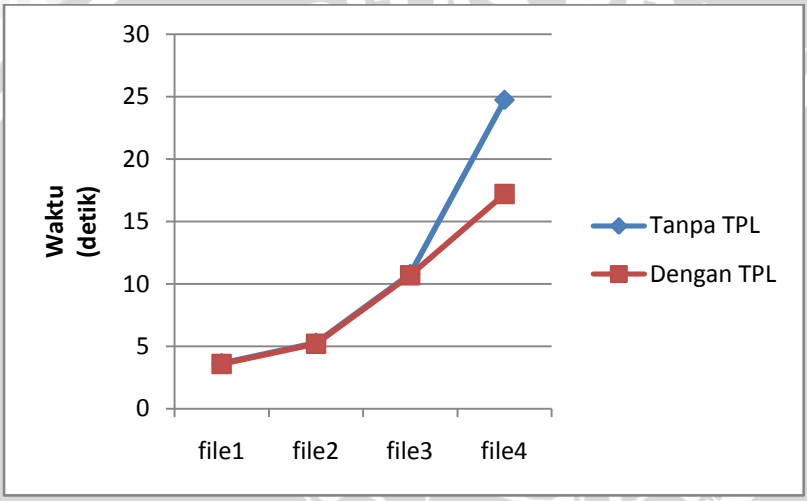
Berdasarkan hasil evaluasi yang telah dilakukan kompresi metode *Huffman* menggunakan *task parallel library* membuktikan bahwa, dengan *file* yang sama waktu kompresi yang didapatkan lebih cepat dibandingkan dengan kompresi metode *Huffman* tanpa *task parallel library*. Hasil ini ditunjukkan pada Tabel 4.7 Perbandingan Waktu Kompresi *File* Uji .txt.

Tabel 4.7 Perbandingan Waktu Kompresi *File* Uji .txt

Nama File	**Ukuran File Sumber (mb)**	**Waktu Kompresi (s)**	
		Tanpa TPL	**Dengan TPL**
file1	14.66	3.65	3.59
file2	29.31	5.28	5.21
file3	58.63	10.79	10.69
file4	117.26	24.72	17.20
Jumlah		**44.44**	**36.66**
Waktu Rata-rata		**11.11**	**9.16**

Berdasarkan hasil yang ditunjukkan Tabel 4.7 Perbandingan Waktu Kompresi menunjukkan bahwa prosentase waktu rata-rata kompresi metode *Huffman* menggunakan *task parallel library* lebih cepat sebesar **17.55%** dibandingkan dengan kompresi metode *Huffman* tanpa *task parallel library*.

Berdasarkan data yang ditunjukkan pada Tabel 4.7 Perbandingan Waktu Kompresi maka dapat dilihat pada Gambar Grafik 4.10 Selisih Waktu Kompresi *File* .txt.



Gambar 4.10 Grafik Selisih Waktu Kompresi *File* .txt.

4.5.3 Hasil Kompresi Dengan *File* Uji .pdf

Hasil uji coba terhadap sistem yang telah dibuat dilakukan untuk mengetahui kinerja dari sistem. Evaluasi dilakukan dengan cara membandingkan hasil kompresi metode *Huffman* tanpa *task parallel library* dan percobaan kedua menggunakan metode *Huffman* menggunakan *task parallel library*. Hasil kompresi metode *Huffman* tanpa *task parallel library* dapat dilihat pada tabel 4.5 Hasil Kompresi *Huffman* tanpa TPL *File* Uji .pdf.

Tabel 4.8 Hasil Kompresi *Huffman* tanpa TPL *File* Uji .pdf

Nama File	Ukuran File Sumber (mb)	Ukuran File Hasil (mb)	Core	Waktu Kompresi (s)	Rasio Kompresi
file1	15.03	14.92	1	2.92	0.73
file2	30.11	29.90	1	6.15	0.69
file3	60.20	59.77	1	12.30	0.71
file4	120.55	119.67	1	25.27	0.72

Berdasarkan hasil evaluasi pada Tabel 4.8 Hasil Kompresi *Huffman* tanpa TPL, menunjukkan bahwa rasio kompresi terendah dimiliki oleh file2 sebesar 0.69% dengan ukuran *file* 30.11 mb dan besar *file* yang didapatkan setelah proses kompresi sebesar 29.90 mb, waktu kompresi selama 6.15 detik. Sedangkan rasio kompresi tertinggi dimiliki oleh file1 sebesar 0.73% dengan ukuran *file* 15.03 mb dan besar *file* yang didapatkan setelah proses kompresi sebesar 14.92 mb dengan waktu kompresi selama 2.92 detik.

