Perkalian Matriks dengan Beberapa Algoritma

Adro Anra Purnama[¶], Surya Dharma[*], Fariz Iftikhar Falakh[†], Senggani Fatah Sedayu[§]
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
Bandung, Indonesia
{[¶]13220005, [*]13220027, [†]13220029, [§]13220035}@std.stei.itb.ac.id

***Abstract*—Metode perkalian matriks yang umumnya diketahui adalah dengan perkalian baris kolom (dalam pemrograman perkalian baris kolom disebut *Naive algorithm*). Dengan berkembangnya ilmu matematika dan komputasi, diperoleh beberapa algoritma perkalian matriks yang memakan lebih sedikit waktu dibandingkan *Naive algorithm*). Penulis akan menggunakan beberapa algoritma untuk menghitung hasil perkalian 2 buah matriks dengan ukuran besar. Lalu penulis akan membandingkan kompleksitas waktu dari algoritma-algoritma tersebut dengan menganalisis kompleksitas waktu dan ruang.**

***Index Terms*—matriks, *Naive algorithm*, algoritma Strassen, algoritma Cannon**

I. Pendahuluan

Perkalian matriks merupakan suatu operasi biner dengan operan dua buah matriks yang menghasilkan sebuah matriks. Perkalian matriks memiliki peran penting dalam dunia matematika serta memiliki jangkauan aplikasi yang sangat luas, misalnya *signal processing*). Karena banyaknya kegunaan operasi ini, matematikawan mencoba untuk mencari metode perkalian matriks yang lebih memakan sedikit waktu. Dari usaha tersebut, banyak metode dan algoritma yang dikembangkan agar perkalian matriks—terutama untuk matriks dengan ukuran besar—menjadi lebih efisien.

Algoritma-algoritma yang digunakan oleh penulis adalah *Naive algorithm*, algoritma Strassen, dan algoritma Cannon. Ketiga algoritma tersebut akan diimplementasikan dalam bahasa pemrograman C berdasarkan pseudocode. Penulis akan membandingkan seberapa cepat suatu algoritma menyelesaikan hasil perkalian 2 buah matriks dengan algoritma lain dengan menganalisis kompleksitas waktu algoritma tersebut.

II. Studi Pustaka

A. Naive Algorithm

Naive algorithm seperti namanya, merupakan sebuah algoritma yang sangat polos di mana pada algoritma ini kita akan mengalkulasikan setiap *entry* menjadi sebuah *sum of product*. Karena hal inilah *Naive Algorithm* memiliki lebih banyak perhitungan dibandingkan algoritma-algoritma perkalian matriks lainnya. Pada algoritma, untuk sebuah matriks $n \times n$ yang dikalikan dengan matriks berukuran $n \times n$ lainnya, algoritma ini membutuhkan n^3 perhitungan, sehingga time complexity dari algoritma ini adalah $O(n^3)$.

Algorithm 1: Naive Algorithm

Input: Matriks A, matriks B ukuran $n \times n$
Output: Matriks C ukuran $n \times n$
Procedure
for $i = 0$ **to** n **do**
  for $j = 0$ *to* **to** n **do**
    $C[i][j] = 0$
    for $k = 0$ **to** n **do**
      $C[i][j] = C[i][j] + A[i][k] \cdot B[k][j]$
    end
  end
end

