UNIVERSITAS INDONESIA

ALGORITMA MARKOV CLUSTERING PARALEL UNTUK PENGELOMPOKAN PROTEIN

SKRIPSI

MUHAMMAD FAUZAN AKBAR MASYHUDI
0706261783

FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
PROGRAM STUDI SARJANA MATEMATIKA
DEPOK
JULI 2012

UNIVERSITAS INDONESIA

ALGORITMA MARKOV CLUSTERING PARALEL UNTUK PENGELOMPOKAN PROTEIN

SKRIPSI

Diajukan sebagai salah satu syarat untuk memperoleh gelar sarjana sains

MUHAMMAD FAUZAN AKBAR MASYHUDI
0706261783

FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
PROGRAM STUDI SARJANA MATEMATIKA
DEPOK
JULI 2012

HALAMAN PERNYATAAN ORISINALITAS

Skripsi ini adalah hasil karya saya sendiri, dan semua sumber baik yang dikutip maupun dirujuk telah saya nyatakan dengan benar.

Nama	: Muhammad Fauzan Akbar Masyhudi
NPM	: 0706261783
Tanda Tangan	:
Tanggal	: Juli 2012

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

HALAMAN PENGESAHAN

Skripsi ini diajukan oleh :
Nama : Muhammad Fauzan Akbar Masyhudi
NPM : 0706261783
Program Studi : Matematika
Judul Skripsi : Algoritma Markov Clustering Paralel Untuk Pengelompokan Protein

Telah berhasil diperiksa oleh Dewan Pembimbing untuk diuji sebagai bagian persyaratan yang diperlukan untuk memperoleh gelar Sarjana Sains pada Program Studi Matematika, Fakultas Matematika dan Ilmu Pengetahuan Alam Universitas Indonesia

DEWAN PEMBIMBING

Pembimbing : Alhadi Bustamam, Ph.D. ( )
Penguji I : Bevina D. Handari, Ph.D. ( )
Penguji II : Gatot F. Hertono, Ph.D. ( )
Penguji III : Drs. Frederik M. P., M.Kom ( )

Ditetapkan di : Depok
Tanggal : Juli 2012

KATA PENGANTAR

Alhamdulillah, segala puji syukur penulis panjatkan kepada Allah SWT, karena atas berkat dan rahmat-Nya, penulis dapat menyelesaikan skripsi ini. Penulisan skripsi ini dilakukan dalam rangka memenuhi salah satu syarat untuk mencapai gelar Sarjana Sains Jurusan Matematika pada Fakultas Matematika dan Ilmu Pengetahuan Alam Universitas Indonesia.

Penulis menyadari bahwa, tanpa bantuan dan bimbingan dari berbagai pihak, dari masa perkuliahan sampai pada penyusunan skripsi ini, sangatlah sulit bagi penulis untuk menyelesaikan skripsi ini. Oleh karena itu, penulis mengucapkan terima kasih kepada berbagai pihak antara lain:

(1) Pembimbing tugas akhir penulis, Bapak Alhadi Bustamam, Ph.D. yang telah menyediakan waktu, tenaga, dan pikiran untuk mengarahkan serta membimbing penulis dalam penyusunan skripsi ini. Terima kasih untuk kesabaran, nasehat, doa, dan dukungan yang telah diberikan selama penyusunan skripsi ini.

(2) Dra. Ida Fithriani M.S., selaku pembimbing akademik penulis yang telah memberikan masukan dan dukungan selama masa perkuliahan penulis.

(3) Dr. Yudi Satria dan Rahmi Rusin, M.ScTech, selaku ketua dan sekretaris Departemen Matematika, atas segala bantuan dan dukungan yang telah diberikan.

(4) Seluruh dosen dan staf pengajar departemen Matematika UI atas seluruh ilmu yang telah mereka berikan pada penulis

(5) Seluruh staf Tata Usaha, staf Perpustakaan, serta karyawan Departemen Matematika, terima kasih atas segala bantuannya.

(6) Keluarga penulis, yang telah memberikan bantuan material maupun dukungan moral.

(7) Teman-teman Matematika angkatan 2005 – 2011 yang telah menemani dan memotivasi penulis selama masa perkuliahan penulis. Terima kasih karena kalian telah memberikan penulis masa-masa yang layak dikenang.

(8) Anak-anak Komunitas Igo Indonesia terutama teman-teman yang menjadi tempat pelampiasan penulis dikala stress.

(9) Anak-anak JOSOL sebagai salah satu teman-teman bercanda penulis yang paling baik.

(10) Calon istri penulis yang selalu memotivasi penulis dengan senyum manisnya.

(11) Teman-teman penulis lainnya yang tidak dapat disebutkan satu per satu.

(12) Bon Jovi, Gun & Roses, Scorpions, The Beatles, Queen, dan pemusik-pemusik lain yang lagu-lagunya telah menemani penulis.

Penulis berharap Allah SWT berkenan membalas kebaikan semua pihak yang telah membantu penulis. Semoga skripsi ini bermanfaat bagi perkembangan ilmu pengetahuan. Akhir kata, penulis memohon maaaf apabila terdapat kesalahan ataupun kekurangan dalam penulisan skripsi ini.

Penulis

2012

ABSTRAK

Nama : Muhammad Fauzan Akbar Masyhudi
Program Studi : Matematika
Judul : Algoritma Markov Clustering Paralel Untuk Pengelompokan Protein

Algoritma Markov Clustering adalah algoritma pengelompokan yang banyak digunakan pada bidang bioinformatik. Operasi utama pada algoritma ini adalah operasi ekspansi. Pada operasi ekspansi dilakukan perkalian dua buah matriks. Karena data pada bidang bioinformatik umumnya berukuran sangat besar dan memiliki tingkat sparsity yang sangat tinggi, diperlukan metode untuk menghemat penggunaan memori dan mempercepat proses komputasi. Sementara itu, Graphics Processing Unit (GPU) berkembang menjadi suatu platform komputasi paralel dengan performa yang lebih baik dari pada Central Processing Unit (CPU). Pada skripsi ini data yang diproses disimpan dalam bentuk sparse matriks ELL-R dan perkalian matriks yang dilakukan menggunakan *Sparse Matrix Matrix Product* (SpMM) ELL-R. SpMM ELL-R dibuat dengan melakukan *Sparse Matrix Vector Product* (SpMV) ELL-R beberapa kali. Algoritma MCL yang dibuat menggunakan komputasi paralel dengan GPU.

Kata Kunci : bioinformatik, ELL-R, GPU, komputasi paralel, markov clustering, SpMM, SpMV
x+42 halaman ; 18 gambar ;16 tabel, 4 lampiran
Daftar Pustaka : 18 (1953-2010)

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

ABSTRACT

Nama : Muhammad Fauzan Akbar Masyhudi
Program Studi : Mathematics
Judul : Parallel Algorithm of Markov Clustering for Protein Clusterization

Markov Clustering Algorithm is a clustering algorithm that used often in bioinformatics. The main operation of this algorithm is expand operation. The multiplication of two matrix was done in expand operation. Because data processed in bioinformatics usually have a vast amount of information and have high sparsity, a method to save memory usage and make the computating process faster is needed. Meanwhile, Graphics Processing Unit (GPU) developed into a parallel computing platform with better performance compared to Central Processing Unit (CPU). In this *skripsi*, processed data stored using ELL-R sparse matrix and matrix multiplication done using Sparse Matrix Matrix Product (SpMM) ELL-R. SpMM ELL-R made by doing Sparse Matrix Vector Product (SpMV) ELL-R several times. MCL Algorithm made using parallel computing with GPU.

Keywords : bioinformatics, ELL-R, GPU, parallel computing, markov clustering, SpMM, SpMV
x+42 pages ; 18 pictures ;16 tables, 4 attachments
Bibliography : 18 (1953-2010)

DAFTAR ISI

HALAMAN PERNYATAAN ORISINALITAS iii
HALAMAN PENGESAHAN iv
KATA PENGANTAR v
ABSTRAK vii
ABSTRACT viii
DAFTAR ISI ix
DAFTAR GAMBAR xi
DAFTAR TABEL xii
DAFTAR LAMPIRAN xiii

1. PENDAHULUAN 1

1.1 Latar Belakang 1
1.2 Rumusan Masalah dan Ruang Lingkup 2
1.3 Tujuan Penelitian 2
1.4 Metode Penelitian 3

2. LANDASAN TEORI 4

2.1 Biologi Molekuler 4
2.2 Metode Penyimpanan Sparse Matriks 7
2.3 Markov Clustering (MCL) 13
2.4 SpMV dan SpMM 23
2.5 Komputasi Paralel, *Massive Paralel Computing*, dan CUDA 24

3. ALGORITMA MCL PARALEL DAN KERNEL PENDUKUNGNYA 32

3.1 Algoritma MCL Paralel 32
3.2 Metode Penyimpanan ELL-R Paralel dan Prune Paralel 33
3.3 Kernel Ekspansi 37
3.4 *Reduction* dan *Parallel Reduction* 40
3.5 Algoritma Inflasi Paralel 41
3.6 Algoritma *Find Chaos* Paralel 42

4. SIMULASI DAN ANALISA 43

4.1 Implementasi Kernel Kompresi. 44
4.2 Implementasi Kernel Ekspansi 45
4.3 Implementasi Kernel Inflasi 46
4.4 Implementasi Kernel *Find Chaos* 46
4.5 Analisa MCL Paralel 47

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

5. KESIMPULAN DAN SARAN **49**

DAFTAR PUSTAKA 51
LAMPIRAN.. 53

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

DAFTAR GAMBAR

Gambar 2.1 Struktur Molekul DNA 5
Gambar 2.2 Molekul Insulin 5
Gambar 2.3 Hubungan DNA, RNA, dan Protein 6
Gambar 2.4 Ilustrasi Proses Pembentukan Protein 6
Gambar 2.5 *Sparsity Pattern* yang Tidak Cocok dengan Metode DIA 9
Gambar 2.6 Graf Interaksi Protein pada Manusia 14
Gambar 2.7 Hasil Y2H dari Graf Interaksi Protein pada Manusia 14
Gambar 2.8 Contoh Proses MCL 17
Gambar 2.9 Contoh Input untuk Proses MCL 18
Gambar 2.10 Output dari Proses MCL dengan Input Graf pada Gambar 2.9 22
Gambar 2.11 Ilustrasi Proses SpMV 23
Gambar 2.12 Ilustrasi pembagian *Load Balance* yang Baik (a) dan Tidak Baik (b) 26
Gambar 2.13 Arsitektur Fermi 28
Gambar 2.14 *Streaming Multiprocessor* pada Arsitektur Fermi 29
Gambar 3.1 Ilustrasi Proses Kompresi ELL-R 35
Gambar 3.2 Ilustrasi Proses Pencarian Kolom Pengali 39
Gambar 3.3 Ilustrasi SpMV ELL-R 39
Gambar 3.4 Ilustrasi *Reduction* dan *Parallel Reduction* Dengan Operasi + ... 40

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

DAFTAR TABEL

Tabel 2.1 Contoh Penyimpanan Matriks Sparse 10
Tabel 2.2 Algoritma MCL 16
Tabel 2.3 Algoritma SpMV ELL-R 24
Tabel 3.1 Algoritma MCL Paralel 32
Tabel 3.2 Algoritma Kernel Kompresi 34
Tabel 3.3 Algoritma Kernel Ekspansi 37
Tabel 3.4 Algoritma Hadamard Paralel 41
Tabel 3.5 Algoritma Normalize Paralel 41
Tabel 3.6 Algoritma *Find Chaos* Paralel 42
Tabel 4.1 Waktu Tunggu (ms) Untuk Proses Kompresi 44
Tabel 4.2 Waktu Tunggu (ms) Untuk Proses Ekspansi 45
Tabel 4.3 *Speed-up* Proses Ekspansi 45
Tabel 4.4 Waktu Tunggu (ms) Untuk Proses Inflasi 46
Tabel 4.5 *Speed-up* Proses Inflasi 46
Tabel 4.6 Waktu Tunggu (ms) Untuk Proses *Find Chaos* 47
Tabel 4.7 *Speed-up* Proses *Find Chaos* 47

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

DAFTAR LAMPIRAN

Lampiran 1 *Listing* Program Sparse Generator 53
Lampiran 2 *Listing* Program MCL Serial... 54
Lampiran 3 *Listing* Program Utama MCL Paralel.. 56
Lampiran 4 *Listing* Kernel Program MCL Paralel 59

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

BAB 1
PENDAHULUAN

1.1 Latar Belakang

Proses Markov clustering (MCL) memiliki peranan yang sangat penting di berbagai bidang, terutama bidang bioinformatik (Dongen, 2008). Pada bidang bioinformatik, data-data yang akan dikelompokan biasanya berukuran sangat besar. Selain itu, umumnya matriks representasi dari data tersebut merupakan sparse matriks dimana persentasi dari elemen bukan nol sangat kecil (≤ 2%)(Vazquez et al., 2010). Karena dua sebab ini, untuk menyelesaikan proses MCL dengan efisien diperlukan proses komputasi.

Karena data yang akan diproses berukuran besar, penyelesaian proses MCL dengan satu komputer (prosesor) akan memakan waktu yang cukup lama. Hal ini dikarenakan satu prosesor hanya dapat melakukan komputasi sekuensial saja. Oleh karena itu, diperlukan komputasi paralel untuk mempercepat proses komputasi dalam penyelesaian proses MCL tersebut.

Komputasi paralel adalah teknik untuk melakukan komputasi secara bersamaan dengan memanfaatkan beberapa komputer/prosesor. Komputasi paralel umumnya memiliki efektifitas saat kapasitas data yang diproses sangat besar, baik karena harus mengolah data dalam jumlah besar ataupun karena tuntutan proses komputasi yang banyak. Dengan komputasi paralel diharapkan proses komputasi dapat diselesaikan dengan lebih cepat.

Klasifikasi komputasi paralel ada berbagai macam, misalnya *cluster computing, multicore computing,* dan *massive parralel processing* (Barney, 2010). *Massive parralel processing/computing* adalah komputasi paralel dengan menggunakan *General Purpose Graphic Processing Unit* (GPGPU) dalam jumlah banyak.

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

Selain dengan komputasi paralel, beban kerja komputer juga dapat dijadikan lebih ringan dengan cara penyimpanan matriks yang lebih efisien. Umumnya matriks representasi dari data pada bidang bioinformatik merupakan matriks sparse maka penyimpanan data dapat dikompres dengan hanya menyimpan elemen-elemen selain nol saja beserta informasi mengenai koordinat elemen-elemen tersebut (Bell & Garland, 2009). Diharapkan dengan pengurangan beban kerja proses komputasi dapat berlangsung semakin cepat.

Tujuan utama dari skripsi ini adalah menerapkan algoritma paralel untuk menyelesaikan proses MCL pada dataset interaksi protein yang telah dikompres sehingga diharapkan adanya peningkatan kecepatan (*speed up*) proses komputasi. Oleh karena itu, pada skripsi ini akan dibangun algoritma paralel yang memiliki *speed-up* yang baik untuk menyelesaikan proses MCL dengan bantuan mesin GPGPU. Selain itu, pada skripsi ini juga akan dikaji *speed up* dari proses penyelesaian metode MCL dimana data yang akan dikelompokan dikompres dengan metode ELL-R sparse matriks.

1.2 Permasalahan dan Ruang Lingkup Penelitian

Permasalahan yang akan dibahas pada skripsi ini adalah bagaimana membangun algoritma paralel dengan *speed-up* yang baik untuk menyelesaikan proses Markov Clustering (MCL) pada data interaksi protein yang telah dikompres dengan metode ELL-R.

Pada skripsi ini, mesin GPGPU yang digunakan adalah NVIDIA GTX 460, dataset protein yang digunakan untuk simulasi adalah dataset *artificial* yang dibuat dari matriks random, dan metode penyimpanan matriks sparse yang digunakan untuk menyimpan representasi data protein adalah metode ELL-R.

1.3 Jenis Penelitian dan Metode Penelitian

Jenis penelitian yang digunakan adalah studi literatur, dan metode penelitian yang digunakan adalah pengembangan algoritma dan simulasi.

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

1.4 Tujuan Penelitian

Tujuan dari penulisan skripsi ini adalah membangun algoritma paralel dengan *speed-up* yang baik untuk menyelesaikan proses Markov clustering. Pada skripsi ini juga diberikan algoritma paralel untuk menyelesaikan proses Markov Clustering pada dataset interaksi protein. Algoritma ini akan disimulasikan pada gcc dengan CUDA dan mesin GPGPU NVIDIA GTX 460. Selain itu, pada skripsi ini juga akan dikaji *speed-up* yang bertujuan untuk mengukur kinerja dari algoritma paralel pada penyelesaian proses Markov Clustering.

BAB 2
LANDASAN TEORI

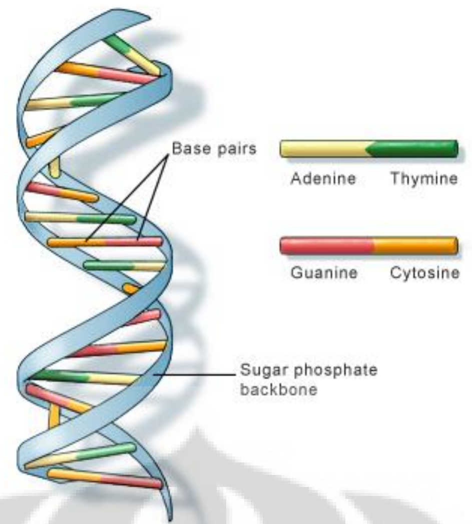
2.1 Biologi Molekuler

Setiap makhluk hidup, termasuk manusia, memiliki informasi-informasi biologis yang dibutuhkan untuk tumbuh, bertahan hidup, dan melanjutkan keturunan. Informasi-informasi ini tersimpan dalam suatu molekul yang disebut DNA (*deoxyribonucleic acid*). DNA juga menyimpan informasi-informasi biologis yang membuat setiap spesies unik. Selain itu, pada DNA juga tersimpan informasi yang akan diturunkan suatu individu ke keturunannya. Dengan kata lain, DNA pada suatu individu merupakan kombinasi dari DNA orang tua-nya. (Waterman, 1995)

Umumnya DNA terletak di nukleus tetapi terkadang ada pula DNA yang terdapat pada mitokondria. DNA yang terletak pada mitokondia disebut juga mtDNA. DNA tersusun dari empat bahan kimia dasar yaitu basa purin adenin (A) dan guanin (G) serta basa pirimidin sitosin/*cytosine* (C) dan timin (T) (Watson dan Crick, 1953). keempat bahan dasar ini membentuk pasangan yang disebut pasangan dasar, yaitu antara A dengan T dan C dengan G. Pasangan dasar ini membentuk untaian ganda yang sangat panjang yang disebut *double helix*. Untaian ganda yang terbentuk pada setiap organisme memiliki urutan pasangan dasar yang berbeda. Perbedaan urutan inilah yang menyebabkan perbedaan pada tiap-tiap individu (Waterman, 1995).