Tabel 4.9 Hasil Kompresi *Huffman* dengan TPL *File* uji .pdf

Nama File	Ukuran File Sumber (mb)	Ukuran File Hasil (mb)	Core	Waktu Kompresi (s)	Rasio Kompresi
file1	15.03	14.92	2	2.92	0.73
file2	30.11	29.90	2	4.21	0.69
file3	60.20	59.77	2	8.61	0.71
file4	120.55	119.67	2	15.54	0.72

Berdasarkan hasil evaluasi pada Tabel 4.9 Hasil Kompresi *Huffman* tanpa TPL, menunjukkan bahwa rasio kompresi terendah dimiliki oleh file2 sebesar 0.69% dengan ukuran *file* 30.11 mb dan besar *file* yang didapatkan setelah proses kompresi sebesar 29.90 mb, waktu kompresi selama 4.21 detik. Sedangkan rasio kompresi tertinggi dimiliki oleh file1 sebesar 0.73% dengan ukuran *file* 15.03 mb dan besar *file* yang didapatkan setelah proses kompresi sebesar 14.92 mb dengan waktu kompresi selama 2.92 detik.

Berdasarkan hasil evaluasi yang telah dilakukan kompresi metode *Huffman* menggunakan *task parallel library* membuktikan bahwa, dengan *file* yang sama waktu kompresi yang didapatkan lebih cepat dibandingkan dengan kompresi metode *Huffman* tanpa *task*

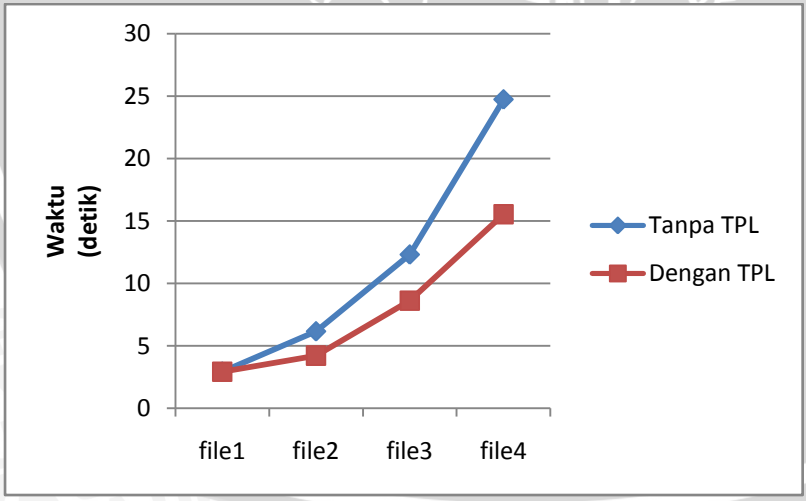
parallel library. Hasil ini ditunjukkan pada Tabel 4.10 Perbandingan Waktu Kompresi *File* Uji .pdf.

Tabel 4.10 Perbandingan Waktu Kompresi *File* Uji .pdf

Nama File	Ukuran File Sumber (mb)	Waktu Dekompresi (s)	
		Tanpa TPL	Dengan TPL
file1	15.03	2.92	2.92
file2	30.11	6.15	4.21
file3	60.20	12.30	8.61
file4	120.55	25.27	15.54
Jumlah		**46.64**	**31.28**
Waktu Rata-rata		**11.66**	**7.82**

Berdasarkan hasil yang ditunjukkan Tabel 4.10 Perbandingan Waktu Kompresi menunjukkan bahwa prosentase waktu rata-rata kompresi metode *Huffman* menggunakan *task parallel library* lebih cepat sebesar **32.93%** dibandingkan dengan kompresi metode *Huffman* tanpa *task parallel library*.