B. Strassen Algorithm

Algoritma Strassen merupakan algoritma untuk mencari hasil perkalian yang lebih cepat dibanding dengan Algoritma naive. Algoritma Naive melakukan perhitungan perkalian matriks biasa yang memerlukan n^3 perhitungan, sedangkan untuk Algoritma Strassen memerlukan $n^{2.81}$ perhitungan. Berkurangnya banyak perhitungan disebabkan oleh penggunaan pengurangan dan penjumlahan pada perhitungan sebelumnya untuk mendapat nilai baru. Karena proses pengurangan dan penjumlahan lebih ringan dibanding dengan perkalian, maka time complexity dari algoritma menurun.

```
function KaliMatriksStrassen(A, B : Matriks, n : integer) → Matriks
{ Memberikan hasil kali matriks A dan B yang berukuran n × n. }
Deklarasi
  i, j, k : integer
  A11, A12, A21, A22, B11, B12, B21, B22,  C11, C12, C21, C22, M1, M2, M3, M4, M5, M6, M7 : Matriks

Algoritma:
  if n = 1 then   { matriks berukuran 1 x 1 atau sebagai scalar }
    return A * B   { perkalian dua buah scalar biasa }
  else
    Bagi A menjadi A11, A12, A21, dan A22 yang masing-masing berukuran n/2 x n/2
    Bagi B menjadi B11, B12, B21, dan B22 yang masing-masing berukuran n/2 x n/2
    M1 ← KaliMatriksStrassen(A12 – A22, B21 + B22, n/2)
    M2 ← KaliMatriksStrassen (A11 + A22, B11 + B22 , n/2)
    M3 ← KaliMatriksStrassen (A11 – A21, B11 + B12 , n/2)
    M4 ← KaliMatriksStrassen (A11 + A12, B22 , n/2)
    M5 ← KaliMatriksStrassen (A11, B12 – B22 , n/2)
    M6 ← KaliMatriksStrassen (A22, B21 – B11 , n/2)
    M7 ← KaliMatriksStrassen (A21 + A22, B11 , n/2)
    C11 ←  M1 + M2 – M4 + M6
    C12 ←  M4 + M5
    C21 ←  M6 + M7
    C22 ← M2 – M3 + M5 – M7
    return C { C adalah gabungan C11, C12, C13, C14 }
  endif
```

Fig. 1. *Pseudocode* algoritma Strassen

C. Cannon Algorithm

Dalam *computer science*, algoritma Cannon adalah algoritma terdistribusi (dirancang untuk dilakukan pada *hardware* komputer yang terdiri dari berbagai prosesor) untuk perkalian matriks dan mesh (jaringan yang terbentuk dari sel dan titik) 2 dimensi. Algoritma ini cocok digunakan untuk mesh N*N (atau matriks N*N) dan tidak cocok digunakan untuk matriks bukan persegi. Modifikasi matriks dengan algoritma Cannon berhubungan dengan geser sirkular (*circular shift*). Geser sirkular adalah operasi menggeser elemen-elemen array (atau matriks), mirip seperti *bit shifting*. Misalnya digunakan kasus geser sirkular kiri dan *bit shiifting* ke kiri. Perbedaannya adalah dengan *bit shiifting*, elemen paling kiri akan hilang, sementara dengan geser sirkular kiri, elemen paling kiri akan dipindahkan ke paling kanan.

Algorithm 2: Algoritma Cannon

Result: Hasil perkalian matriks A berukuran n*n dan matriks B berukuran n*n
Procedure
for $i = 0$ **to** $n - 1$ **do**
  Geser sirkular kiri $baris[i]$ A i kali
end
for $i = 0$ **to** $n - 1$ **do**
  Geser sirkular atas $kolom[i]$ B i kali
end
for $k = 0$ **to** $n - 1$ **do**
  for $i = 0$ **to** $n - 1$ **do**
    for $j = 0$ **to** $n - 1$ **do**
      $C[i][j] \leftarrow C[i][j] + A[i][j] \times B[i][j]$
    end
  end
  Geser sirkular kiri semua baris A 1 kali
  Geser sirkular atas semua baris B 1 kali
end

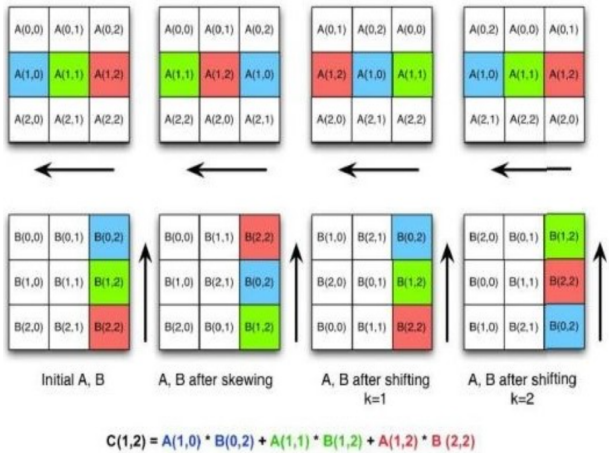

Fig. 2. Ilustrasi algoritma Cannon

D. Kompleksitas

Ada banyak hal yang harus dipertimbangkan dalam membuat sebuah program. Salah satunya adalah cara/algoritma implementasinya yang bergantung pada ukuran input. Untuk membandingkan performa beberapa algoritma, digunakan analisis asimtotik. Ada 3 jenis notasi untuk menyatakan kompleksitas algoritma, yaitu notasi Θ (theta), notasi *Big O*, dan notasi Ω (omega).