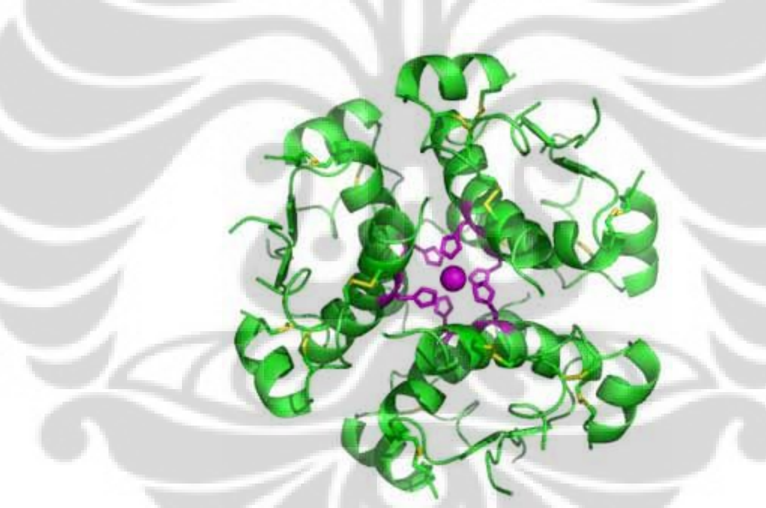
Meskipun informasi-informasi biologis penting tersebut terdapat pada DNA, yang menjalankan sebagian besar bahkan hampir seluruh fungsi-fungsi kehidupan pada tingkat molekul/sel dalam setiap individu adalah protein. Protein adalah suatu molekul kompleks yang tersusun dari rantai asam amino. Diantara protein-protein ini, ada protein yang dapat bekerja sendiri. Akan tetapi sebagian besar protein bekerja bersama-sama untuk menjalankan suatu fungsi. Salah satu contoh dari protein adalah insulin yang berguna untuk mengatur kadar gula dalam darah. Insulin tersusun atas 51 asam amino. (Wang, 2003)

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012



Gambar 2.1 Struktur Molekul DNA

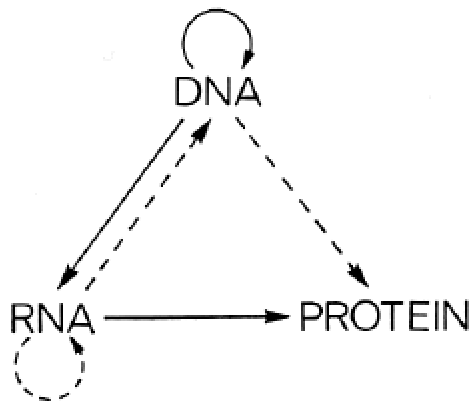
[Diambil dari http://ghr.nlm.nih.gov, 28 Februari 2012, 17.28]



Gambar 2.2 Molekul Insulin

[Diambil dari http://www.labspaces.net, 10 Januari 2012, 10.31]

Hubungan antara DNA dan protein dijalaskan dalam *Central Dogma of Molecular Biology*. Hubungan ini juga melibatkan satu molekul lain yaitu RNA (*ribonucleic acid*). Terdapat tiga jenis RNA yaitu *messenger* RNA (mRNA) yang berfungsi menyalin dan menyimpan informasi dari DNA untuk pembentukan protein, *transfer* RNA (tRNA) yang berfungsi memindahkan mRNA untuk diproses lebih lanjut, dan *ribosomal* RNA (rRNA) yang berfungsi sebagai katalis dalam proses pembentukan protein. (Crick, 1958)



Gambar 2.3 Hubungan DNA, RNA, dan Protein

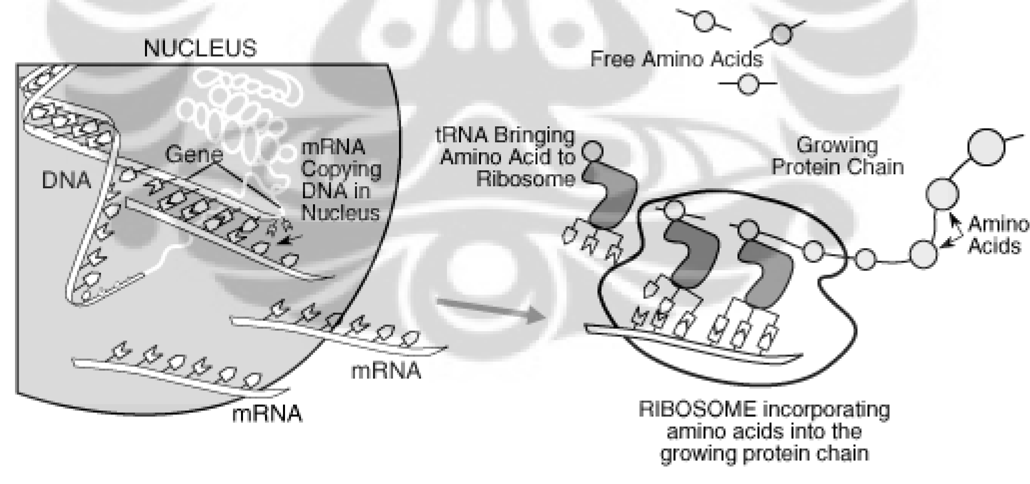
[Diambil dari jurnal NATURE vol. 227 hal. 561 oleh Crick, F. H. C.]

Pada gambar 2.2 ,

tanda ⟶ menunjukan perubahan yang umum terjadi

tanda - -➝ menunjukan perubahan yang hanya terjadi pada kasus tertentu.
Umumnya hanya terjadi pada beberapa jenis virus.

dan tidak ada tanda berarti perubahan tersebut tidak mungkin terjadi



Gambar 2.4 Ilustrasi Proses Pembentukan Protein

[Diambil dari http://fourier.eng.hmc.edu/bioinformatics/, 17 Januari 2012, 08.00]

Proses pembentukan protein terdiri dari dua tahap, transkripsi dan translasi. Pada proses transkripsi, informasi-informasi biologis dari DNA disalin dan disimpan di mRNA. Informasi-informasi biologis ini kemudian dibawa dari

inti sel (nukleus) ke ribosom untuk menjalani proses translasi. Di ribosom, tRNA akan mengikat dan membawa berbagai asam amino. Asam amino ini kemudian bersama dengan mRNA membentuk berbagai protein sesuai dengan informasi pada mRNA tersebut. (Waterman, 1995)

Dalam tubuh suatu individu multiselular terdapat banyak sekali protein. Dalam tubuh manusia saja terdapat lebih dari 50.000 jenis protein (Hansmann, 2006). Protein-protein ini saling berinteraksi untuk menjalankan fungsi-fungsi di dalam tubuh. Meskipun begitu, suatu protein tidak berinteraksi dengan seluruh protein lainnya. Umumnya suatu protein hanya berinteraksi dengan beberapa protein lainnya dan suatu protein dapat memiliki beberapa interaksi yang masing-masing menjalankan fungsi yang berbeda (Wang, 2003).

Seluruh protein yang ada dalam tubuh setiap individu akan saling berinteraksi dan menghasilkan data interaksi protein. Data interaksi protein ini dapat direpresentasikan dalam bentuk matriks dimana 0 menandakan tidak ada hubungan/interaksi antara dua protein yang bersesuaian dengan baris dan kolom dari matriks tersebut dan 1 menandakan ada hubungan/interaksi.

Umumnya suatu protein hanya berinteraksi dengan paling banyak 15-30 protein lain (Satuluri et al., 2010). Karena jumlah interaksi yang sedikit dan jumlah protein dalam suatu individu yang sangat banyak, matriks representasi dari data interaksi protein suatu individu akan memiliki banyak sekali elemen nol. Matriks seperti ini disebut juga matriks sparse.

2.2 Metode Penyimpanan Matriks Sparse

Umumnya matriks representasi dari data-data yang digunakan dalam bidang bioinformatika, termasuk data interaksi protein, berukuran sangat besar ($\geq 10^5$) dengan persentasi dari elemen tak-nol yang sangat rendah ($\leq$ 2%) (Vazquez et al., 2010). Karena berukuran sangat besar, proses pengolahan dan analisa data secara manual akan memakan waktu cukup lama. Selain itu, proses pengolahan dan analisa data secara manual memiliki tingkat kesalahan yang cukup tinggi. Karena dua alasan inilah, untuk proses pengolahan dan analisa data perlu digunakan bantuan komputer.

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

Karena ukuran matriks yang sangat besar dan persentasi yang sangat tinggi dari elemen nol, maka penulisan matriks secara menyeluruh tidaklah efisien dan menghabiskan cukup banyak memori komputer. Untuk mengatasi masalah ini dikembangkan metode penyimpanan matriks dimana elemen nol tidak disimpan (Bell dan Garland, 2008). Metode penyimpanan yang dikembangkan untuk menyimpan matriks sparse hanya akan menyimpan elemen-elemen tak-nol beserta informasi tambahan yang dapat digunakan untuk mengetahui baris dan kolom dari elemen tersebut. Informasi tambahan ini bisa menunjukan posisi baris dan kolom dari suatu elemen tak-nol secara eksplisit bisa juga secara implisit. Metode-metode yang ada antara lain DIA, COO, CSR, ELL, HYB, dan ELL-R.

Diantara metode-metode yang ada, DIA adalah metode yang paling sederhana. Ide dari metode ini adalah menyimpan matriks secara diagonal (DIA). Pada metode ini, matriks sparse disimpan menjadi satu matrik data dan satu vektor offset. Matriks data berisi elemen-elemen tak-nol dan vektor offset menunjukan data pada kolom matriks data yang bersesuaian ada di diagonal mana. 0 menandakan diagonal utama, $-a$ menandakan a diagonal di bawah diagonal utama, dan a menandakan a diagonal di atas diagonal utama

Metode COO menyimpan matriks sparse sebagai tiga vektor: vektor entry yang berisi data elemen tak-nol, vektor kolom yang berisi kolom dari data pada vektor entry yang bersesuaian, dan vektor baris yang berisi baris dari data pada vektor entry yang bersesuaian.

Metode CSR (*compressed sparse rows*) juga menyimpan matriks sparse menjadi tiga vektor seperti metode COO. Hanya saja metode ini bisa menghemat lebih banyak memori. Hal ini disebabkan karena CSR tidak menggunakan vektor baris. Sebagai gantinya, metode CSR menggunakan vektor pointer dimana vektor ini menunjukan elemen ke berapa dari vektor entry dimana terjadi pergantian baris ke baris berikutnya.

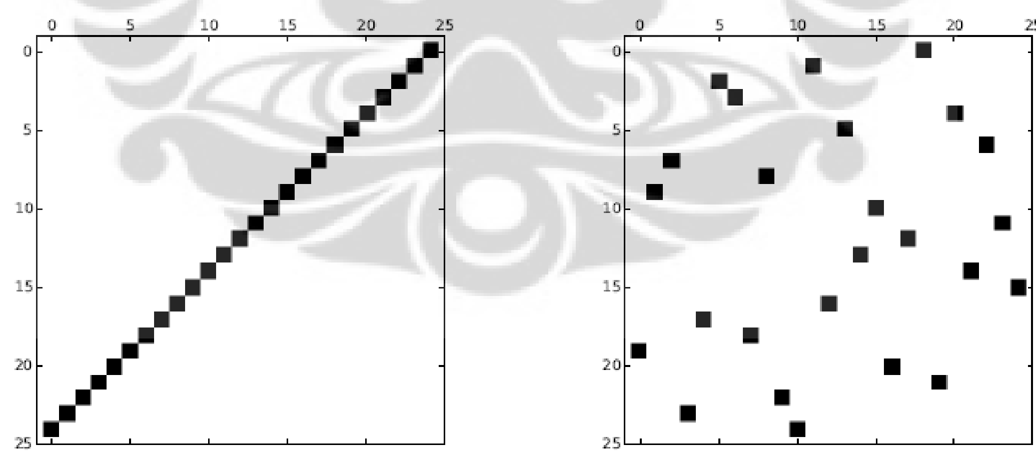
Metode ELL (Ellpack) menyimpan matriks sparse menjadi dua matriks: matriks data yang berisi data elemen tak-nol dan matriks kolom yang berisi informasi kolom untuk data tak-nol yang bersesuaian. Penyimpanan dengan metode ELL dapat diterapkan dengan cepat saat elemen-elemen tak-nol pada matriks awal membentuk suatu pola maupun saat elemen-elemen tak-nol tersebut

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

benar-benar acak sekalipun. Karena inilah metode ELL cukup banyak dikembangkan. Salah satu metode yang dikembangkan dari metode ELL adalah metode HYB (gabungan ELL dan COO) dan ELL-R.

Pada metode ELL-R, dua matriks yang dihasilkan pada metode ELL diubah menjadi dua vektor, vektor data dan vektor kolom. Sedangkan untuk informasi mengenai baris digunakan satu vektor tambahan yang menunjukan banyak elemen tak-nol total dari baris pada matriks awal yang bersesuaian. Metode ini memiliki kemiripan dengan metode COO dan CSR dimana matriks sparse disimpan menjadi tiga vektor.

Pada metode DIA, kolom dan baris dari suatu elemen disimpan secara implisit dan dapat diketehui dengan batuan data dari vektor offset. Meskipun begitu, diantara metode-metode ini, metode DIA merupakan metode yang tidak efektif. Hal ini disebabkan metode ini hanya baik diterapkan saat matriks sparse yang akan disimpan memiliki pola dimana diagonal-diagonal yang sejajar dengan diagonal utama cukup padat. Matriks-matriks representasi pada dunia nyata umumnya memiliki bentuk/pola yang tidak cocok untuk disipan dengan metode DIA (Bell dan Garland, 2008). Karena hal inilah metode DIA jarang digunakan.



Gambar 2.5 *Sparsity Pattern* yang Tidak Cocok dengan Metode DIA

Gambar di sebelah kiri menunjukan *worst case* dari metode DIA. Dengan *sparsity pattern* seperti ini, metode DIA akan memerlukan lebih banyak memori dibandingkan jika matriks disimpan dengan cara penyimpanan biasa. Gambar di

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

sebelah kanan menunjukan metriks sparse yang tidak berpola. Elemen-elemen tak-nol dari matriks ini tersebar secara acak. Matriks sparse seperti inilah yang banyak dijumpai di dunia nyata. Penggunaan metode DIA pada matriks seperti ini juga tidak efisien karena diagonal-diagonal matriks ini tidak cukup padat.

Tabel 2.1 Contoh Penyimpanan Matriks Sparse

Metode	Contoh Penyimpanan
Matriks Biasa	$A = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$
DIA	$\text{Data} = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix}$ $\text{Offset} = [-2 \quad 0 \quad 1]$
COO	$\text{Data} = [1 \quad 7 \quad 2 \quad 8 \quad 5 \quad 3 \quad 9 \quad 6 \quad 4]$ $\text{Col} = [1 \quad 2 \quad 2 \quad 3 \quad 1 \quad 3 \quad 4 \quad 2 \quad 4]$ $\text{Row} = [1 \quad 1 \quad 2 \quad 2 \quad 3 \quad 3 \quad 3 \quad 4 \quad 4]$
ELL	$\text{Data} = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix}$ $\text{Col} = \begin{bmatrix} 1 & 2 & * \\ 2 & 3 & * \\ 1 & 3 & 4 \\ 2 & 4 & * \end{bmatrix}$
HYB	$\text{Data1} = \begin{bmatrix} 1 & 7 \\ 2 & 8 \\ 5 & 3 \\ 6 & 4 \end{bmatrix}$ $\text{Col1} = \begin{bmatrix} 1 & 2 \\ 2 & 3 \\ 1 & 3 \\ 2 & 4 \end{bmatrix}$ $\text{Data2} = [9]$ $\text{Col2} = [4]$ $\text{Row2} = [3]$
CSR	$\text{Data} = [1 \quad 7 \quad 2 \quad 8 \quad 5 \quad 3 \quad 9 \quad 6 \quad 4]$ $\text{Col} = [1 \quad 2 \quad 2 \quad 3 \quad 1 \quad 3 \quad 4 \quad 2 \quad 4]$ $\text{Ptr} = [1 \quad 3 \quad 5 \quad 8]$
ELL-R	$\text{Data} = [1 \quad 7 \quad 2 \quad 8 \quad 5 \quad 3 \quad 9 \quad 6 \quad 4]$ $\text{Col} = [1 \quad 2 \quad 2 \quad 3 \quad 1 \quad 3 \quad 4 \quad 2 \quad 4]$ $\text{SRow} = [2 \quad 2 \quad 3 \quad 2]$

Berbeda dengan metode DIA yang dipengaruhi *sparsity pattern*, metode COO tidak dipengaruhi oleh *sparsity pattern*. Metode COO bisa diterapkan pada matriks sparse apapun. Pada metode COO, baris dan kolom dari suatu elemen tak-nol ditampilkan secara eksplisit. Untuk lebih menghemat penggunaan memori dikembangkanlah metode-metode lain yang menampilkan baris secara implisit dan kolom secara eksplisit maupun sebaliknya.

Pada metode ELL, informasi mengenai baris disimpan secara implisit pada matriks data sedangkan informasi mengenai kolom ditampilkan secara eksplisit pada matriks kolom. Metode ini efektif saat jumlah elemen tak-nol pada tiap baris relatif sama. Metode ini akan kehilangan efektifitasnya saat ada baris dengan elemen tak-nol yang jauh lebih banyak dibandingkan dengan rata-rata elemen tak-nol per baris. Hal ini disebabkan karena ukuran matriks data dan matriks kolom akan membesar secara signifikan tetapi elemen yang perlu disimpan hanya bertambah beberapa elemen saja.

Misalkan baris ketiga dari matiks A pada Tabel 2.1 diganti menjadi $[5 \quad 1 \quad 3 \quad 9]$. Di sini data yang bertambah hanya satu elemen saja. Akan tetapi metode peyimpanan ELL akan menghasilkan:

$$\text{Data} = \begin{bmatrix} 1 & 7 & * & * \\ 2 & 8 & * & * \\ 5 & 1 & 3 & 9 \\ 6 & 4 & * & * \end{bmatrix} \qquad \text{Col} = \begin{bmatrix} 1 & 2 & * & * \\ 2 & 3 & * & * \\ 1 & 2 & 3 & 4 \\ 2 & 4 & * & * \end{bmatrix}$$

Pada contoh ini, untuk menyimpan matriks biasa dibutuhkan memori untuk menyimpan matriks 4×4 sedangkan untuk menyimpan matriks dengan metode ELL dibutuhkan memori untuk menyimpan dua matriks 4×4. Dari contoh ini terlihat bahwa *worst case* dari metode ELL akan terjadi saat ada baris tanpa elemen nol. Meskipun tidak seluruh elemen pada suatu baris tak-nol, baris-baris dengan elemen tak-nol yang lebih banyak akan menyebabkan metode ELL menyiapkan kolom lebih banyak padahal hanya beberapa baris dari kolom tersebut yang membutuhkan tambahan memori. Karena hal inilah metode ELL kurang efisien.