Berdasarkan data yang ditunjukkan pada Tabel 4.10 Perbandingan Waktu Kompresi maka dapat dilihat pada Gambar Grafik 4.11 Selisih Waktu Kompresi *File* .pdf.



Gambar 4.11 Grafik Selisih Waktu Kompresi *File* .pdf.

4.5.4 Hasil Dekompresi Dengan *File* Uji .doc

Evaluasi juga dilakukan dengan cara membandingkan hasil dekompresi metode *Huffman* tanpa *task parallel library* dan hasil dekompresi metode *Huffman* dengan *task parallel library*. Hasil dekompresi metode *Huffman* tanpa *task parallel library* dapat dilihat pada tabel 4.11 Hasil Dekompresi *Huffman* tanpa TPL *File* Uji .doc.

Tabel 4.11 Hasil Dekompresi *Huffman* tanpa TPL *File* Uji .doc

Nama File	Ukuran File Sumber (mb)	Ukuran File Hasil (mb)	Core	Waktu Dekompresi (s)	Akurasi (%)
file1	21.74	21.74	1	6.00	100
file2	43.46	43.46	1	11.28	100
file3	86.90	86.90	1	24.78	100
file4	173.77	173.77	1	48.12	100

Berdasarkan hasil evaluasi pada Tabel Tabel 4.11 Hasil Dekompresi *Huffman* tanpa TPL, menunjukkan bahwa waktu dekompresi terendah dimiliki oleh file1 sebesar 6.00 detik dengan ukuran *file* 21.74 mb dan besar *file* yang didapatkan setelah proses dekompresi sebesar 21.74 mb. Sedangkan waktu dekompresi tertinggi dimiliki oleh file4 sebesar 48.12 detik dengan ukuran *file* 173.77 mb dan besar *file* yang didapatkan setelah proses dekompresi sebesar 173.77 mb.

Tabel 4.12 Hasil Dekompresi *Huffman* dengan TPL *File* Uji.doc

Nama File	Ukuran File Sumber (mb)	Ukuran File Hasil (mb)	Core	Waktu Dekompresi (s)	Akurasi (%)
file1	21.74	21.74	2	3.58	100
file2	43.46	43.46	2	5.26	100
file3	86.90	86.90	2	10.73	100
file4	173.77	173.77	2	20.89	100

Berdasarkan hasil evaluasi pada Tabel 4.12 Hasil Dekompresi *Huffman* dengan TPL, menunjukkan menunjukkan bahwa waktu dekompresi terendah dimiliki oleh file1 sebesar 3.58 detik dengan ukuran *file* 21.74 mb dan besar *file* yang didapatkan setelah proses kompresi sebesar 21.74 mb. Sedangkan waktu dekompresi tertinggi

dimiliki oleh file4 sebesar 20.89 detik dengan ukuran *file* 173.77 mb dan besar *file* yang didapatkan setelah proses dekompresi sebesar 173.77 mb.

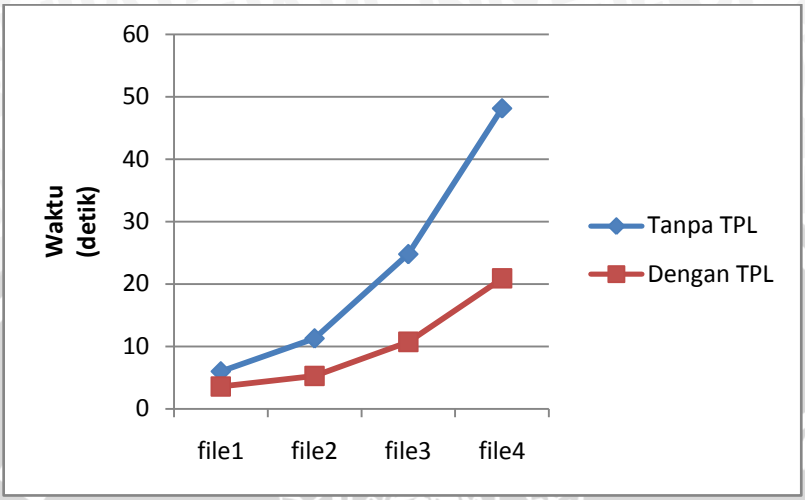
Berdasarkan hasil evaluasi yang telah dilakukan, dekompresi metode *Huffman* dengan *task parallel library*, membuktikan bahwa dengan *file* yang sama waktu dekompresi yang didapatkan lebih cepat dibandingkan dengan dekompresi metode *Huffman* tanpa *task parallel library*. Hasil ini ditunjukkan pada Tabel 4.13 Perbandingan Waktu Dekompresi *File* Uji .doc.