Notasi *Big O* menyatakan batas atas asimtotik dari kompleksitas suatu algoritma. Notasi Ω menyatakan batas bawah asimtotik dari kompleksitas suatu algoritma. Notasi Θ merupakan gabungan notasi *Big O* dan Notasi Ω. Notasi yang biasanya digunakan adalah notasi *Big O*.

Definisi *Big O* secara formal adalah ada suatu fungsi T yang memenuhi $T(n) = O(f(n))$ jika terdapat suatu konstanta c dan N_0 sehingga $T(n) \leq c \cdot f(n)$ untuk semua $n \geq N_0$. Dalam penulisan notasi, pengguna hanya fokus pada bagian dominan dari sebuah fungsi kompleksitas. Misal terdapat sebuah fungsi $T(n) = n^2 + n + 1$. Karena n^2 tumbuh lebih cepat dibandingkan dengan n dan 1 untuk input n yang makin besar, maka notasi *Big O* dari $T(n)$ adalah $O(n^2)$. Cara lain untuk menentukan *Big O* dari $T(n)$ adalah dengan menggunakan definisi notasi *Big O*. Pilih $f(n) = n^2$, $c = 2$, dan $N_0 = 2$, maka $T(n)$ akan selalu lebih kecil dari $c \cdot f(n)$ untuk semua $n \geq N_0$ sehingga diperoleh notasi *Big O* dari $T(n)$ yaitu $O(n^2)$.

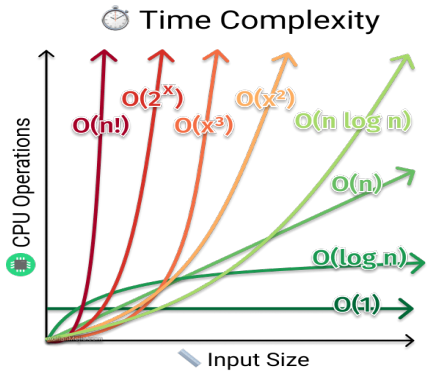

Fig. 3. Kurva perbandingan kompleksitas waktu suatu algoritma

III. Metodologi Penelitian

Proses penelitian ini terdiri dari 3 tahap. Pertama, dilakukan studi literatur dengan mencari dan membaca referensi yang berkaitan. Setelah melakukan studi literatur, dilakukan implementasi algoritma dalam bahasa pemrograman C. Implementasi akan direvisi berulang kali sampai hasil yang diinginkan sesuai dengan algoritma. Tahap terakhir dari penelitian adalah pemaparan kode yang telah dibuat dan berhasil dijalankan ke dalam laporan.

IV. Implementasi dan Pengujian

A. Implementasi Naive Algorithm dalam Bahasa C

Untuk mengimplementasikan *Naive algorithm* pada bahasa pemrograman C, diperlukan beberapa prosedur, yaitu :

deklarasi matriks, pengisian nilai matriks, pencetakan matriks sebelum dikalikan, perkalian matriks, serta pencetakan matriks hasil perkalian.

1) Deklarasi Matriks: Deklarasi matriks merupakan sebuah prosedur di mana user bisa memilih ukuran dari matriks yang ingin dikalikan. Lalu karena ukurannya yang belum pasti serta range ukuran matriks yang bisa menjadi sangat besar, maka digunakan *dynamic memory*.

2) Pengisian nilai matriks: Prosedur ini akan mengisi nilai matriks yang sebelumnya telah kita deklarasikan ukurannya dengan menggunakan fungsi random, di mana program akan mengisi setiap *cell* pada matriks dengan nilai random yang berkisar di range 0-9.