Untuk mengatasi hal ini dikembangkan metode *hybird* (HYB). Sesuai dengan namanya, metode HYB merupakan metode yang terdiri dari gabungan dua metode yang berbeda, ELL dan COO. Pada metode ini, sebagian besar matriks akan disimpan dengan metode ELL sementara baris-baris dengan elemen tak-nol yang lebih banyak akan disimpan dengan metode COO. Dengan penggunaan metode ini jumlah memori yang dibutuhkan akan berkurang secara signifikan. Akan tetapi, seperti telah dijelaskan sebelumnya, pada metode COO informasi baris dan kolom ditampilkan secara eksplisit. Karena itu penggunaan memori masih bisa dikurangi lebih lanjut.

Dua metode yang terakhir adalah CSR dan ELL-R. *Best case* dari dua metode ini memang tidak lebih baik dibanding metode-metode lainnya. Akan tetapi kedua metode ini adalah metode yang paling stabil (performa *best case* dan *worst case* tidak terlalu berbeda jauh). Pada dua metode ini, informasi baris ditampilkan secara implisit dan informasi kolom ditampilkan secara eksplisit. Sepintas kedua metode ini terlihat serupa. Perbedaan besar baru akan terlihat saat terdapat baris tanpa elemen tak-nol. Seandainya terdapat baris seperti ini metode CSR akan membutuhkan lebih banyak memori dari pada metode ELL-R. Hal ini dikarenakan vektor ketiga pada metode CSR (vektor pointer) menunjukan indeks pada vektor data dan vektor kolom dimana informasi mengenai baris berikutnya pada matriks awal dimulai. Hal ini menyebabkan semua vektor pada metode CSR memerlukan tambahan memori. Sementara pada metode ELL-R hanya vektor ketiga (vektor jumlah baris) saja yang memerlukan tambahan memori.

Contoh metode CSR dan ELL-R pada Tabel 2.1 memerlukan jumlah memori yang sama. Akan tetapi jika matriks A pada Tabel 2.1 diperbesar dengan menambahankan satu baris $[0 \quad 0 \quad 0 \quad 0]$ sebagai baris kelima maka metode CSR akan menghasilkan:

$$\text{Data} = [1 \quad 7 \quad 2 \quad 8 \quad 5 \quad 3 \quad 9 \quad 6 \quad 4 \quad 0]$$
$$\text{Col} = [1 \quad 2 \quad 2 \quad 3 \quad 1 \quad 3 \quad 4 \quad 2 \quad 4 \quad 1]$$
$$\text{Ptr} = [1 \quad 3 \quad 5 \quad 8 \quad 10\,]$$

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

Sedangkan metode ELL-R akan menghasilkan:

$$\text{Data} = [1 \quad 7 \quad 2 \quad 8 \quad 5 \quad 3 \quad 9 \quad 6 \quad 4]$$
$$\text{Col} = [1 \quad 2 \quad 2 \quad 3 \quad 1 \quad 3 \quad 4 \quad 2 \quad 4]$$
$$\text{SRow} = [2 \quad 2 \quad 3 \quad 2 \quad 0\,]$$

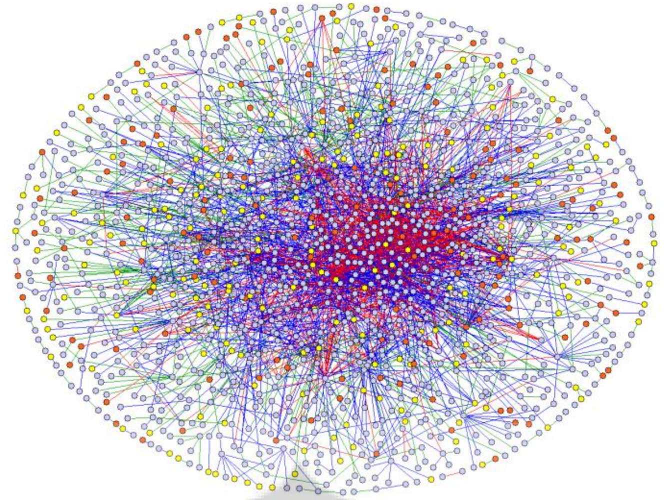
Terlihat bahwa metode CSR memerlukan memori untuk menyimpan tiga elemen tambahan untuk setiap baris tanpa elemen nol sedangkan metode ELL-R hanya memerlukan memori untuk menyimpan satu elemen tambahan untuk setiap baris tanpa elemen nol. Selain itu, semakin banyak data yang disimpan vektor pointer pada metode CSR akan menyimpan karakter lebih banyak dari pada vektor jumlah baris pada matode ELL-R.

Dengan mempertimbangkan stabilitas dan efisiensi dari tiap metode, metode ELL-R adalah metode penyimpanan yang paling baik. Oleh karena itu, pada skripsi ini data interaksi protein akan disimpan dengan menggunakan metode ELL-R.

2.3 Markov Clustering (MCL)

Seperti dijelaskan sebelumnya pada Subbab 2.1 bahwa suatu protein umumnya paling banyak hanya memiliki 15-30 interaksi. Selain itu, interaksi-interaksi tersebut bisa memiliki fungsi yang berbeda. Untuk mendapatkan informasi yang berguna diperlukan suatu proses pengelompokan dari data interaksi protein yang ada. Informasi yang diperoleh dari hasil pengelompokan nantinya dapat digunakan untuk mempelajari struktur dan fungsi dari suatu sistem. Informasi ini juga dapat digunakan untuk mempelajari gen-gen yang bermasalah karena suatu penyakit (Bustamam et al., 2010). Selain itu, informasi yang didapat juga dapat digunakan untuk mempelajari organisme yang bersangkutan dan membantu proses perancangan obat bagi organisme tersebut (Satuluri et al., 2010). Hasil yang didapat dapat membantu proses perancangan obat karena obat-obatan umumnya berinteraksi dengan protein yang memiliki gugus tertentu (Shargel dan Yu, 1999).

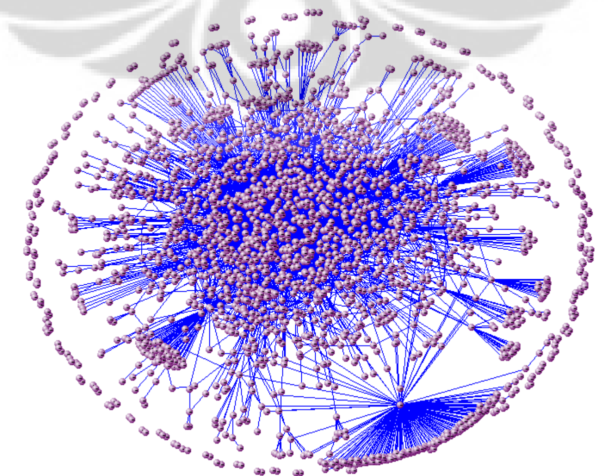
Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012



Gambar 2.6 Graf Interaksi Protein pada Manusia

[Diambil dari http://www.mdc-berlin.de/, 7 Februari 2012, 17.20]

Metode-metode yang telah dikembangkan untuk melakukan proses pengelompokan protein antara lain *yeast two-hybird system* (Y2H), *protein co-immunoprecipitation* (cOIP), dan *mass spectrometry* (MS). Sayangnya hasil pengelompokan dengan menggunakan metode-metode ini masih kurang baik. Metode-metode ini masih menghasilkan satu atau beberapa kelompok (*cluster*) yang terlalu besar serta banyak kelompok dengan hanya satu anggota (*cluster singelton*). Hal ini menyebabkan hasil yang kurang baik karena cluster yang terlalu besar tidak memberikan informasi yang spesifik sedangkan *cluster singelton* tidak memberikan informasi yang berarti. (Satuluri et al., 2010)



Gambar 2.7 Hasil Y2H dari Graf Interaksi Protein pada Manusia

[Diambil dari http://interactome.dfci.harvard.edu/, 7 Februari 2012, 17.29]

Pada tahun 2000, Stijn van Dongen mengembangkan metode pengelompokan yang diberi nama Markov Clustering (MCL). Metode MCL dibuat berdasarkan simulasi dari stokastik flow pada suatu graf. Metode ini diberi nama Markov Clustering karena kolom stokastik matriks dapat direpresentasikan sebagai matriks yang berisi kemungkinan transisi dari suatu *random walk* atau Markov *chain* yang terdefinisi pada suatu graf.

Kolom stokastik matriks adalah matriks dimana jumlah dari seluruh elemen pada setiap kolom sama dengan satu. Kolom stokastik matriks M yang memiliki kolom sebanyak verteks dari suatu graf G dapat direpresentasikan sebagai matriks yang berisi kemungkinan transisi dari suatu *random walk* atau Markov *chain* yang terdefinisi pada graf G. Elemen $M(i,j)$ menandakan kemungkinan transisi dari verteks v_i ke verteks v_j. Elemen $M(i,j)$ juga dapat dianggap sebagai stokastik flow dari verteks v_i ke verteks v_j. Jika dianggap sebagai stokastik flow maka kolom dari matriks M menunjukan flow keluar (*out-flow*) dan baris dari matriks M menunjukan flow masuk (*in-flow*). Pada kolom stokastik matriks jumlah dari seluruh elemen pada tiap baris tidak harus sama dengan satu. (Dongen, 2008)

Ada lima kelebihan metode MCL dibandingkan dengan metode-metode lainnya. Metode MCL menghasilkan *wellbalanced* cluster. Metode MCL termasuk *bootstrapping method*, user tidak perlu memasukan informasi tambahan untuk menjalankan metode ini. Hasil pengelompokan dapat diatur dengan menggunakan parameter inflasi r. Metode ini memiliki skalabilitas yang baik. Kelebihan yang terakhir adalah proses-proses dari metode ini dapat dijelaskan secara matematika. (Dongen, 2008)

Selain lima kelebihan ini, metode MCL memiliki dua kelebihan lain. Jika dibandingkan dengan metode-metode lainnya metode MCL lebih handal dan dapat memberikan hasil lebih cepat. Metode MCL dikatakan lebih handal karena saat input yang diberikan merupakan *undirected graph* metode MCL masih dapat memberikan hasil pengelompokan dengan informasi yang berarti sedangkan metode-metode lain tidak. Selain itu, metode MCL dapat memberikan hasil pengelompokan dengan lebih cepat karena metode ini hanya menggunakan operasi-operasi matematika sederhana. (Bustamam et al., 2010)

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

Metode MCL terdiri dari dua operasi utama, operasi *expand*(M) dan *inflate*(M,r). Operasi *expand*(M) adalah operasi perkalian matriks. Pada operasi ini dilakukan perkalian matriks $M*M$. Operasi *expand*(M) dilakukan dengan tujuan untuk meningkatkan flow antar verteks yang sudah ada dan membuka flow-flow potensial baru. Pada operasi *inflate*(M,r) setiap elemen pada matriks M dipangkatkan dengan parameter inflasi r. Nilai r harus lebih besar dari satu, umumnya nilai r yang digunakan adalah dua. Nilai r harus lebih besar dari satu karena proses inflasi bertujuan memperkuat flow-flow yang kuat dan memperlemah flow-flow yang lemah. Operasi pemangkatan ini disebut juga sebagai operasi Hadamard. (Dongen, 2008)(Satuluri et al, 2010)

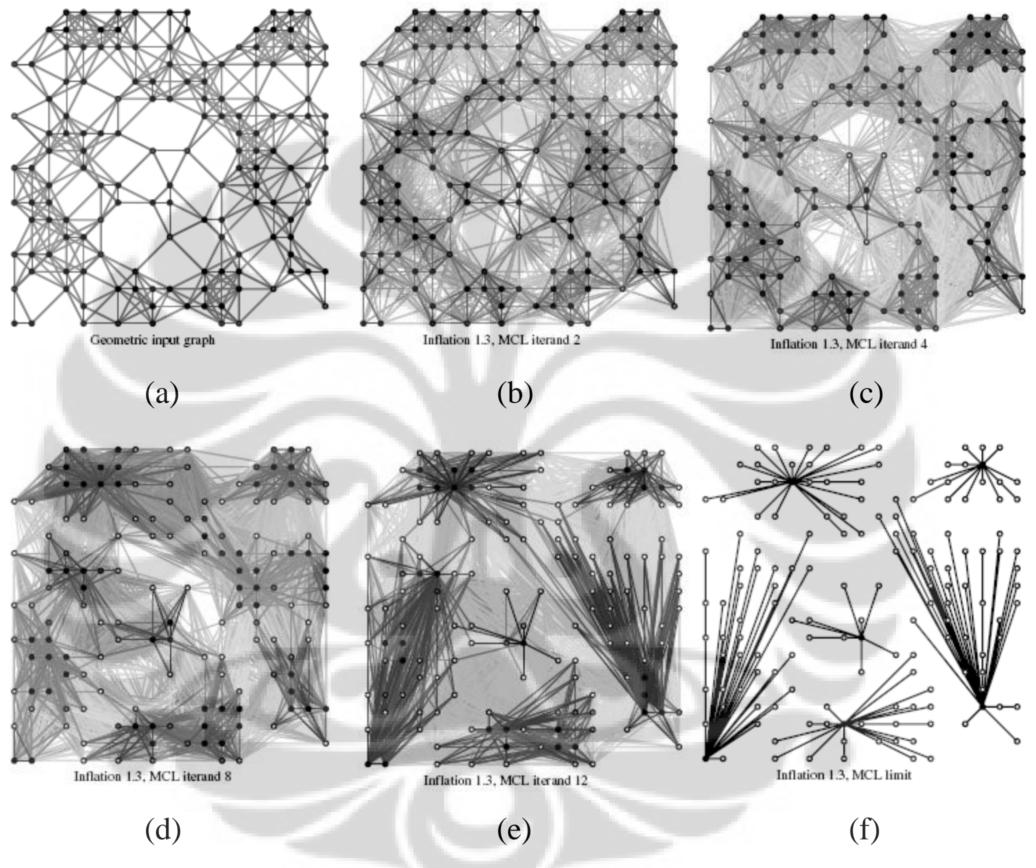
Selain operasi *expand*(M) dan *inflate*(M,r), metode MCL memiliki dua operasi tambahan yaitu *normalize*(M) dan *prune*(M). Pada *normalize*(M) matriks yang akan diproses diubah menjadi kolom stokastik matriks. Operasi *normalize*(M) langsung dijalankan setelah operasi *inflate*(M,r) selesai sehingga sering kali operasi ini dianggap jadi satu dengan operasi *inflate*(M,r). Operasi *prune*(M) dilakukan dengan tujuan mempercepat konvergensi metode MCL. Pada operasi ini, elemen-elemen yang sudah mendekati nol akan langsung diubah menjadi nol. (Satuluri et al, 2010)

Tabel 2.2 Algoritma MCL

Input : Matriks M, parameter inflasi r
Output : Matriks M dimana elemen-elemennya telah ter-*cluster*
1. $M := M + I$ // tambahkan *self-loop* pada graf
2. $M :=$ *normalize*(M)
repeat
3. $M :=$ *expand*(M)
4. $M :=$ *inflate*(M, r)
5. $M :=$ *normalize*(M)
6. $M :=$ *prune*(M)
until M converges

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

Proses MCL dikatakan telah mencapai konvergensi apabila nilai *global chaos* telah mendekati nol. Apabila nilai *global chaos* telah mendekati nol berarti sudah tidak ada perubahan signifikan pada cluster baru. Nilai *global chaos* adalah maksimum dari nilai *chaos* pada setiap kolom. Nilai *Chaos* pada kolom ke-i dapat dihitung dengan cara menghitung nilai maksimum kolom ke-i dikurang jumlah kuadrat dari elemen-elemen pada kolom ke-i.



Gambar 2.8 Contoh Proses MCL

[Diambil dari jurnal SIAM Vol. 30 No. 1 hal. 129 oleh Dongen, S. V.]

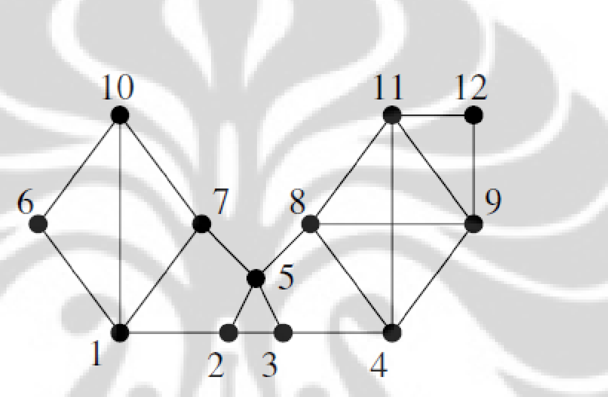
Pada Gambar 2.8 diperlihatkan graf input (a), graf hasil MCL setelah 2 iterasi (b), graf hasil MCL setelah 4 iterasi (c), graf hasil MCL setelah 8 iterasi (d), graf hasil MCL setelah 12 iterasi (e), dan graf output dari metode MCL (f).

Pada Gambar 2.8 dapat dilihat hasil dari proses ekspansi dan inflasi. Dari Gambar 2.8 (a)-(e) terlihat bahwa jika suatu gambar dibandingkan dengan gambar pada iterasi-iterasi sebelumnya terdapat flow/edge baru yang muncul karena proses ekspansi. Selain itu terlihat pula bahwa ketebalan dari tiap-tiap edge

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

berbeda-beda. Edge yang lebih tebal menunjukan hubungan yang lebih kuat jika dibandingkan dengan edge yang lebih tipis. Melemah dan menguatnya suatu hubungan terjadi setelah proses inflasi dilaksanakan. Seiring dengan jumlah iterasi yang dilakukan, hubungan yang kuat akan menjadi semakin kuat dan yang lemah menjadi semakin lemah sehingga pada akhirnya menghilang dan didapat hasil akhir pada Gambar 2.8 (f).