Tabel 4.13 Perbandingan Waktu Dekompresi *File* Uji .doc

Nama File	**Ukuran File Sumber (mb)**	**Waktu Dekompresi (s)**	
		Tanpa TPL	**Dengan TPL**
file1	21.74	6.00	3.58
file2	43.46	11.28	5.26
file3	86.90	24.78	10.73
file4	173.77	48.12	20.89
Jumlah		**90.18**	**40.46**
Waktu Rata-rata		**22.54**	**10.11**

Berdasarkan hasil yang ditunjukkan Tabel 4.13 Perbandingan Waktu Dekompresi menunjukkan bahwa prosentase waktu rata-rata dekompresi metode *Huffman* menggunakan *task parallel library* lebih cepat sebesar 55.14% dibandingkan dengan dekompresi metode *Huffman* tanpa *task parallel library*.

Berdasarkan data yang ditunjukkan pada Tabel 4.13 Perbandingan Waktu Dekompresi maka dapat dilihat pada Gambar Grafik 4.12 Selisih Waktu Dekompresi *File* .doc.



Gambar 4.12 Grafik Selisih Waktu Dekompresi *File* Uji .doc

4.5.5 Hasil Dekompresi Dengan *File* Uji .txt

Evaluasi juga dilakukan dengan cara membandingkan hasil dekompresi metode *Huffman* tanpa *task parallel library* dan hasil dekompresi metode *Huffman* dengan *task parallel library*. Hasil dekompresi metode *Huffman* tanpa *task parallel library* dapat dilihat pada tabel 4.14 Hasil Dekompresi *Huffman* tanpa TPL *File* Uji .txt.

Tabel 4.14 Hasil Dekompresi *Huffman* tanpa TPL *File* Uji .txt

Nama File	Ukuran File Sumber (mb)	Ukuran File Hasil (mb)	Core	Waktu Dekompresi (s)	Akurasi (%)
file1	14.66	14.66	1	3.81	100
file2	29.31	29.31	1	7.80	100
file3	58.63	58.63	1	15.72	100
file4	117.26	117.26	1	31.35	100

Berdasarkan hasil evaluasi pada Tabel Tabel 4.14 Hasil Dekompresi *Huffman* tanpa TPL, menunjukkan bahwa waktu dekompresi terendah dimiliki oleh file1 sebesar 3.810 detik dengan ukuran *file* 14.66 mb dan besar *file* yang didapatkan setelah proses dekompresi sebesar 14.66 mb. Sedangkan waktu dekompresi tertinggi dimiliki oleh file4 sebesar 31.35 detik dengan ukuran *file*

117.26 mb dan besar *file* yang didapatkan setelah proses dekompresi sebesar 117.26 mb.

Tabel 4.15 Hasil Dekompresi *Huffman* dengan TPL *File* Uji.txt

Nama File	Ukuran File Sumber (mb)	Ukuran File Hasil (mb)	Core	Waktu Dekompresi (s)	Akurasi (%)
file1	14.66	14.66	2	3.81	100
file2	29.31	29.31	2	5.57	100
file3	58.63	58.63	2	10.69	100
file4	117.26	117.26	2	17.20	100

Berdasarkan hasil evaluasi pada Tabel 4.15 Hasil Dekompresi *Huffman* dengan TPL, menunjukkan menunjukkan bahwa waktu dekompresi terendah dimiliki oleh file1 selama 3.81 detik dengan ukuran *file* 14.66 mb dan besar *file* yang didapatkan setelah proses kompresi sebesar 14.66 mb. Sedangkan waktu dekompresi tertinggi dimiliki oleh file4 selama 17.20 detik dengan ukuran *file* 117.26 mb dan besar *file* yang didapatkan setelah proses dekompresi sebesar 117.26 mb.