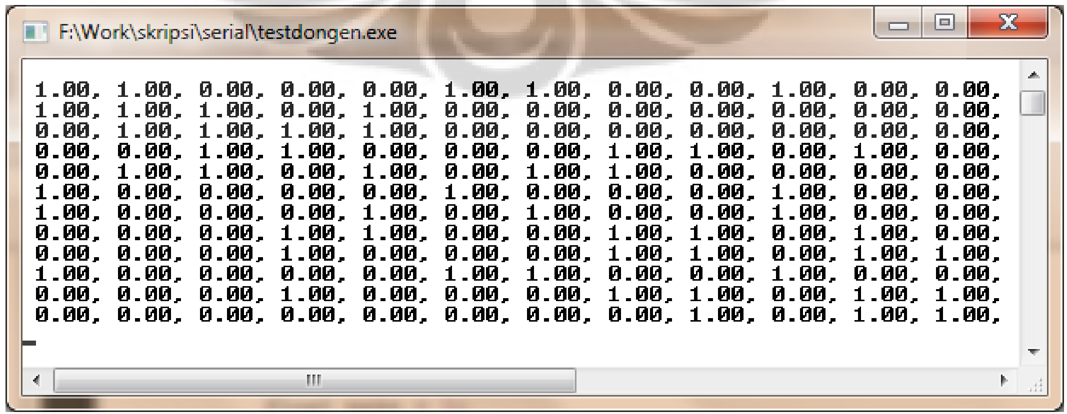
Untuk mendapatkan gambaran yang lebih baik tentang proses MCL dapat digunakan contoh yang lebih sederhana. Dalam hal ini akan digunakan graf dengan 12 verteks dan 20 edge sebagai input. Graf ini dapat dilihat pada Gambar 2.9.



Gambar 2.9 Contoh Input untuk Proses MCL

[Diambil dari jurnal SIAM Vol. 30 No. 1 hal. 129 oleh Dongen, S. V.]

Dari graf ini didapat matriks representasi sebagai berikut:



F:\Work\skripsi\serial\testdongen.exe

```
1.00, 1.00, 0.00, 0.00, 0.00, 1.00, 1.00, 0.00, 0.00, 1.00, 0.00, 0.00,
1.00, 1.00, 1.00, 0.00, 1.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 1.00, 1.00, 1.00, 1.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 1.00, 1.00, 0.00, 0.00, 0.00, 1.00, 1.00, 0.00, 1.00, 0.00,
0.00, 1.00, 1.00, 0.00, 1.00, 0.00, 1.00, 1.00, 0.00, 0.00, 0.00, 0.00,
1.00, 0.00, 0.00, 0.00, 0.00, 1.00, 0.00, 0.00, 0.00, 1.00, 0.00, 0.00,
1.00, 0.00, 0.00, 0.00, 1.00, 0.00, 1.00, 0.00, 0.00, 1.00, 0.00, 0.00,
0.00, 0.00, 0.00, 1.00, 1.00, 0.00, 0.00, 1.00, 1.00, 0.00, 1.00, 0.00,
0.00, 0.00, 0.00, 1.00, 0.00, 0.00, 0.00, 1.00, 1.00, 0.00, 1.00, 1.00,
1.00, 0.00, 0.00, 0.00, 0.00, 1.00, 1.00, 0.00, 0.00, 1.00, 0.00, 0.00,
0.00, 0.00, 0.00, 1.00, 0.00, 0.00, 0.00, 1.00, 1.00, 0.00, 1.00, 1.00,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 1.00, 0.00, 1.00, 1.00,
```

Selanjutnya proses MCL akan dijalankan dengan matriks ini sebagai inputnya.

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

Proses MCL dengan input ini berlangsung sebanyak 7 iterasi dimana hasil dari tiap-tiap iterasi adalah sebagai berikut:

Iterasi pertama:

```
F:\Work\skripsi\serial\testdongen.exe

 energy = 0.032653

0.37, 0.09, 0.02, 0.00, 0.06, 0.28, 0.19, 0.00, 0.00, 0.31, 0.00, 0.00,
0.06, 0.35, 0.20, 0.01, 0.15, 0.03, 0.08, 0.01, 0.00, 0.02, 0.00, 0.00,
0.01, 0.20, 0.35, 0.05, 0.15, 0.00, 0.02, 0.05, 0.01, 0.00, 0.01, 0.00,
0.00, 0.02, 0.09, 0.29, 0.06, 0.00, 0.00, 0.18, 0.17, 0.00, 0.17, 0.11,
0.06, 0.20, 0.20, 0.05, 0.40, 0.00, 0.08, 0.05, 0.01, 0.02, 0.01, 0.00,
0.13, 0.02, 0.00, 0.00, 0.00, 0.28, 0.08, 0.00, 0.00, 0.17, 0.00, 0.00,
0.13, 0.09, 0.02, 0.00, 0.06, 0.13, 0.33, 0.01, 0.00, 0.17, 0.00, 0.00,
0.00, 0.02, 0.09, 0.19, 0.06, 0.00, 0.02, 0.29, 0.17, 0.00, 0.17, 0.11,
0.00, 0.00, 0.02, 0.19, 0.02, 0.00, 0.00, 0.18, 0.27, 0.00, 0.27, 0.26,
0.24, 0.02, 0.00, 0.00, 0.02, 0.28, 0.19, 0.00, 0.00, 0.31, 0.00, 0.00,
0.00, 0.00, 0.02, 0.19, 0.02, 0.00, 0.00, 0.18, 0.27, 0.00, 0.27, 0.26,
0.00, 0.00, 0.00, 0.05, 0.00, 0.00, 0.00, 0.05, 0.10, 0.00, 0.10, 0.26,
```

Perhatikan bahwa pada matriks awal kolom pertama hanya memiliki 5 elemen tak-nol. Setelah iterasi pertama banyak elemen tak-nol pada kolom pertama bartambah menjadi 7. Terjadi perubahan juga pada kolom kedua yang awalnya hanya memiliki 4 elemen tak-nol setalah iterasi pertama jadi memiliki 9 elemen tak-nol. Begitu pula dengan kolom-kolom lainnya. Penambahan jumlah elemen tak-nol ini disebabkan karena proses ekspansi yang membuka flow-flow potensial baru.

Iterasi kedua:

```
F:\Work\skripsi\serial\testdongen.exe

 energy = 0.066689

0.42, 0.08, 0.02, 0.00, 0.05, 0.40, 0.32, 0.00, 0.00, 0.41, 0.00, 0.00,
0.03, 0.29, 0.21, 0.00, 0.17, 0.01, 0.05, 0.00, 0.00, 0.01, 0.00, 0.00,
0.01, 0.21, 0.29, 0.02, 0.16, 0.00, 0.01, 0.02, 0.00, 0.00, 0.00, 0.00,
0.00, 0.02, 0.07, 0.23, 0.05, 0.00, 0.00, 0.21, 0.18, 0.00, 0.18, 0.15,
0.03, 0.29, 0.28, 0.02, 0.43, 0.01, 0.06, 0.02, 0.00, 0.01, 0.00, 0.00,
0.10, 0.01, 0.00, 0.00, 0.00, 0.14, 0.08, 0.00, 0.00, 0.12, 0.00, 0.00,
0.14, 0.06, 0.02, 0.00, 0.04, 0.13, 0.24, 0.00, 0.00, 0.15, 0.00, 0.00,
0.00, 0.02, 0.07, 0.21, 0.05, 0.00, 0.00, 0.23, 0.18, 0.00, 0.18, 0.15,
0.00, 0.00, 0.02, 0.25, 0.01, 0.00, 0.00, 0.25, 0.29, 0.00, 0.29, 0.31,
0.27, 0.03, 0.00, 0.00, 0.01, 0.30, 0.23, 0.00, 0.00, 0.30, 0.00, 0.00,
0.00, 0.00, 0.02, 0.25, 0.01, 0.00, 0.00, 0.25, 0.29, 0.00, 0.29, 0.31,
0.00, 0.00, 0.00, 0.03, 0.00, 0.00, 0.00, 0.03, 0.05, 0.00, 0.05, 0.08,
```


Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

Iterasi ketiga:

```
F:\Work\skripsi\serial\testdongen.exe

 energy = 0.059026

0.58, 0.08, 0.02, 0.00, 0.04, 0.58, 0.56, 0.00, 0.00, 0.58, 0.00, 0.00,
0.01, 0.21, 0.19, 0.00, 0.17, 0.00, 0.01, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.17, 0.20, 0.00, 0.16, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.01, 0.04, 0.18, 0.02, 0.00, 0.00, 0.18, 0.17, 0.00, 0.17, 0.16,
0.01, 0.46, 0.47, 0.00, 0.53, 0.00, 0.02, 0.00, 0.00, 0.00, 0.00, 0.00,
0.04, 0.00, 0.00, 0.00, 0.00, 0.04, 0.04, 0.00, 0.00, 0.04, 0.00, 0.00,
0.09, 0.03, 0.01, 0.00, 0.02, 0.08, 0.10, 0.00, 0.00, 0.09, 0.00, 0.00,
0.00, 0.01, 0.04, 0.17, 0.02, 0.00, 0.00, 0.18, 0.17, 0.00, 0.17, 0.16,
0.00, 0.00, 0.02, 0.32, 0.01, 0.00, 0.00, 0.32, 0.33, 0.00, 0.33, 0.34,
0.27, 0.02, 0.00, 0.00, 0.01, 0.28, 0.26, 0.00, 0.00, 0.28, 0.00, 0.00,
0.00, 0.00, 0.02, 0.32, 0.01, 0.00, 0.00, 0.32, 0.33, 0.00, 0.33, 0.34,
0.00, 0.00, 0.00, 0.01, 0.00, 0.00, 0.00, 0.01, 0.01, 0.00, 0.01, 0.01,
```

Iterasi keempat

```
F:\Work\skripsi\serial\testdongen.exe

 energy = 0.061315

0.80, 0.06, 0.01, 0.00, 0.03, 0.80, 0.80, 0.00, 0.00, 0.80, 0.00, 0.00,
0.00, 0.10, 0.10, 0.00, 0.10, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.09, 0.09, 0.00, 0.08, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.01, 0.11, 0.00, 0.00, 0.00, 0.11, 0.11, 0.00, 0.11, 0.11,
0.00, 0.74, 0.76, 0.00, 0.76, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.02, 0.00, 0.00, 0.00, 0.00, 0.02, 0.02, 0.00, 0.00, 0.02, 0.00, 0.00,
0.00, 0.00, 0.01, 0.11, 0.00, 0.00, 0.00, 0.11, 0.11, 0.00, 0.11, 0.11,
0.00, 0.00, 0.01, 0.39, 0.00, 0.00, 0.00, 0.39, 0.39, 0.00, 0.39, 0.39,
0.18, 0.01, 0.00, 0.00, 0.00, 0.18, 0.18, 0.00, 0.00, 0.18, 0.00, 0.00,
0.00, 0.00, 0.01, 0.39, 0.00, 0.00, 0.00, 0.39, 0.39, 0.00, 0.39, 0.39,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
```

Iterasi kelima

```
F:\Work\skripsi\serial\testdongen.exe

 energy = 0.030105

0.95, 0.01, 0.00, 0.00, 0.01, 0.95, 0.95, 0.00, 0.00, 0.95, 0.00, 0.00,
0.00, 0.02, 0.02, 0.00, 0.02, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.01, 0.01, 0.00, 0.01, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.04, 0.00, 0.00, 0.00, 0.04, 0.04, 0.00, 0.04, 0.04,
0.00, 0.96, 0.97, 0.00, 0.96, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.03, 0.00, 0.00, 0.00, 0.03, 0.03, 0.00, 0.03, 0.03,
0.00, 0.00, 0.00, 0.46, 0.00, 0.00, 0.00, 0.46, 0.46, 0.00, 0.46, 0.46,
0.05, 0.00, 0.00, 0.00, 0.00, 0.05, 0.05, 0.00, 0.00, 0.05, 0.00, 0.00,
0.00, 0.00, 0.00, 0.46, 0.00, 0.00, 0.00, 0.46, 0.46, 0.00, 0.46, 0.46,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
```


Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

Pada hasil dari iterasi kedua sampai kelima terlihat bahwa pada masing-masing kolom muncul flow yang semakin lama menjadi semakin kuat dan juga flow yang semakin lama menjadi semakin lemah. Melemah dan menguatnya flow ini disebabkan karena proses inflasi. Flow-flow baru yang muncul dari proses ekspansi tidak terlihat karena nilainya memang semakin kecil.

Iterasi keenam

```
F:\Work\skripsi\serial\testdongen.exe

 energy = 0.002786

1.00, 0.00, 0.00, 0.00, 0.00, 1.00, 1.00, 0.00, 0.00, 1.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 1.00, 1.00, 0.00, 1.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.50, 0.00, 0.00, 0.00, 0.50, 0.50, 0.00, 0.50, 0.50,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.50, 0.00, 0.00, 0.00, 0.50, 0.50, 0.00, 0.50, 0.50,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
```

Hasil akhir, iterasi ketujuh:

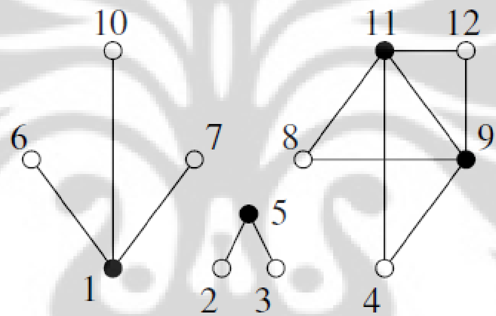
```
F:\Work\skripsi\serial\testdongen.exe

 energy = 0.000016

1.00, 0.00, 0.00, 0.00, 0.00, 1.00, 1.00, 0.00, 0.00, 1.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 1.00, 1.00, 0.00, 1.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.50, 0.00, 0.00, 0.00, 0.50, 0.50, 0.00, 0.50, 0.50,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
0.00, 0.00, 0.00, 0.50, 0.00, 0.00, 0.00, 0.50, 0.50, 0.00, 0.50, 0.50,
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00,
```

Proses MCL berhenti setelah iterasi ketujuh. Pada contoh ini matriks yang dihasilkan pada iterasi keenam dan ketujuh terlihat sama karena program hanya menampilkan hasil sampai dua angka dibelakang koma. Seandainya kedua matriks ini ditampilkan dengan presisi yang lebih tinggi akan terlihat perbedaannya. Karena hal inilah walaupun matriks setelah iterasi keenam dan ketujuh terlihat sama nilai *energy* dari kedua matriks ini berbeda.

Graf output bisa didapatkan dengan melihat baris-baris yang masih memiliki elemen tak-nol. Pada contoh ini baris pertama, kelima, kesembilan, dan kesebelas. Dari sini diketahui bahwa pusat dari cluster-cluster yang dihasilkan adalah vertek 1, 5, 9, dan 11. Elemen dari masing-masing cluster dapat dilihat dari baris-baris tersebut secara keseluruhan. Sebagai contoh baris pertama pada iterasi ketujuh berisi:

[1.00 0.00 0.00 0.00 0.00 1.00 1.00 0.00 0.00 1.00 0.00 0.00]

Hal ini menunjukan bahwa diperoleh cluster dengan pusat verteks 1 dan elemen-elemen cluster adalah verteks 1, 6, 7, 10. Dengan cara yang sama didapat dua cluster lain dimana cluster kedua berpusat di verteks 5 dan beranggotakan verteks 2, 3, dan 5 sedangkan cluster ketiga berpusat di verteks 9, 11 dan beranggotakan verteks 4, 8, 9, 11, dan 12.



Gambar 2.10 Output dari Proses MCL dengan Input Graf pada Gambar 2.9

[Diambil dari jurnal SIAM Vol. 30 No. 1 hal. 129 oleh Dongen, S. V.]

Pada metode MCL banyaknya cluster yang dihasilkan dapat dilihat dari banyaknya baris tak-nol yang bebas linear. Baris tak-nol dengan elemen yang sama menunjukan bahwa terdapat suatu cluster dengan pusat cluster lebih dari satu. Dari contoh bisa dilihat baris kesembilan dan kesebelas dimana verteks 9 dan 11 merupakan pusat cluster dari cluster yang sama.

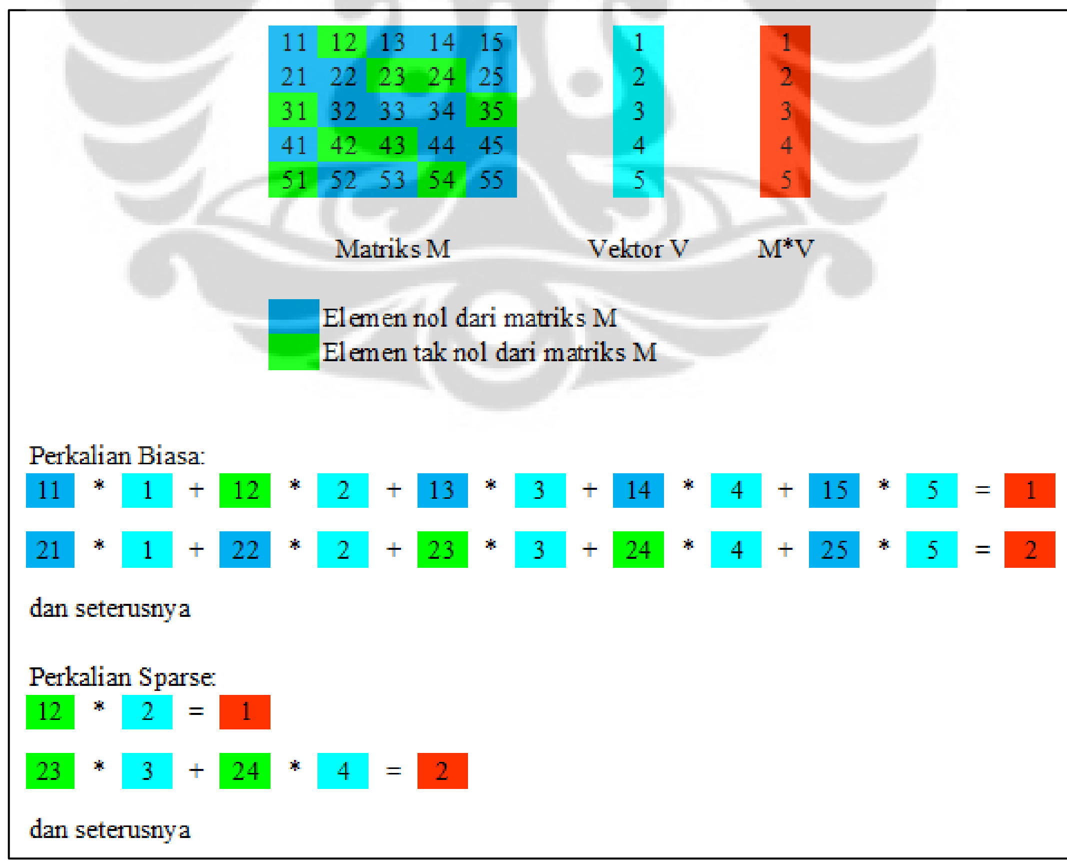
Banyak pusat cluster pada suatu cluster adalah $1/n$, dimana n adalah nilai dari elemen tak-nol pada baris yang merepresentasikan cluster tersebut. Dari contoh terlihat bahwa baris pertama memiliki 1 sebagai nilai elemen tak-nol dan cluster pertama memiliki satu pusat. Baris kesembilan memiliki 0.5 sebagai nilai elemen tak-nol sehingga banyak pusat cluster pada cluster tersebut adalah dua (1/0.5).

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

2.4 SpMV dan SpMM

Selain menghemat memori, metode penyimpanan sparse matriks juga dapat dimanfaatkan untuk mempercepat proses perkalian matriks. Hal ini dapat dilakukan dengan cara memodifikasi perkalian matriks biasa menjadi perkalian matriks sparse/*Sparse Matrix Matrix product* (SpMM). Dengan bertambah cepatnya proses perkalian matriks, diharapkan proses MCL juga akan mengalami pertambahan kecepatan.

Pada prakteknya SpMM terdiri dari beberapa SpMV (*Sparse Matrix Vector product*). Pada SpMV dilakukan perkalian matriks *M*, yang disimpan dengan suatu metode penyimpanan sparse matriks, dengan vektor *V*, yang disimpan secara biasa. Pada SpMM matriks *M* dikalikan dengan vektor-vektor kolom/baris dari matriks *N* untuk menghasilkan *M**N*. Algoritma SpMV berbeda tergantung metode penyimpanan yang digunakan. (Bell dan Garland, 2008) (Bustamam et al. 2010)



Gambar 2.11 Ilustrasi Proses SpMV

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

Dari ilustrasi ini terlihat bahwa untuk mendapatkan nilai M*V metode perkalian sparse membutuhkan jumlah operasi yang lebih sedikit dibandingkan dengan metode perkalian biasa. Terlihat, untuk mendapatkan M*V[1] dan M*V[2] saja metode perkalian biasa membutuhkan 10 operasi perkalian dan 8 operasi penjumlahan. Sedangkan metode perkalian sparse hanya membutuhkan 3 operasi perkalian dan 1 operasi penjumlahan.

Tabel 2.3 Algoritma SpMV ELL-R

Input : ELL-R sparse matriks *M*, vektor *V*
Output : Vektor $M*V^t$
1. **set** k := 0
2. **for** i = 1..n row
3. **if** srow[i] ≠ 0 **do**
4. **for** j = 1..srow[i]
5. k++
6. $M*V^t$ [i] :=+ data[k]**V*[col[k]]
end for
7. **else** $M*V^t$ [i] := 0
end if
end for

2.5 Komputasi Paralel, *Massive Parallel Computing*, dan CUDA

Untuk mempercepat proses MCL/R-MCL lebih lanjut dapat diterapkan komputasi paralel. Komputasi paralel adalah teknik untuk melakukan proses komputasi dengan memanfaatkan dua atau lebih komputer/prosessor secara bersamaan. Dengan memanfaatkan lebih banyak prosessor, beban kerja yang perlu dilakukan dapat dibagi dan didistribusikan ke prosessor lainnya sehingga dengan beban kerja yang lebih sedikit diharapkan program dapat dijalankan dengan lebih cepat.

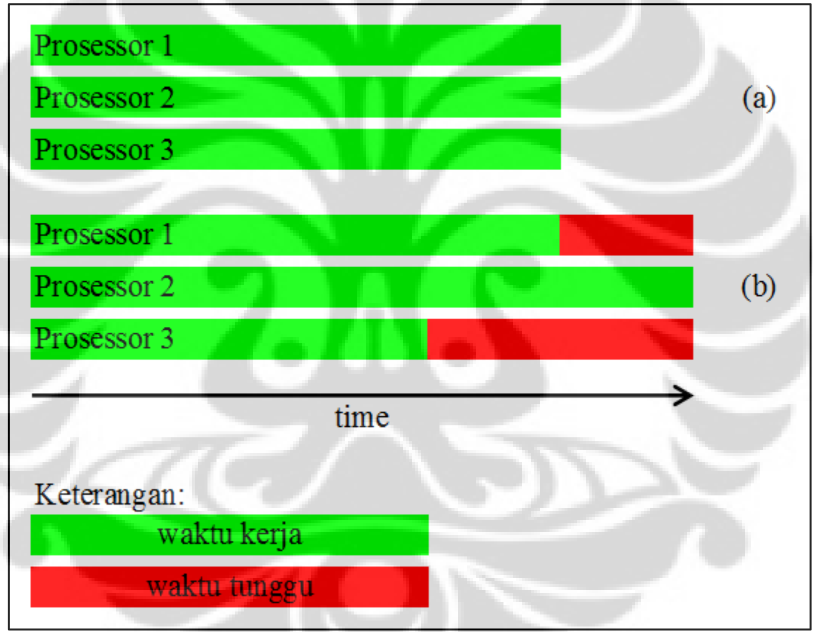
Tidak semua program dapat dijalankan secara paralel dan tidak semua program yang dijalankan secara paralel dapat berjalan lebih cepat dibandingkan program yang dijalankan secara sekuensial (Coffin, 1992). Karena itu tantangan dalam menjalankan komputasi paralel adalah mencari bagian-bagian yang dapat dijalankan secara paralel, menentukan apakah bagian tersebut layak dijalankan secara paralel, lalu memastikan *running-time* dari program paralel lebih cepat jika dibandingkan dengan *running-time* dari program sekuensial.

Dalam komputasi paralel penambahan prosessor tidak serta-merta menambah kecepatan *running-time*. Adakalanya penambahan prosessor justru membuat program berjalan lebih lambat. Hal ini disebabkan karena suatu fenomena yang disebut *bottleneck*. Fenomena *bottleneck* terjadi karena perpindahan data antara memori/prosessor juga memakan waktu. Fenomena ini sering terjadi saat hasil-hasil yang diperoleh saat komputasi paralel berlangsung disatukan kembali untuk memperoleh hasil secara keseluruhan. Karena hal inilah terkadang pada suatu program tidak semua bagian yang bisa dikerjakan secara paralel dikerjakan secara parelel. Terkadang beberapa bagian tetap dikerjakan secara sekuensial untuk mengurangi fenomena *bottleneck*. (Barney, 2010)

Selain memilih bagian-bagian mana yang akan dikerjakan secara paralel, seorang programmer juga perlu memperhatikan *data dependency, load balance, concurency,* dan sinkronisasi data pada bagian-bagian yang akan dikerjakan secara paralel. Jika keempat hal ini dilakukan dengan baik maka performa dari komputasi paralel juga dapat menjadi semakin baik.

Data dependency sesuai dengan namanya adalah ketergantungan data. Yang dimaksud dengan memperhatikan *data dependency* adalah memperhatikan apakah perintah yang akan dijalankan memerlukan data dari perintah sebelumnya. Jika data dari perintah sebelumnya diperlukan maka sebelum perintah yang baru dijalankan data yang dibutuhkan tersebut sudah harus tersedia dan dapat diakses. Jika hal ini tidak dipenuhi maka akan terdapat waktu tunggu sebelum perintah baru dapat dijalankan. (Barney, 2010)

Yang dimaksud dengan *load balance* adalah pembagian kerja yang seimbang. Untuk mendapatkan kecepatan maksimum setiap prosessor yang digunakan harus diberi beban kerja yang sama. Pembagian beban kerja yang tidak seimbang dapat menyebabkan tambahan *cost*, baik waktu maupun biaya. Saat beban kerja yang diberikan tidak seimbang, prosessor dengan bagian kerja lebih sedikit dapat menyelesaikan kerja lebih cepat. Meskipun begitu, prosessor tersebut tetap harus menyala dan menunggu prosessor lain selesai untuk kemudian menyatukan kembali data yang telah diolah. Selain itu, prosessor dengan beban kerja lebih banyak akan terus bekerja padahal ada prosessor lain yang telah menyelesaikan kerjanya. (Coffin, 1992)



Gambar 2.12 Ilustrasi Pembagian *Load Balance* yang Baik (a) dan Tidak Baik (b)

Selain itu, programmer juga perlu memperhatikan apakah perintah-perintah pada tiap-tiap prosessor dapat dijalankan secara bersamaan. Kemampuan suatu perintah untuk dapat dijalankan secara bersamaan inilah yang disebut *concurrency*. *Concurrency* berkaitan erat dengan *data dependency*. Seandainya ada prosessor yang mulai bekerja lebih lambat dibanding prosessor lainnya maka waktu penyelesaian program akan melambat juga. (Barney, 2010)

Hal selanjutnya yang perlu diperhatikan adalah sinkronisasi data. Yang dimaksud dengan sinkronisasi data adalah proses penyelarasan data antar prosessor pada suatu tahap paralel dengan tahap paralel lainnya atau dengan tahap sekuensial. Hal yang perlu diperhatikan pada tahap sinkronisasi data adalah memastikan semua prosessor telah menyelesaikan tugas mereka serta memastikan tidak terjadi konflik memori. (Sanders dan Kandrot, 2010)

Berdasarkan taksonomi Flynn, proses komputasi dapat dikelompokan berdasarkan jumlah perintah dan jumlah data yang diproses. Pembagian proses komputasi berdasarkan taksonomi Flynn adalah: *Single Instruction Single Data* (SISD), *Single Instruction Multiple Data* (SIMD), *Multiple Instruction Single Data* (MISD), dan *Multiple Instruction Multiple Data* (MIMD). Pada pengelompokan ini SISD berlaku pada komputasi sekuensial sedangkan SIMD, MISD, dan MIMD berlaku pada komputasi paralel. (Fountain, 1994)

Pada awalnya peningkatan kinerja komputer dilakukan dengan cara meningkatkan kecepatan prosessor. Akan tetapi pada tahun 2003 cara ini dianggap mulai mencapai batasnya. Peningkatan kecepatan prosessor menjadi sulit karena konsumsi energi yang dibutuhkan menjadi semakin besar dan panas yang dihasilkan menjadi semakin tinggi. Terhambat karena dua alasan ini, banyak perusahaan yang mulai mengembangkan komputer dengan dua prosessor pada tahun 2005. Komputer dengan jumlah CPU (*central processing unit*) lebih dari satu disebut *multicore computer*. Kemunculan *multicore computer* kemudian memicu kemunculan *multicore computing*. Pada tahun 2006, perusahaan NVIDIA mulai mengembangkan *general purpose* (GP) *graphics processing unit* (GPU). Dengan GPGPU, *graphics card* yang awalnya hanya bisa melakukan pengolahan citra dapat digunakan untuk mengolah proses-proses komputasi lain. GPGPU inilah yang memicu kemunculan *massive parallel computing.* (Bustamam et al., 2010)(Sanders dan Kandrot, 2010)

Selain kemunculan GPGPU, arsitektur dari *graphics card* saat itu juga telah memiliki banyak prosessor karena tuntutan proses pengolahan citra yang tinggi terutama untuk pembuatan animasi film dan game. *Graphics card* pertama yang telah menggunakan GPGPU saja (GeForce 8800 GT) telah memiliki 112 prosessor, jauh lebih banyak dari pada jumlah prosessor pada *multicore computer*.

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

Saat ini *graphics card* buatan NVIDIA memiliki jumlah prosessor yang bervariasi antara 8 (GeForce 8400 GS) – 3.072 (GeForce GTX 690). Selain itu, NVIDIA juga mengembangkan arsitektur *graphics card* Tesla dan Fermi. Arsitektur ini menempatkan GPU dan memori sedemikian rupa sehingga dapat meningkankan kecepatan proses komunikasi data antara GPU dan memori. Jumlah prosessor yang sangat banyak dan arsitektur *graphics card* yang mendukung inilah yang membuat *speed-up* dari *massive parallel computing* sangat tinggi. (Sanders dan Kandrot, 2010)

NVIDIA juga mengembangkan bahasa pemrograman CUDA (*Compute Unified Device Architecture*) yang merupakan ekstensi dari bahasa pemrograman standar C. Dengan menggunakan CUDA, programmer bisa membuat program yang dapat dijalankan pada CPU dan GPU. Bagian-bagian serial dari program akan dijalankan pada CPU sedangkan bagian-bagian paralel akan dijalankan pada GPU. Bagian-bagian paralel yang dijalankan pada GPU disebut sebagai kernel. (Vazquez el al., 2009)

Gambar 2.13 Arsitektur Fermi

[Diambil dari http://techgage.com/articles/nvidia/fermi/, 4 Maret 2012, 22.25]

Gambar 2.14 *Streaming Multiprocessor* pada Arsitektur Fermi

[Diambil dari http://www.geforce.com, 4 Maret 2012, 22.33]

Pada arsitektur fermi terdapat tiga jenis memori yang dapat diakses oleh GPU/core/thread yaitu *per-thread local memory*, *per-block shared memory*, dan *global memory*. Masing-masing memori memiliki kapasitas penyimpanan data serta kecepatan komunikasi data yang berbeda. Urutan kapasitas penyimpanan data secara berturut-turut dimulai dari yang paling besar adalah *global memory*, *per-block shared memory*, dan *per-thread local memory*. Sedangkan urutan kecepatan komunikasi data secara berturut-turut dimulai dari yang paling cepat adalah *per-thread local memory*, *per-block shared memory*, dan *global memory*. Karena kapasitas penyimpanan yang berbeda-beda, programmer tidak bisa langsung menempatkan semua data di *per-thread local memory* sehingga untuk mendapatkan *speed-up* yang optimal programmer harus dapat mengkombinasikan pengguanaan memori sesuai dengan ukuran data yang akan diproses. (Vazquez et al., 2009)

Terdapat beberapa fungsi *built-in* pada CUDA yang dapat digunakan oleh programmer. Fungsi-fungsi dasar CUDA yang banyak digunakan antara lain adalah: cudaMalloc, cudaMemcpy, dan cudaFree. Fungsi cudaMalloc dan cudaFree mirip dengan fungsi malloc dan free biasa. Hanya saja, fungsi malloc meminta agar komputer menyediakan sejumlah memori di harddisk untuk digunakan nanti sedangkan cudaMalloc meminta GPU untuk menyediakan sejumlah memori tersebut. Begitu pula dengan cudaFree, fungsi free akan membebaskan memori yang dialokasikan pada harddisk sedangkan cudaFree akan membebaskan memori yang dialokasikan pada GPU. Fungsi cudaMemcpy berguna untuk mengirim data dari CPU ke GPU untuk kemudian diolah di GPU dan mengambil data yang telah diolah dari GPU ke CPU untuk kemudian ditampilkan ke pengguna/programmer. (NVIDIA, 2007)

Dalam GPU terdapat hierarki grid, block, dan thread. Grid merupakan kumpulan dari beberapa block dan block merupakan kumpulan dari beberapa thread. Jumlah block dalam suatu grid dapat diatur dengan fungsi gridDim dan jumlah thread dalam suatu block dapat diatur dengan fungsi blockDim. Masing-masing block pada suatu grid memiliki indeks pembeda yang disebut block indeks begitu pula dengan thread, masing-masing thread pada suatu block memiliki indeks pembeda yang disebut thread indeks. (NVIDIA, 2007)

Pada CUDA, semua perintah yang mengandung block indeks dan thread indeks akan langsung dikerjakan secara paralel oleh thread dengan thread indeks dan block indeks yang bersesuaian. Fungsi block indeks dinyatakan dengan blockIdx.x dan thread indeks dinyatakan dengan threadIdx.x. Jika programmer ingin menerapkan thread indeks lebih dari satu dimensi maka programmer dapat menerapkan threadIdx.y dan threadIdx.z begitu pula dengan block indeks. (Sanders dan Kandrot, 2008)

Misalkan seorang programmer ingin mengolah suatu array *A* dengan jumlah elemen dalam array sebanyak delapan. Jika array *A* tersebut dibagi menjadi dua block maka pada GPU akan terdapat block 0 dan block 1 dengan blockDim 4. Pada masing-masing block akan terdapat thread 0-3.

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

Block 0

Thread 0	Thread 1	Thread 2	Thread 3

Block 1

Thread 0	Thread 1	Thread 2	Thread 3

Disini, jika dimasukan perintah $A[i]$ = blockIdx.x maka array *A* akan menjadi:

0	0	0	0	1	1	1	1

Jika dimasukan perintah $A[i]$ = threadIdx.x maka array *A* akan menjadi:

0	1	2	3	0	1	2	3

Dan jika dimasukan perintah $A[i]$ = blockIdx.x*blockDim.x + threadIdx.x maka array *A* akan menjadi:

0	1	2	3	4	5	6	7

Untuk mendapatkan hasil yang diinginkan seorang programmer harus dapat menggunakan blockIdx, blockDim, dan threadIdx dengan cermat.

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

BAB 3
ALGORITMA MCL PARALEL DAN KERNEL PENDUKUNGNYA

3.1 Algoritma MCL Paralel

Proses MCL dapat dijalankan secara paralel. Hal ini bisa dilakukan karena tidak ada *data dependency* pada proses-proses MCL yang berlangsung pada satu kolom. Akan tetapi proses pada tiap-tiap kolom tidak dapat dilaksanakan bersamaan secara paralel karena terbentur masalah *data dependency* pada proses penyimpanan dan perkalian sparse matriks ELL-R. Oleh karena itu proses MCL yang dibuat akan melakukan *loop* serial pada tiap-tiap kolom dan melaksanakan proses paralel dalam setiap *loop*.

Tabel 3.1 Algoritma MCL Paralel

Input : Matriks *M*, threshold, energy

Output : Matriks *M* dimana elemen-elemennya telah ter-*cluster*

1. **for** j = 0..(n col-1)

2. **do** kompresi(*M*(:, j)) **in parallel**

//to get *Mcomp* (*M* in ELL-R format) after full loop

end for

3. **while** energy > threshold

4. **for** j = 0..(n col-1)

5. **do** expand(*Mcomp*, *M*(:, j)) **in parallel**

6. **do** inflate(*M*(:, j), *r*) **in parallel**

//expand and inflate to get new *M*

7. **do** find chaos(*M*(:, j)) **in parallel** //to get new energy after full loop

8. **do** kompresi(*M*(:, j)) **in parallel** //to get new *Mcomp* after full loop

end for

end while

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

Fungsi kompresi, fungsi *expand*, fungsi *inflate*, dan fungsi *find chaos* kesemuanya dijalankan secara paralel dengan GPU/CUDA. Seperti telah dijelaskan pada Subbab 2.5 bahwa bagian yang dijalankan dengan GPU disebut kernel, untuk selanjutnya fungsi kompresi, *expand*, *inflate*, dan *find chaos* akan disebut kernel kompresi, ekspansi, inflasi, dan *find chaos*. Fungsi prune yang berguna untuk mempercepat proses konvergensi merupakan fungsi optional yang bisa diterapkan bisa pula tidak. Seandainya fungsi prune diterapkan maka fungsi ini akan dijalankan sekaligus pada kernel kompresi. Kegunaan serta ide paralel pada setiap kernel akan dibahas pada subbab berikutnya. Alasan mengapa fungsi prune diterapkan pada kernel kompresi juga akan dijelaskan pada subbab berikutnya.