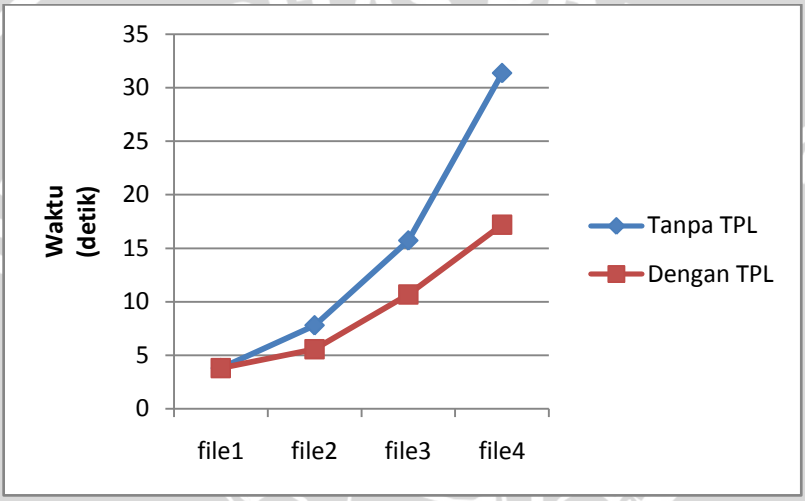
Berdasarkan hasil evaluasi yang telah dilakukan, dekompresi metode *Huffman* dengan *task parallel library*, membuktikan bahwa dengan *file* yang sama waktu dekompresi yang didapatkan lebih cepat dibandingkan dengan dekompresi metode *Huffman* tanpa *task parallel library*. Hasil ini ditunjukkan pada Tabel 4.16 Perbandingan Waktu Dekompresi *File* Uji .txt.

Tabel 4.16 Perbandingan Waktu Dekompresi *File* Uji .txt

Nama File	**Ukuran File Sumber (mb)**	**Waktu Dekompresi (s)**	
		Tanpa TPL	**Dengan TPL**
file1	21.74	3.81	3.81
file2	43.46	7.80	5.57
file3	86.90	15.72	10.69
file4	173.77	31.35	17.20
Jumlah		**58.05**	**37.27**
Waktu Rata-rata		**14.51**	**9.31**

Berdasarkan hasil yang ditunjukkan Tabel 4.16 Perbandingan Waktu Dekompresi menunjukkan bahwa prosentase waktu rata-rata dekompresi metode *Huffman* menggunakan *task parallel library* lebih cepat sebesar 35.83% dibandingkan dengan dekompresi metode *Huffman* tanpa *task parallel library*.

Berdasarkan data yang ditunjukkan pada Tabel 4.16 Perbandingan Waktu Dekompresi maka dapat dilihat pada Gambar Grafik 4.13 Selisih Waktu Dekompresi *File* .txt.



Gambar 4.13 Grafik Selisih Waktu Dekompresi *File* Uji .txt

4.5.6 Hasil Dekompresi Dengan *File* Uji .pdf

Evaluasi juga dilakukan dengan cara membandingkan hasil dekompresi metode *Huffman* tanpa *task parallel library* dan hasil dekompresi metode *Huffman* dengan *task parallel library*. Hasil dekompresi metode *Huffman* tanpa *task parallel library* dapat dilihat pada tabel 4.14 Hasil Dekompresi *Huffman* tanpa TPL *File* Uji .pdf.

Tabel 4.17 Hasil Dekompresi *Huffman* tanpa TPL *File* Uji .pdf

Nama File	Ukuran File Sumber (mb)	Ukuran File Hasil (mb)	Core	Waktu Dekompresi (s)	Akurasi (%)
file1	15.03	15.03	1	6.81	100
file2	30.11	30.11	1	13.41	100
file3	60.20	60.20	1	28.29	100
file4	120.55	120.55	1	53.95	100

Berdasarkan hasil evaluasi pada Tabel Tabel 4.17 Hasil Dekompresi *Huffman* tanpa TPL, menunjukkan bahwa waktu dekompresi terendah dimiliki oleh file1 selama 6.81 detik dengan ukuran *file* 15.03 mb dan besar *file* yang didapatkan setelah proses dekompresi sebesar 15.03 mb. Sedangkan waktu dekompresi tertinggi dimiliki oleh file4 selama 120.55 detik dengan ukuran *file* 120.55 mb dan besar *file* yang didapatkan setelah proses dekompresi sebesar 120.55 mb.

Tabel 4.18 Hasil Dekompresi *Huffman* dengan TPL *File* Uji .pdf

Nama File	Ukuran File Sumber (mb)	Ukuran File Hasil (mb)	Core	Waktu Dekompresi (s)	Akurasi (%)
file1	15.03	15.03	2	6.81	100
file2	30.11	30.11	2	9.41	100
file3	60.20	60.20	2	19.41	100
file4	120.55	120.55	2	30.22	100

Berdasarkan hasil evaluasi pada Tabel 4.18 Hasil Dekompresi *Huffman* dengan TPL, menunjukkan menunjukkan bahwa waktu dekompresi terendah dimiliki oleh file1 selama 6.81 detik dengan ukuran *file* 15.03 mb dan besar *file* yang didapatkan setelah proses kompresi sebesar 15.03 mb. Sedangkan waktu dekompresi tertinggi dimiliki oleh file4 selama 30.22 detik dengan ukuran *file* 120.55 mb dan besar *file* yang didapatkan setelah proses dekompresi sebesar 120.55 mb.

Berdasarkan hasil evaluasi yang telah dilakukan, dekompresi metode *Huffman* dengan *task parallel library*, membuktikan bahwa dengan *file* yang sama waktu dekompresi yang didapatkan lebih cepat dibandingkan dengan dekompresi metode *Huffman* tanpa *task*

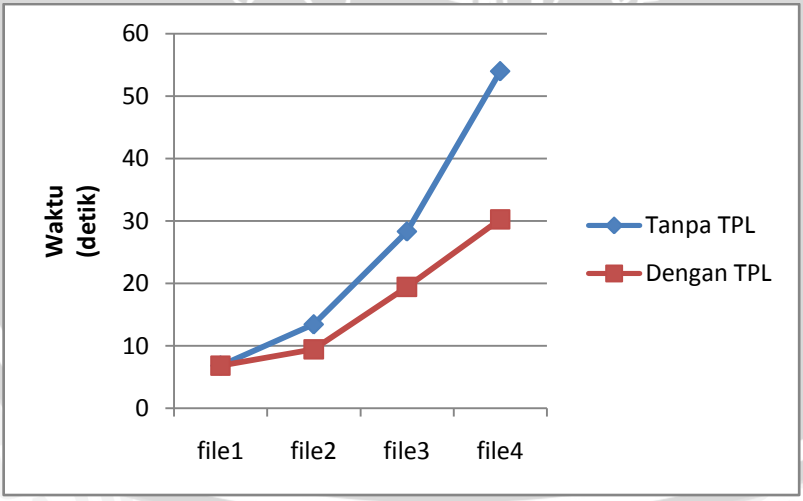
parallel library. Hasil ini ditunjukkan pada Tabel 4.19 Perbandingan Waktu Dekompresi *File* Uji .pdf.

Tabel 4.19 Perbandingan Waktu Dekompresi *File* Uji .pdf

Nama File	Ukuran File Sumber (mb)	Waktu Dekompresi (s)	
		Tanpa TPL	Dengan TPL
file1	15.03	6.81	6.81
file2	30.11	13.41	9.41
file3	60.20	28.29	19.41
file4	120.55	53.95	30.22
Jumlah		**102.46**	**65.85**
Waktu Rata-rata		**25.61**	**16.46**

Berdasarkan hasil yang ditunjukkan Tabel 4.19 Perbandingan Waktu Dekompresi menunjukkan bahwa prosentase waktu rata-rata dekompresi metode *Huffman* menggunakan *task parallel library* lebih cepat sebesar 35.72% dibandingkan dengan dekompresi metode *Huffman* tanpa *task parallel library*.

Berdasarkan data yang ditunjukkan pada Tabel 4.19 Perbandingan Waktu Dekompresi maka dapat dilihat pada Gambar Grafik 4.14 Selisih Waktu Dekompresi *File* .pdf.