3.2 Metode Penyimpanan ELL-R Paralel dan Prune Paralel

Seperti dijelaskan sebelumnya pada Subbab 2.2, pada metode penyimpanan ELL-R matriks representasi akan disimpan menjadi tiga vektor yaitu vektor data yang berisi informasi tentang nilai elemen-elemen tak-nol, vektor kolom yang berisi informasi letak kolom dari elemen-elemen tak-nol yang bersesuaian, dan vektor srow yang berisi jumlah elemen tak-nol pada tiap-tiap baris. Karena itu, poin penting yang harus dimiliki algoritma ini adalah mampu tidaknya algoritma yang dibuat untuk menemukan semua elemen tak-nol dari matriks representasi dan mampu tidaknya algoritma ini untuk menempatkan informasi dari elemen tak-nol yang didapat ke vektor-vektor yang sesuai.

Pada metode penyimpanan ELL-R proses penyimpanan elemen antar baris berlangsung secara independen tetapi proses penyimpanan elemen pada antar kolom tidak. Hal ini dikarenakan elemen-elemen tak-nol di kolom-kolom sebelumnya akan mempengaruhi nilai vektor srow. Karena itu metode penyimpanan ini tidak bisa dilakukan sepenuhnya secara paralel. Akan tetapi untuk mempercepat proses dapat dilakukan *looping* serial untuk tiap-tiap kolom dan di dalam setiap *loop* tersebut dapat dilakukan pengecekan perbaris secara paralel.

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

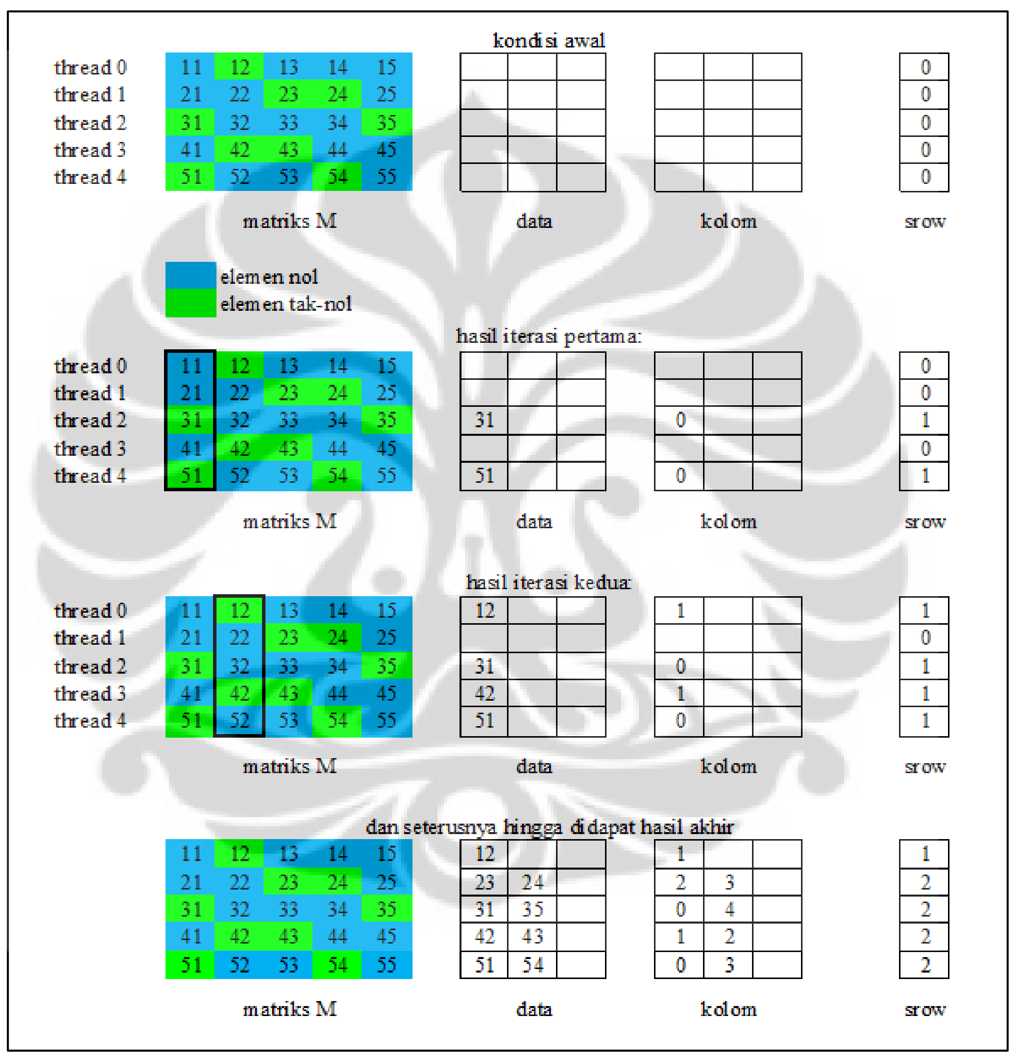
Karena pada bahasa pemrograman C ukuran array harus ditetapkan sebelumnya maka harus diperkirakan ukuran array yang sesuai untuk array data dan array kolom. Nilai yang digunakan untuk mendefinisikan ukuran array harus dipertimbangkan dengan baik karena array yang terlalu besar akan menyebabkan pemborosan memori dan array yang terlalu kecil memiliki resiko array tidak dapat menyimpan semua data. Besar ukuran array yang dipilih untuk array data dan array kolom adalah jumlah baris dari matriks representasi dikali dengan 30. Nilai 30 dipilih karena umumnya suatu protein paling banyak berinteraksi dengan 15-30 protein lain (Satuluri et al., 2010). Untuk array srow ukuran yang dibutuhkan cukup sejumlah baris dari matriks representasi.

Tabel 3.2 Algoritma Kernel Kompresi

Input : Vektor $M(:,j)$, *tolerance*
Output : Vektor data, kolom, dan srow
1. **if** $j = 0$ **do**
2. Each thread i where $i = 0..(n\ row-1)$ **do in parallel**
3. srow[i] = 0
end do
end if
4. Each thread i where $i = 0..(n\ row-1)$ **do in parallel**
5. **if** $M(i, j) >$ toleransi **do**
6. **set** pos = srow[i]* n row + i
7. data[pos] = $M(i, j)$
8. kolom[pos] = j
9. srow[i]+=1
end if
end do

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

Algoritma pada Tabel 3.2 adalah algoritma kompresi untuk melakukan pengkompresan dengan format penyimpanan ELL-R. Akan tetapi algoritma ini hanya bekerja untuk satu vektor saja sehingga untuk mengkompres satu matriks diperlukan *loop* tambahan sejumlah kolom dari matriks yang akan dikompres. Dalam hal ini *loop* yang dibutuhkan telah tersedia pada program utama(algoritma pada Tabel 3.1).



Gambar 3.1 Ilustrasi Proses Kompresi ELL-R

Pada kondisi awal GPU akan menerima matriks M sebagai input. Selanjutnya GPU akan menyediakan tiga vektor kosong: vektor data, vektor kolom, dan vektor srow. Karena matriks M memiliki lima baris maka proses paralel pada kernel ini akan dilakukan dengan menggunakan lima thread. Sebelum

masuk pada iterasi pertama setiap thread akan bekerja untuk mengisi vektor srow sehingga vektor srow yang awalnya adalah vektor kosong menjadi vektor nol. Selanjutnya pada iterasi pertama akan diperiksa kolom pertama dari matriks M. Setiap thread akan memeriksa satu elemen dari kolom pertama. Jika elemen yang diperiksa bernilai 0 maka kerja thread tersebut telah selesai. Jika elemen yang diperiksa bernilai lebih besar dari nol/toleransi maka thread tersebut akan menambah nilai pada array srow yang bersesuaian dengan 1 dan kemudian mengisi array data dan array kolom yang bersesuaian. Pada iterasi kedua akan diperiksa kolom kedua dengan cara yang sama. Begitu pula pada iterasi-iterasi berikutnya hingga didapat hasil akhir pada iterasi kelima.

Ada dua alasan mengapa operasi prune dilaksanakan sekaligus dengan operasi kompresi. Alasan yang pertama adalah untuk mengurangi jumlah operasi yang dilakukan sehingga diharapkan program dapat berjalan lebih cepat. Jumlah operasi dapat dikurangi karena elemen-elemen yang terambil oleh operasi prune merupakan subset dari elemen-elemen yang terambil oleh operasi kompresi biasa. Sehingga jika operasi prune dilakukan terpisah perlu dilakukan dua kali pengecekan yaitu:

1. **if** $M(i, j) > tolerance$ **then** $M(i, j) = 0$
2. **if** $M(i, j) > 0$ **do** line 6-9

Selain itu, operasi prune dilakukan sekaligus dengan operasi kompresi dengan tujuan agar elemen-elemen yang ter-prune lebih banyak. Elemen-elemen yang ter-prune dapat diharapkan lebih banyak karena pada main program operasi prune dilakukan setalah operasi inflasi. Setelah operasi inflasi elemen-elemen yang lemah (bernilai kecil) akan semakin lemah dan elemen yang kuat (bernilai besar) akan semakin kuat sehingga diharapkan setelah operasi inflasi elemen-elemen yang nilainya lebih kecil dari toleransi semakin banyak sehingga proses MCL bisa berlangsung lebih cepat.

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

Seperti dikatakan sebelumnya bahwa operasi prune adalah operasi opsional. Jika operasi prune tidak digunakan maka kode pada baris ke-5 cukup diganti menjadi "**if** *M*(i, j) > 0". Seandainya operasi prune tidak diterapkan maka kode pada baris ke-5 ini akan berfungsi untuk mencari elemen tak-nol saja.

Worst case dari proses ini akan terjadi saat matriks yang input berupa matriks tanpa elemen nol. Seandainya *worst case* ini terjadi pada matriks berdimensi n maka proses serial akan membutuhkan n^2 tahap untuk menyelesaikan proses kompresi. Pada program paralel, karena masing-masing thread akan mengolah satu baris maka proses paralel hanya membutuhkan n tahap. Meskipun terlihat bahwa proses paralel memerlukan lebih sedikit tahapan, performa dari kernel kompresi sangat dipengaruhi banyak thread yang tersedia dan besar *blocksize* yang dipilih.

3.3 Kernel Ekspansi

Selanjutnya akan dibahas tentang kernel ekspansi. Pada kernel ekspansi akan dilakukan proses ekspansi secara paralel. Pada kernel ini terdapat dua proses yang akan dilakukan yaitu pencarian kolom pengali dan SpMV paralel ELL-R. Proses pencarian kolom pengali perlu dilakukan karena perkalian sparse matriks dengan sparse matriks sulit dilakukan sehingga untuk melakukan SpMM perlu dilakukan SpMV berulang-ulang.

Tabel 3.3 Algoritma Kernel Ekspansi

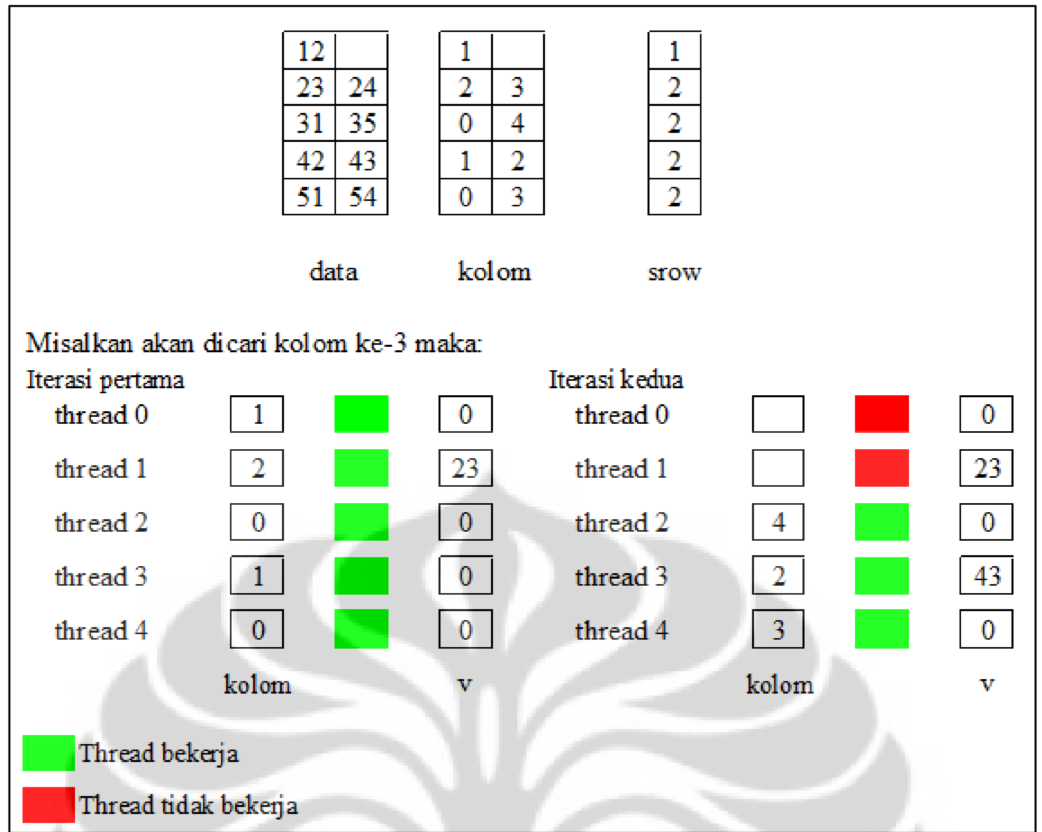
Input : ELL-R sparse matriks *M* (data, kolom, srow)
Output : vektor kolom ke-*j* dari *M**M*
1. Each thread *i* where *i* = 0..(n row-1) **do in parallel**
2.   **if** *i* < n row **do**
3.   **set** maks = srow[*i*]
4.   **set** *k* = 0

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

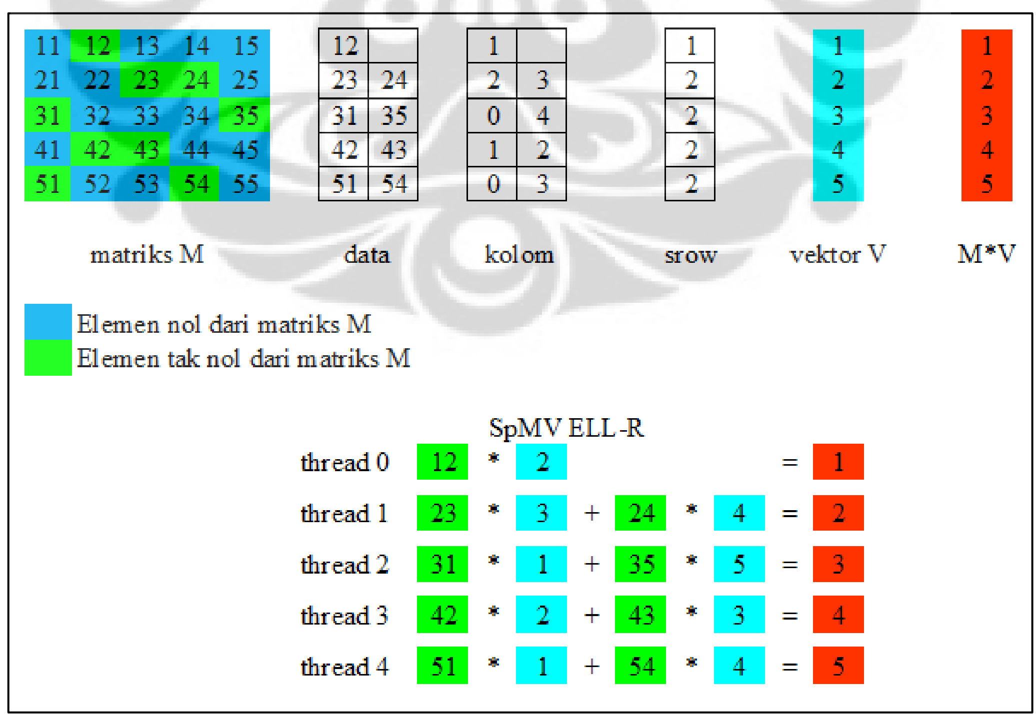
```
5.      while k < maks do
6.        pos = k*n row + i
7.        if kolom[pos] = j do
8.          v[i] = data[pos]
9.          i = maks
10.       else do
11.         v[i] = 0
12.         i+=1
          end if
        end while
      end if
13. set sval = 0
14. for k = 0..(maks-1) do
15.   pos = k*n row + i
16.   sval+=data[pos]*v[kolom[pos]]
    end for
17. y[i] = sval
  end do
```

Pada algoritma ini perintah ke-1 sampai 12 berfungsi untuk mencari kolom ke-*j* pada matriks *M* dari sparse matriks ELL-R *M* sebagai kolom pengali. Bagian yang berfungsi untuk melakukan SpMV antara matriks *M* dengan vektor $M(:, j)$ adalah baris ke-13 sampai 17. Pada algoritma ini masing-masing thread akan mengolah data pada baris yang bersesuaian dan melakukan dua kali *inner-loop*. *inner-loop* pertama (baris ke-5) digunakan untuk proses pencarian kolom pengali dan *inner-loop* kedua (baris ke-14) digunakan untuk proses SpMV ELL-R.

Kernel ekspansi ini hanya memproses satu vektor saja. Dengan kata lain, kernel ini hanya melakukan SpMV saja sehingga untuk melakukan proses ekspansi diperlukan *loop* tambahan agar SpMV pada kernel ini menjadi SpMM. Sama seperti pada Subbab 3.2, *loop* tambahan yang diperlukan telah tersedia pada program utama.



Gambar 3.2 Ilustrasi Proses Pencarian Kolom Pengali

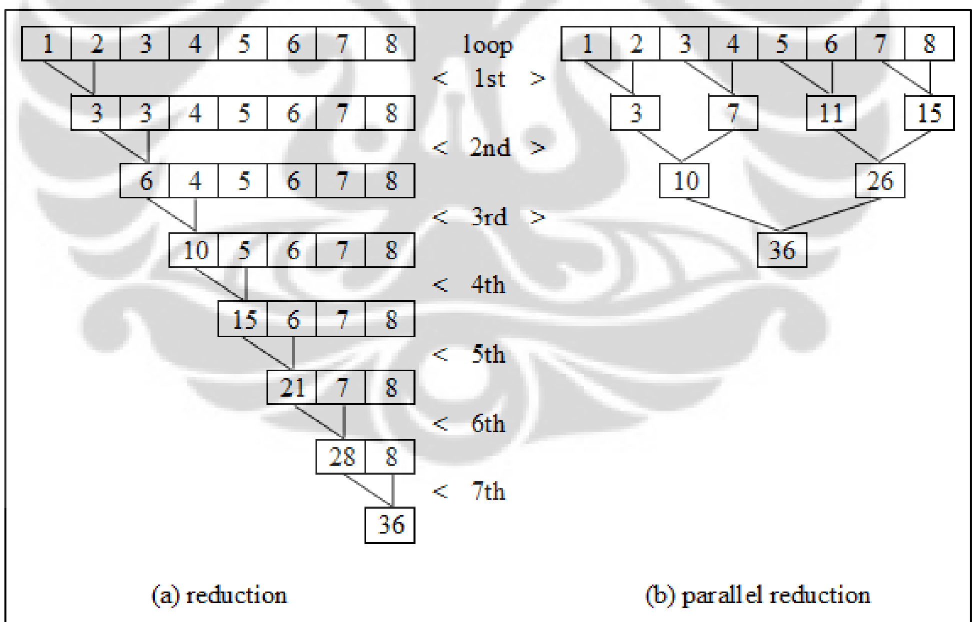


Gambar 3.3 Ilustrasi SpMV ELL-R

Seperti pada subbab sebelumnya, *worst case* pada kernel ini juga terjadi saat matriks yang diproses tidak memiliki elemen nol. Jika hal ini terjadi maka proses serial akan membutuhkan $2n^3 - n^2$ tahap sementara proses paralel membutuhkan $3n^2 - n$ tahap dimana n^2 tahap digunakan untuk mencari kolom pengali dan $2n^2 - n$ digunakan untuk proses SpMM.

3.4 *Reduction* dan *Parallel Reduction*

Sebelum membahas kernel berikutnya akan dijelaskan mengenai konsep *reduction* dan *parallel reduction*. Konsep ini disebut *reduction* karena setelah proses ini dilakukan nilai-nilai pada suatu vektor akan berkurang (*reduced*) menjadi satu nilai tunggal. Konsep ini dapat dilakukan dengan menggunakan operasi komutatif seperti +, *, min, dan maks. Konsep *parallel reduction* adalah *reduction* yang dilakukan secara paralel.



Gambar 3.4 Ilustrasi *Reduction* dan *Parallel Reduction* Dengan Operasi +

Pada skripsi ini data pada tiap-tiap block akan direduksi dengan *parallel reduction*. Hasil dari setiap block kemudian disatukan dengan *reduction*. Seandainya hanya terdapat satu block maka *reduction* tidak dilakukan. Terdapat dua jenis reduksi yang akan digunakan yaitu *sum reduction* dan *max reduction*.

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

3.5 Algoritma Inflasi Paralel

Pada skripsi ini operasi inflasi terdiri dari dua operasi, operasi Hadamard dan operasi *normalize*. Pada operasi Hadamard semua elemen pada kolom yang diproses akan dipangkatkan sesuai dengan parameter inflasi r yang digunakan. Sedangkan pada operasi *normalize* kolom yang diproses akan dinormalisasi sehingga jumlah semua elemen pada kolom tersebut sama dengan satu. Pada operasi *normalize* akan digunakan *parallel sum reduction* seperti yang telah dijelaskan sebelumnya pada Subbab 3.4.

Tebel 3.4 Algoritma Hadamard Paralel

Input : vektor kolom $M(:, j)$, parameter inflasi r
Output : vektor kolom $M(:, j)$
1. Each thread i where i = 0..(n row-1) **do in parallel**
2. $M[i, j]$ = pow($M[i, j]$, r)
end do

Tebel 3.5 Algoritma Normalize Paralel

Input : vektor kolom $M(:, j)$, n blocks
Output : vektor kolom $M(:, j)$ yang telah dinormalisasi
1. **do** *parallel sum reduction* pada $M(:, j)$
2. **if** n blocks > 1 **do** *sum reduction*
end if
3. **set** total = result of 1-2
4. Each thread i where i = 0..(n row-1) **do in parallel**
5. $M[i, j] = M[i, j]$/total
end do

Pada kernel ini proses serial akan membutuhkan $3n^2 - n$ sedangkan proses paralel akan membutuhkan $n\ (2 + {}^2\log blocksize + \lceil n/blocksize \rceil\ )$.

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

3.6 Algoritma *Find Chaos* Paralel

Pada algoritma ini akan dihitung *chaos* pada tiap-tiap kolom. *Chaos* yang didapat kemudian dibandingkan dengan *chaos* sebelumnya. Setelah *for loop* pada program utama selesai dieksekusi akan didapat *global chaos* yang akan digunakan untuk menentukan apakah program berhenti atau melakukan iterasi selanjutnya.

Tebel 3.6 Algoritma *Find Chaos* Paralel

Input : vektor kolom *M*(:, *j*), n blocks, *chaos*
Output : *chaos*
1. **if** *j* = 0 **set** *chaos* = 0
end if
2. **do** *parallel max reduction* pada *M*(:, *j*)
3. **if** n blocks > 1 **do** *max reduction*
end if
4. **set** maks = result of 2-3
5. Each thread *i* where *i* = 0..(n row-1) **do in parallel**
6. *W*[*i*] = pow(*M*(*i*, *j*), 2)
end do
7. **do** *parallel sum reduction* pada *W*
8. **if** n blocks > 1 **do** *sum reduction*
end if
9. **set** sqr = result of 7-8
10. tmpchaos = maks-sqr
11. **if** *chaos* < tmpchaos **set** *chaos* = tmpchaos
end if

Pada kernel ini proses serial akan membutuhkan $3n^2 - 2n$ sedangkan proses paralel akan membutuhkan 2 ($^2\log blocksize + \lceil n/blocksize \rceil$) +1.

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

BAB 4
SIMULASI DAN ANALISA

Pada Bab 3 telah dijelaskan mengenai algoritma MCL paralel beserta kernel-kernel pendukungnya. Pada Bab ini akan diberikan hasil simulasi dari kernel-kernel tersebut. *Running time* yang didapat kemudian dibandingkan dengan *Running time* pada program serial untuk mendapatkan nilai *speed-up*. Karena pemilihan blocksize bisa menghasilkan *running time* yang berbeda-beda maka akan dianalisa *speed-up* dari tiap-tiap kernel. Hasil yang didapat kemudian akan digunakan sebagai dasar untuk pemilihan blocksize secara general.

Program akan dijalankan pada CPU dan GPU dengan spesifikasi sebagai berikut:

CPU:

- Prosesor : Pentium(R) Dual-Core E5500 @2.80 GHz
- Sistem Operasi : Ubuntu 10.04 (lucid)
- Perangkat Lunak : gcc 4.4.3

GPU:

- Mesin : GTX 460
- Arsitektur : Fermi
- CUDA Cores : 336
- Processor Clock : 1.350 MHz
- Perangkat Lunak : gcc 4.4.3 dan CUDA SDK 3.2

Karena Mesin GPU yang digunakan memiliki arsitektur Fermi maka kernel yang dieksekusi maksimal menggunakan blocksize 1024.

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

4.1 Implementasi Kernel Kompresi

Untuk simulasi pada kernel ini dan kernel-kernel selanjutanya matriks yang digunakan adalah matriks random yang dibuat dengan fungsi sprand pada MATLAB. Matriks random tersebut kemudian diubah menjadi matriks biner dan disimpan dalam format txt. Matriks yang sudah disimpan ini kemudian dipanggil kembali untuk proses komputasi serial dan paralel.

Tabel 4.1 Waktu Tunggu (ms) Untuk Proses Kompresi

Dimensi Matriks	CPU	GPU dengan blocksize sejumlah:				
		32	64	128	256	512
500	2,672	20,406	10,684	5,965	4,559	4,881
1.000	20,821	159,798	81,091	43,041	31,272	34,185
2.000	161,242	-	632,763	329,664	237,122	262,448
2.500	354,633	-	-	640,166	460,563	514,456

Karena saat blocksize 512 program telah mengalami penurunan performa maka simulasi untuk blocksize 1.024 tidak dilakukan. Terlihat bahwa program bekerja paling baik saat blocksize 256.

Terlihat bahwa program paralel bekerja paling baik saat blocksize 256. Sayangnya proses ini masih belum berhasil memperoleh *speed-up*. Hal ini disebabkan karena proses transfer data dari CPU ke GPU cukup memakan waktu. Meskipun begitu proses ini tetap dilakukan secara paralel agar tidak perlu dilakukan transfer data dari GPU ke CPU, dan CPU ke GPU lagi. Seandainya proses kompresi dilakukan secara serial maka proses transfer data perlu dilakukan pada setiap *loop* dan menyebabkan perlambatan yang lebih parah.

Walaupun belum menghasilkan *speed-up*, semakin banyak data yang diproses program dapat memiliki perbandingan waktu tunggu yang semakin baik. Diharapkan seandainya data yang diproses semakin banyak program dapat memberikan *speed-up* yang signifikan. Akan tetapi disini simulasi dihentikan pada angka 2.500 karena komputer yang digunakan tidak mampu melakukan proses ekspansi pada data dengan ukuran lebih besar

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

4.2 Implementasi Kernel Ekspansi

Pada Subbab ini perkalian matriks serial yang dibandingkan adalah perkalian matriks serial biasa. Sedangkan perkalian matriks yang dilakukan adalah perkalian matriks ELL-R

Tabel 4.2 Waktu Tunggu (ms) Untuk Proses Ekspansi

Dimensi Matriks	CPU	GPU dengan blocksize sejumlah:				
		32	64	128	256	512
500	15,345	22,746	11,849	6,867	4,835	3,600
1.000	123,217	179,798	91,098	53,081	37,270	28,209
2.000	598,476	-	642,769	429,694	307,122	172,478
2.500	1.721,543	-	-	750,169	465,563	334,476

Terlihat bahwa *speed-up* terbaik dicapai pada saat blocksize 512. Simulasi untuk blocksize 1.024 tidak dilakukan karena pada subbab sebelumnya telah terjadi penurunan performa pada saat blocksize 512.

Tabel 4.3 *Speed-up* Proses Ekspansi

Dimensi Matriks	*Speed-up* 256	*Speed-up* 512
500	3,17	4,26
1.000	3,30	4,36
2.000	1,94	3,46
2.500	3,69	5,14

Pada proses ini waktu tunggu dipengaruhi jumlah elemen baris terbanyak. Jika ada baris dengan jumlah elemen yang lebih banyak dari baris lainnya pada proses sinkronisasi thread-thread lain akan menunggu hingga baris tersebut selesai diolah. Karena itulah jumlah elemen yang semakin banyak belum tentu mempercepat proses. Kecepatan proses lebih ditentukan dengan kondisi matriks yang akan diproses. Semakin sedikit jumlah elemen tak-nol pada tiap-tiap baris proses ekspansi dapat berlangsung lebih cepat.

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

Terlihat bahwa pada proses ekspansi telah muncul *speed-up*. Diharapkan *speed-up* yang dihasilkan dapat memperbaiki waktu tunggu secara keseluruhan setelah terjadi penurunan kecepatan pada proses kompresi.

4.3 Implementasi Kernel Inflasi

Tabel 4.4 Waktu Tunggu (ms) Untuk Proses inflasi

Dimensi Matriks	CPU	GPU dengan blocksize sejumlah:				
		32	64	128	256	512
500	0,283	2,249	1,356	0,767	0,523	0,372
1.000	0,866	4,186	2,090	0,904	0,642	0,436
2.000	18,134	32,584	20,365	13,576	8,205	7,083
2.500	22,388	40,215	26,810	17,186	8,644	8,091

Tabel 4.5 *Speed-up* Proses Inflasi

Dimensi Matriks	*Speed-up* 256	*Speed-up* 512
500	-	-
1.000	1,34	1,98
2.000	2,21	2,56
2.500	2,59	2,76

Terlihat bahwa waktu tunggu tercepat dicapai pada saat blocksize 512. Meskipun begitu, *speed-up* yang didapat saat blocksize 256 juga sudah cukup baik. Disini implementasi hanya dilakukan sampai blocksize 512. Untuk blocksize 1.024 tidak dilakukan karena proses *parallel reduction* yang dibuat masih mengacu pada arsitektur tesla dimana blocksize maksimum adalah 512.

4.4 Implementasi Kernel *Find Chaos*

Pada program yang dibuat kernel *find chaos* (terdapat di lampiran 4) terdiri dari dua kernel. Pada implementasi di subbab ini dua kernel tersebut dijalankan secara berurutan dan kemudian dihitung waktu total saat kedua kernel ini bekerja.

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

Tabel 4.6 Waktu Tunggu (ms) Untuk Proses *Find Chaos*

Dimensi Matriks	CPU	GPU dengan blocksize sejumlah:				
		32	64	128	256	512
500	1,849	20,490	15,076	7,967	5,203	2,759
1.000	4,598	28,186	23,090	10,942	6,942	3,831
2.000	21,402	38,583	31,565	23,577	15,971	14,268
2.500	32,373	45,153	36,801	27,185	20,477	18,081

Tabel 4.7 *Speed-up* Proses *Find Chaos*

Dimensi Matriks	*Speed-up* 256	*Speed-up* 512
500	-	-
1.000	-	1,20
2.000	1,34	1,50
2.500	1,58	1,79

Implementasi hanya dilakukan sampai blocksize 512 karena alasan yang sama seperti pada subbab sebelumnya.

4.5 Analisa MCL Paralel

Pada Subbab ini akan diperkirakan pemilihan blocksize yang dapat memberikan *speed-up* terbaik untuk proses MCL. Implementasi untuk proses MCL paralel secara keseluruhan tidak dapat dilakukan karena terdapat masalah pada sistem. Meskipun setiap kernel telah bekerja dengan baik, saat pengeksekusian kernel kompresi yang kedua (baris kedelapan pada Tabel 3.1) array data, kolom, dan srow yang baru tidak terisi sehingga proses selalu berhenti pada *loop* kedua.

Ada beberapa hal yang layak dipertimbangkan dari Subbab 4.1-4.4. Pada Subbab 4.1 terlihat bahwa kernel kompresi memiliki performa terbaik dengan blocksize 256. Selain itu, untuk jumlah data yang cukup banyak blocksize 256 juga sudah menghasilkan *speed-up* untuk kernel lainnya. Karena dua alasan ini maka diperkirakan proses MCL bisa memperoleh *speed-up* dengan blocksize 256.

Akan tetapi kernel ekspansi, inflasi dan *find chaos* memiliki *speed-up* terbaik dicapai saat blocksize 512. Selain itu juga terlihat bahwa semakin banyak data yang diproses *speed-up* yang diperoleh bisa semakin baik. Karena pada tubuh manusia terdapat lebih dari 50.000 protein, maka diprediksi kernel ekspansi, inflasi, dan *find chaos* dapat memberikan *speed-up* yang lebih baik lagi. Selain itu, karena data yang diolah semakin banyak, diharapkan kernel kompresi juga dapat memberikan kontribusi pada *speed-up* total. Dengan mempertimbangkan hal ini maka diprediksi bahwa pemilihan blocksize 512 akan memberikan *speed-up* terbaik saat program digunakan untuk proses pengelompokan protein sebenarnya.

BAB 5
KESIMPULAN DAN SARAN

Pada skripsi ini telah dibangun algoritma MCL paralel beserta kernel-kernel pendukungnya. Masing –masing kernel telah diimplementesikan dengan gcc dan CUDA SDK. Berdasarkan hasil implementasi diketahui bahwa:

1. Kernel kompresi bekerja paling baik dengan blocksize 256. Untuk matriks dengan dimensi ≤ 2.500 kernel ini belum menghasilkan *speed-up*. Semakin banyak data yang diproses performa kernel menjadi semakin baik.
2. Kernel ekspansi bekerja paling baik dengan blocksize 512. Ketika blocksize 256 dan 512 kernel ini telah memperoleh *speed-up* saat memproses matriks berdimensi 500. Semakin banyak data yang diproses performa kernel menjadi semakin baik.
3. Kernel inflasi bekerja paling baik dengan blocksize 512. Ketika blocksize 256 dan 512 kernel kernel ini mulai memperoleh *speed-up* saat dimensi matriks yang diproses ≥ 1.000. Semakin banyak data yang diproses performa kernel menjadi semakin baik.
4. Kernel *find chaos* bekerja paling baik dengan blocksize 512. Ketika blocksize 256 belum diperoleh *speed-up* saat matriks yang diproses berdimensi 1.000 akan tetapi ketika blocksize 512 kernel ini sudah memperoleh *speed-up* saat matriks yang diproses berdimensi 1.000. Semakin banyak data yang diproses performa kernel menjadi semakin baik.

Berdasarkan hasil implementasi tiap-tiap kernel maka diprediksi pemilihan blocksize 256 sudah dapat menghasilkan *speed-up* untuk proses MCL. Berdasarkan hasil tersebut diprediksi pula bahwa proses MCL paralel akan memperoleh *speed-up* terbaik saat blocksize 512.

Penulis menyadari bahwa masih banyak kekurangan pada skripsi ini. Kekurangan pada skripsi ini antara lain saat pengeksekusian kernel kompresi dimana ada array yang tidak ter-*update*. Hal ini menyebabkan analisa *speed-up* untuk proses MCL paralel secara keseluruhan tidak dapat dilakukan. Hal ini diharapkan dapat dijadikan topik penelitian lebih lanjut khususnya penelitian di Departemen Matematika FMIPA UI. Selain itu proses MCL ini belum diterapkan pada dataset protein yang sebenarnya. Hal ini diharapkan dapat dijadikan topik penelitian gabungan antara Departemen Matematika, Kimia, dan Biologi FMIPA UI.

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

DAFTAR PUSTAKA

Barney, B. (2010). *Introduction to Parallel Computing*. *Retrieved* February 9, 2012 *from* https://computing.llnl.gov/tutorials/parallel_comp/

Bell, N., Garland, M. (2008). *Efficient Sparse Matrix-Vector Multiplication on CUDA*.

Bustamam, A., Burrage, K., Hamilton, N. A. (2010). *Fast Parallel Markov Clustering in Bioinformatics using Massively Parallel Graphics Processing Unit Computing*. Second Intl. Workshop on High Performance Computational System Biology.

Coffin, M. H. (1992). *Parallel Programming*: *a New Approach*. New Jersey: Silicon Press.

Crick, F. H. C. (1970). *Central Dogma of Molecular Biology*. *Journal NATURE* vol. 227, 561-562.

Dongen, S. V. (2008). *Graph Clustering via Discrete Uncoupling Process*. *Journal SIAM* Vol. 30 No. 1, 121-141.

Fountain, T. J. (1994). *Parallel Computing*: *Principles and Practice*. Cambridge: Cambridge University Press.

Hansmann, U. H. E. (2006). *Simulation of Proteins and Protein Interactions*.

NVIDIA. (2007). *CUDA Programmong Guide*. Santa Clara: NVIDIA Corp.

Sanders, J., Kandrot, E. (2008). *CUDA by Example*: *an Introduction to General Purpose GPU Programming*. New Jersey: Addison-Wesley.