Gambar 4.14 Grafik Selisih Waktu Dekompresi *File* Uji .pdf

4.6 Analisa Hasil Pengujian

Berdasarkan hasil pengujian dengan tiga format *file* yang berbeda yaitu, .doc, .txt. dan .pdf didapatkan hasil yang berbeda pada saat menggunakan metode *huffman* tanpa *task parallel library* dan metode *huffman* dengan *task parallel library*.

Tolak ukur keberhasilan percobaan ini diukur dari waktu kecepatan kompresi yang menunjukkan bahwa dengan TPL waktu kompresi lebih cepat dibandingkan dengan waktu kompresi tanpa TPL.

Pada saat melakukan kompresi, dengan ke 3 *file* uji, didapatkan hasil kompresi dengan waktu rata-rata 13,35 detik tanpa TPL ,dan waktu rata-rata 9.03 detik dengan TPL. Demikian juga dengan melakukan dekompresi, didapatkan hasil dekompresi menghasilkan waktu rata-rata 20.88 detik tanpa TPL ,dan waktu rata-rata 11.98 detik dengan TPL.

Rasio kompresi terbaik didapat dari *file* dengan format .doc sebesar 36.9%, 36.30% untuk format .txt. Sedangkan untuk *file* dengan format .pdf rasio kompresi hanya di peroleh rata-rata sebesar 0.71%.

Dari percobaan-percobaan diatas, dapat disimpulkan bahwa kompresi *file* menggunakan metode *hufman code* dengan teknik *task parallel library* lebih cepat dibandingkan dengan kompresi *file* menggunakan metode *hufman code* tanpa *teknik task parallel library*.

BAB V
PENUTUP

5.1 Kesimpulan

Kesimpulan yang diperoleh dari penelitian ini adalah sebagai berikut :

1. Dalam mengatasi permasalahan rendahnya kecepatan kompresi pada metode *Huffman*, maka ditambahkan teknik *task parallel library*. Dimana dilakukan pemanfaatan *core* yang terdapat pada komputer.
2. Hasil evaluasi menunjukkan, prosentase waktu rata-rata kompresi pada metode *Huffman* dengan *task parallel library* lebih cepat sebesar 32.35% dibandingkan dengan kompresi metode *Huffman* tanpa *task parallel library*.

5.2 Saran

Untuk penelitian lebih lanjut, dapat dilakukan penerapan *task parallel library* pada metode kompresi yang lain dengan subjek penelitian berupa *file* gambar, agar dapat diketahui waktu kompresi dan hasil rasio kompresi pada *file* tersebut.


UNIVERSITAS BRAWIJAYA

DAFTAR PUSTAKA

Ayuningtyas, Nadhira. 2007. *Implementasi Algoritma Huffman dalam Aplikasi Kompresi Teks padaLayanan SMS*. Bandung : Jurusan Teknik Informatika Institut Teknologi Bandung.

Callista, Nessya. 2007. *Aplikasi Greedy Pada Algoritma Huffman Untuk Kompresi Teks*. Sekolah Teknik Elektro Dan Informatika Institut Teknologi Bandung : Bandung

Held, G. 1998. *Learn Encryption Techniques with Basic and C++*. Wordware Publishing Inc : Plano

K A, Bharath. *Parallel fast compression unleashing the power of multi-core machines using the .NET TPL (Task Parallel Library"*. *Codeproject*. 08 January 2010. Web. 20 October 2010.

Linawati. 2004. *Perbandingan Kinerja Algoritma Kompresi Huffman, LZW, dan DMC pada Berbagai Tipe File*. Bandung : Jurusan Ilmu Komputer Universitas Katolik Parahyangan.

Prabawa, Arya Tri 2008. Kode *Huffman*. Bandung : Program Studi Teknik Informatika ITB.

Pu, Ida Mengyi. *Fundamental Data Compression*. Linacre House & Jordan Hill. Oxford

Salomon, D. 2004. *Data Compression*. New York : Springer-Verlag.

Wardoyo I., P. Kusdinar dan I.H. Taufik.2005.*Kompresi Teks dengan Menggunakan Algoritma Huffman*. Bandung : Jurusan Teknik Informatika Sekolah Tinggi Telkom.

Widhiartha, Putu Ashintya. 2008. Pengantar Kompresi Data. IlmuKomputer.com : Jakarta