Satuluri, V., Parthasarathy, S. (2009). *Scalable Graph Clustering Using Stochastic Flows*: *Application to Community Discovery*.

Satuluri, V., Parthasarathy, S., Ucar, D. (2010). *Markov Clustering of Protein Interaction Networks with Improved Balance and Scalability*.

Shargel, L., Yu, A. B. C. (1999). *Applied Biopharmaceutics and Pharmacokinetics*. New York: Appleton & Lange.

Vázquez, F., Ortega, G., Fernández, J. J., Garzón, E. M. (2010). *Improving the Performance of the Sparse Matrix Vector Product with GPUs*.

Vázquez, F., Garzón, E. M., Martínez, J. A., Fernández, J. J. (2009). *The Sparse Matrix Vector Product on GPUs*.

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

Wang, R. (2003). *Bioinformatics*: *hybridization of Biology and Computer*. *Retrieved* January 7, 2012 *from* http://fourier.eng.hmc.edu/bioinformatics/intro/

Waterman, M. S. (1995). *Introduction to Computational Biology*. London: Chapman & Hall.

Watson, J. D., Crick, F. H. C. (1953). *Molecular Structure of Nucleic Acids*: *A Structure for Deoxyribose Nucleic Acid*. *Journal NATURE* vol. 171, 737-738.

LAMPIRAN

Lampiran 1 *Listing* Program Sparse Generator

```
function c = spgen(n,m)
b = sprandn(n,n,m);
c = full(b);
for i = 1:n
   for j = 1:n
      if c(i,j) ~= 0
         c(i,j) = 1;
      end
   end
end

fid = fopen('spgen.txt','wt');
for i = 1:n
   for j = 1:n
      fprintf(fid, ' %d,', c(i,j));
   end
   fprintf(fid, '\n');
end
fclose(fid);
```

Lampiran 2 *Listing* Program MCL Serial

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>

#define cs 12
#define ds 144

int main(void){
   cudaEvent_t start, stop;
   float time;
   cudaEventCreate(&start);
   cudaEventCreate(&stop);
   float energy = 1;
   float treshold = 1e-3;
   int n = cs;
   const int N = n*n;
   float data[ds]={1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0,
                   1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                   0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
                   0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0,
                   0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0,
                   1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0,
                   1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0,
                   0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0,
                   0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1,
                   1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0,
                   0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1,
                   0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1};
   float dat2[ds];
   float y[cs];
   cudaEventRecord(start,0);
   while (energy>treshold){
      //proses ekspansi
      for (int k = 0; k<n; k++){
         for (int i = 0; i<n; i++){
            y[i]=data[i*n+k];
         }
         for (int i = 0; i<n; i++){
            float mult= 0;
            for (int j = 0; j<n; j++){
               mult+=data[i*n+j]*y[j];
            }
            dat2[i*n+k]=mult;
         }
```


Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

```
        }
        for (int j = 0; j<n; j++){
            float gchaos = 0;
            //proses inflasi
            float sum = 0;
            for (int i = 0; i<n; i++){
                y[i]=pow(dat2[i*n+j],2);
                sum+=y[i];
            }
            for (int i = 0; i<n; i++) y[i] = y[i]/sum;
            for (int i = 0; i<n; i++) data[i*n+j] = y[i];
            //find chaos
            float maks = 0;
            float sqrt = 0;
            for (int i = 0; i<n; i++){
                if (y[i]>maks) maks = y[i];
                sqrt+=pow(y[i], 2);
            }
            float chaos = maks - sqrt;
            if (chaos>gchaos) gchaos = chaos;
            printf("\n%f\n", maks);
            printf("%f\n", sqrt);
            printf("%f\n", chaos);
            printf("%f\n\n", gchaos);
            energy = gchaos;
        }
    }
    cudaEventRecord(stop,0);
    for (int k = 0; k<N; k++){
        printf(" %f,", data[k]);
        if ((k+1)%n == 0) printf("\n");
    }
    printf("last energy: %f treshold: %f\n", energy, treshold);
    cudaEventElapsedTime(&time, start, stop);
    printf("running time: %f ms\n", time);
    getchar();
    return 0;
}
```

//program ini akan melakukan simulasi untuk data interaksi yang sama dengan data interaksi pada Gambar 2.

Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

Lampiran 3 *Listing Main* Program MCL Paralel

```
// includes, system
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

// includes, kernels
#include "mcl2_kernel.cu"

int main(void)
{
  cudaEvent_t start, stop;
  float time;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  float treshold = 1e-3;
  float energy = 1;
  float *input_h, *temp_h, *data_h, *v_h, *y_h, *chaos_h, *ymax_h; int
*column_h, *nrow_h;
// Pointer to host array
  float *input_d, *temp_d, *data_d, *v_d, *y_d, *chaos_d, *ymax_d; int
*column_d, *nrow_d;
// Pointer to device arrays
  float *datat_d; int *columnt_d, *nrowt_d;
  int n = 12;
  const int N = n*n;  // Number of elements in arrays
  int block_size = 256;
  int n_blocks = N/block_size + (N%block_size == 0 ? 0:1); // n_blocks needed
  int sharedMemSize = n_blocks*block_size*sizeof(float);   //shared mem size
needed

  // Allocate array on host
  input_h  = (float *)malloc(sharedMemSize);
  temp_h   = (float *)malloc(sharedMemSize);
  data_h   = (float *)malloc(sharedMemSize);
  v_h      = (float *)malloc(sharedMemSize);
  y_h      = (float *)malloc(sharedMemSize);
  chaos_h  = (float *)malloc(1*sizeof(float));
  ymax_h   = (float *)malloc(1*sizeof(float));
  column_h = (int *)malloc(sharedMemSize);
  nrow_h   = (int *)malloc(sharedMemSize);
  // Allocate array on device
  cudaMalloc((void **) &input_d, sharedMemSize);
  cudaMalloc((void **) &temp_d, sharedMemSize);
  cudaMalloc((void **) &data_d, sharedMemSize);
```

```
  cudaMalloc((void **) &v_d, sharedMemSize);
  cudaMalloc((void **) &y_d, sharedMemSize);
  cudaMalloc((void **) &column_d, sharedMemSize);
  cudaMalloc((void **) &nrow_d, sharedMemSize);
  cudaMalloc((void **) &chaos_d, sharedMemSize);
  cudaMalloc((void **) &ymax_d , sharedMemSize);

  cudaMalloc((void **) &datat_d, sharedMemSize);
  cudaMalloc((void **) &columnt_d, sharedMemSize);
  cudaMalloc((void **) &nrowt_d, sharedMemSize);

//  Initialize host array and copy it to CUDA device
  float data[144]={1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0,
                   1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                   0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
                   0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0,
                   0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0,
                   1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0,
                   1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0,
                   0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0,
                   0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1,
                   1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0,
                   0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1,
                   0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1};

  for (int i = 0; i < N; i++){
    input_h[i] = data[i];
    printf(" %1.3f,", input_h[i]);
    if ((i+1)%n == 0) printf("\n");
  }

  cudaMemcpy(input_d, input_h, sharedMemSize, cudaMemcpyHostToDevice);

  // Do calculation on device:
  for (int col = 0; col < n; col++){
    ellr <<<n_blocks, block_size>>> (input_d, data_d, column_d, nrow_d, n, col,
n_blocks, 1e-3);
    cudaThreadSynchronize();
  }
  // Do calculation on device:
  cudaEventRecord(start, 0);
  while (energy>treshold){
    for (int col = 0; col < n; col++){
      spellr <<<n_blocks, block_size>>> (data_d, column_d, nrow_d, n, col, v_d,
y_d);
      infmcl <<<n_blocks, block_size, sharedMemSize>>> (y_d, temp_d, n, 2,
n_blocks);
```


Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

```
      fymax  <<<n_blocks, block_size, sharedMemSize>>> (y_d, temp_d, ymax_d,
n, n_blocks);
      fchaos <<<n_blocks, block_size, sharedMemSize>>> (y_d, temp_d, n,
n_blocks, col, chaos_d, ymax_d);
      resellr <<<n_blocks, block_size>>> (y_d, datat_d, columnt_d, nrowt_d, n,
col, n_blocks, 1e-3);
      cudaThreadSynchronize();
    }

//  Retrieve result from device
    cudaMemcpy(chaos_h, chaos_d, 1*sizeof(float), cudaMemcpyDeviceToHost);
    datcol <<<n_blocks, block_size>>> (data_d, data2_d, column_d, column2_d,
n);
    nrw   <<<n_blocks, block_size>>> (nrow_d, nrow2_d, n);
    cudaThreadSynchronize();
//  Print results
    printf("%f\n\n", chaos_h[0]);
    energy = chaos_h[0];
  }
  cudaEventRecord(stop, 0);
  cudaEventSynchronize(stop);

  cudaMemcpy(data_h, datat_d, sharedMemSize, cudaMemcpyDeviceToHost);
  cudaMemcpy(column_h, columnt_d, sharedMemSize,
cudaMemcpyDeviceToHost);
  cudaMemcpy(nrow_h, nrowt_d, sharedMemSize, cudaMemcpyDeviceToHost);
//  cudaMemcpy(y_h, y_d, sharedMemSize, cudaMemcpyDeviceToHost);

  // Print results
  for (int i = 0; i < N/2; i++) printf("\n%d %f %d \n", i+1, data_h[i], column_h[i]);
  for (int i = 0; i < n; i++) printf("%d \n", nrow_h[i]);

  cudaEventElapsedTime(&time, start, stop);
  printf ("Running Time parallel: %f ms\n", time);

  // Cleanup
  free(input_h); free(temp_h); free(data_h); free(column_h); free(nrow_h);
free(v_h); free(y_h); free(ymax_h); free(chaos_h);
  cudaFree(input_d); cudaFree(temp_d); cudaFree(data_d); cudaFree(column_d);
cudaFree(nrow_d); cudaFree(v_d); cudaFree(y_d); cudaFree(ymax_d);
cudaFree(chaos_d);
  cudaFree(datat_d); cudaFree(columnt_d); cudaFree(nrowt_d);
}
```

Lampiran 4 *Listing* Kernel Program MCL Paralel

```
#ifndef _MCL2_KERNEL_H_
#define _MCL2_KERNEL_H_

#ifdef __DEVICE_EMULATION__
#define EMUSYNC __syncthreads()
#else
#define EMUSYNC
#endif

#define BS 256

#include <stdio.h>

// Kernel that executes on the CUDA device
__global__ void ellr(float *input, float *data, int *column, int *nrow, int N, int
col, int n_blocks, int en)
{
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

  if (i < N){
    if (col == 0) nrow[i] = 0;
    int val = input[col*N+i];
    if (val > en){
      int newcol = nrow[i];
      int pos = newcol*N+i;
      data[pos] = val;
      column[pos] = col;
      nrow[i]+=1;
    }
  }
}

__global__ void datcol(float *data, float *datat, int *column, int *columnt, int N)
{
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < (N*N)){
    datat[i]  = data[i];
    columnt[i]= column[i];
  }
}

__global__ void nrw(int *nrow, int *nrowt, int N)
{
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < N) nrowt[i] = nrow[i];
```


Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

```
}

// Kernel that executes on the CUDA device
__global__ void spellr(float *data, int *column, int *nrow, int N, int col, float *v,
float *y)
{
 unsigned int row = blockIdx.x*blockDim.x + threadIdx.x;

 if (row < N){
   int maks = nrow[row];
   int i = 0;
   while (i < maks){
     int pos = i*N + row;
     if (column[pos] == col){
       v[row] = data[pos];
       i = maks;
     }
     else{
       v[row] = 0;
       i+=1;
     }
   }
   __syncthreads();

   float sval = 0;
   for (int i=0; i < maks; i++){
     float rowval = 0;
     float colval = 0;
     int idxcol = 0;
     int pos = i*N + row;
     rowval = data[pos];
     idxcol = column[pos];
     colval = v[idxcol];
     sval+=rowval*colval;
   }
   __syncthreads();
   y[row] = sval;
 }
}

// Kernel that executes on the CUDA device
__global__ void infmcl(float *y, float *temp, int N, int r, int n_blocks)
{
 __shared__ float sdata[BS];
 int blocksize = BS;
 unsigned int tid = threadIdx.x;
 unsigned int bid = blockIdx.x;
 unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
```


Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

```
  float totalsum = 0;
  float nz = pow(y[i],r);
  sdata[tid] = (i < N ? nz : 0);
  __syncthreads();

  //1st red in shared mem
  if (blocksize >= 512) {if (tid < 256) {sdata[tid]+= sdata[tid + 256];}
__syncthreads();}
  if (blocksize >= 256) {if (tid < 128) {sdata[tid]+= sdata[tid + 128];}
__syncthreads();}
  if (blocksize >= 128) {if (tid <  64) {sdata[tid]+= sdata[tid +  64];}
__syncthreads();}
  #ifndef __DEVICE_EMULATION__
    if (tid <32)
  #endif
    {
      volatile float *smem = sdata;
      if (blocksize >= 64) {smem[tid]+= smem[tid + 32]; EMUSYNC;}
      if (blocksize >= 32) {smem[tid]+= smem[tid + 16]; EMUSYNC;}
      if (blocksize >= 16) {smem[tid]+= smem[tid +  8]; EMUSYNC;}
      if (blocksize >=  8) {smem[tid]+= smem[tid +  4]; EMUSYNC;}
      if (blocksize >=  4) {smem[tid]+= smem[tid +  2]; EMUSYNC;}
      if (blocksize >=  2) {smem[tid]+= smem[tid +  1]; EMUSYNC;}
    }
  if (tid == 0) temp[bid] = sdata[0];
  __syncthreads();

  if (tid == 0){
    totalsum = 0;
    for (int i = 0; i < n_blocks; i++) totalsum+=temp[i];
  }
  __syncthreads();

  if (i < N) sdata[tid] = totalsum;
  __syncthreads();

  if (i < N) y[i] = nz/sdata[0];
  __syncthreads();
}

// Kernel that executes on the CUDA device
__global__ void fymax(float *y, float *temp, float *ymax, int N, int n_blocks)
{
  __shared__ float sdata[BS];
  int blocksize = BS;
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;
```

```
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

  float colmax = 0;
  float nz = y[i];
  sdata[tid] = (i < N ? nz : 0);
  __syncthreads();

  //1st red in shared mem
  if (blocksize >= 512) {if ((tid < 256) && (sdata[tid] < sdata[tid + 256]))
{sdata[tid] = sdata[tid + 256];} __syncthreads();}
  if (blocksize >= 256) {if ((tid < 128) && (sdata[tid] < sdata[tid + 128]))
{sdata[tid] = sdata[tid + 128];} __syncthreads();}
  if (blocksize >= 128) {if ((tid <  64) && (sdata[tid] < sdata[tid +  64]))
{sdata[tid] = sdata[tid +  64];} __syncthreads();}
  #ifndef __DEVICE_EMULATION__
    if (tid <32)
  #endif
    {
      volatile float *smem = sdata;
      if ((blocksize >= 64) && (smem[tid] < smem[tid + 32])) {smem[tid] =
smem[tid + 32]; EMUSYNC;}
      if ((blocksize >= 32) && (smem[tid] < smem[tid + 16])) {smem[tid] =
smem[tid + 16]; EMUSYNC;}
      if ((blocksize >= 16) && (smem[tid] < smem[tid +  8])) {smem[tid] =
smem[tid +  8]; EMUSYNC;}
      if ((blocksize >=  8) && (smem[tid] < smem[tid +  4])) {smem[tid] =
smem[tid +  4]; EMUSYNC;}
      if ((blocksize >=  4) && (smem[tid] < smem[tid +  2])) {smem[tid] =
smem[tid +  2]; EMUSYNC;}
      if ((blocksize >=  2) && (smem[tid] < smem[tid +  1])) {smem[tid] =
smem[tid +  1]; EMUSYNC;}
    }
  if (tid == 0) temp[bid] = sdata[0];
  __syncthreads();

  if (tid == 0){
    for (int i = 0; i < n_blocks; i++){
     if (colmax < temp[i]) colmax = temp[i];__syncthreads();
    }
    sdata[0] = colmax;
   }
   __syncthreads();
   ymax[0] = sdata[0];
}

// Kernel that executes on the CUDA device
__global__ void fchaos(float *y, float *temp, int N, int n_blocks, int col, float
*chaos, float *ymax)
```

```
{
  __shared__ float sdata[BS];
  int blocksize = BS;
  unsigned int tid = threadIdx.x;
  unsigned int bid = blockIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

  float total = 0;
  float ysqrsum = 0;

  float nz = pow(y[i],2);
  sdata[tid] = (i < N ? nz : 0);
  __syncthreads();

  //1st red in shared mem
  if (blocksize >= 512) {if (tid < 256) {sdata[tid]+= sdata[tid + 256];}
__syncthreads();}
  if (blocksize >= 256) {if (tid < 128) {sdata[tid]+= sdata[tid + 128];}
__syncthreads();}
  if (blocksize >= 128) {if (tid <  64) {sdata[tid]+= sdata[tid +  64];}
__syncthreads();}
  #ifndef __DEVICE_EMULATION__
    if (tid <32)
  #endif
    {
      volatile float *smem = sdata;
      if (blocksize >= 64) {smem[tid]+= smem[tid + 32]; EMUSYNC;}
      if (blocksize >= 32) {smem[tid]+= smem[tid + 16]; EMUSYNC;}
      if (blocksize >= 16) {smem[tid]+= smem[tid +  8]; EMUSYNC;}
      if (blocksize >=  8) {smem[tid]+= smem[tid +  4]; EMUSYNC;}
      if (blocksize >=  4) {smem[tid]+= smem[tid +  2]; EMUSYNC;}
      if (blocksize >=  2) {smem[tid]+= smem[tid +  1]; EMUSYNC;}
    }
  if (tid == 0) temp[bid] = sdata[0];
  __syncthreads();

  if (tid == 0){
    for (int i = 0; i < n_blocks; i++) total+=temp[i];
  }
  __syncthreads();

  if (i < N) sdata[tid] = total;
  __syncthreads();

  ysqrsum = sdata[0];
  if (col == 0) chaos[0] = 0;
  chaos[0] = (chaos[0] < (ymax[0]-ysqrsum) ? ymax[0]-ysqrsum : chaos[0]);
}
```


Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012

```
// Kernel that executes on the CUDA device
__global__ void resellr(float *y, float *datat, int *columnt, int *nrowt, int N, int
col, int n_blocks, int en)
{
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

  if (i < N){
    if (col == 0) nrowt[i] = 0;
    int val = y[i+20];
    if (val > en){
      int newcol = nrowt[i];
      int pos = newcol*N+i;
      datat[pos] = val;
      columnt[pos] = col;
      nrowt[i]+=1;
    }
  }
}

#endif // #ifndef _MCL2_KERNEL_H_
```


Algoritma markov..., Muhammad Fauzan Akbar Masyhudi, FMIPA UI, 2012