3) Cetak Matriks Sebelum Dikalikan: Fungsi dari prosedur ini sebenarnya hanya untuk pengecekan manual perkalian matriks. Prosedur ini akan mencetak nilai-nilai dari kedua matriks yang sebelumnya telah diisi secara random oleh program.

4) Perkalian Matriks: Prosedur ini akan mencari nilai dari perkalian dua matriks yang memiliki ukuran yang sama ($n \times n$) dengan menggunakan metode *Naive algorithm*.

5) Cetak Matriks Hasil Perkalian: Prosedur ini akan mencetak hasil perkalian dua matriks yang sebelumnya telah dikalikan pada prosedur perkalian matriks.

B. Implementasi Strassen Algorithm dalam Bahasa C

Dalam implementasinya *Strassen Algorithm* membutuhkan 8 fungsi yaitu:

1) Find closest Exponent: Mencari kelipatan pangkat dua yang terdekat dengan dimensi matriks. Hal ini dilakukan karena algoritma strassen melakukan perhitungannya dengan membaginya menjadi submatriks dengan kelipatan 2. Apabila ukuran matriks bukan kelipatan dua maka nilai dari matriks yang kosong akan diisi dengan angka 0.

2) Fill Matrix: Mengisi matriks dengan nilai random dengan range 0-9.

3) Matrix sum: Fungsi ini akan mengembalikan hasil penjumlahan dari dua matriks.

4) Substract Matrix: Fungsi ini akan mengembalikan hasil pengurangan dari dua matriks.

5) Extract Matriks: Algoritma *Strassen* membagi masing-masing matriks yang akan dikalikan menjadi empat submatriks.

6) Print Matrix: Prosedur ini akan mencetak hasil perkalian matriks.

7) Free Struct: Prosedut ini akan membebaskan memori yang digunakan untuk menyimpan matriks yang digunakan agar tidak terjadi *memory leak*.

8) Strassen: Fungsi ini adalah fungsi utama yang menggabungkan semua fungsi dan prosedur yang sudah dideklar untuk menjalankan *Strassen Algorithm*.

C. Implementasi Cannon Algorithm dalam Bahasa C

Ada 7 prosedur yang digunakan untuk mengimplementasikan algoritma Cannon, 1 untuk inisialisasi matriks, 1 untuk mencetak isi matriks (untuk keperluan *debugging*), 4 untuk memodifikasi elemen matriks, dan 1 untuk mengisi matriks hasil. Inisialisasi matriks pada dasarnya mengisi 2 matriks dengan nilai dalam rentang 0 - 9. Pengisian matriks hasil diimplementasikan dengan *nested loop* indeks baris dan kolom, lalu elemen matriks C baris ke-i kolom ke-j didapatkan dari perkalian elemen matriks A baris ke-i kolom ke-j dan elemen matriks B baris ke-i kolom ke-j.

Prosedur yang digunakan untuk memodifikasi elemen matriks adalah geser sirkular kiri baris, geser sirkular atas kolom, geser sirkular kiri matriks, dan geser sirkular atas matriks. Geser sirkular kiri baris diimplementasikan dengan menyimpan elemen pertama array (misalnya di variabel $temp$), lalu dilakukan *looping* untuk menukar elemen array. Setelah itu, elemen array paling terakhir akan disubstitusikan dengan nilai dari $temp$. Geser sirkular atas kolom diimplementasikan dengan cara yang mirip, hanya berbeda pada parameter fungsi dan pengaksesan array saja.

Geser sirkular kiri matriks diimplementasikan dengan menyimpan elemen kolom 0 (misalnya di array arr_temp), lalu dilakukan *looping* untuk menukar elemen setiap baris. Setelah itu, kolom terakhir matriks akan diisi oleh array arr_temp sesuai indeksnya. Geser sirkular atas matriks diimplementasikan dengan cara yang mirip, perbedaannya adalah geser sirkular kiri menyimpan elemen kolom 0, sedangkan geser sirkular atas menyimpan elemen baris 0.

Kompleksitas waktu dari implementasi program algoritma Cannon adalah $O(n^3)$, sedangkan kompleksitas ruangnya adalah $O(n^2)$. Pengujian menunjukkan bahwa algoritma Cannon tidak lebih cepat dan efisien dari *Naive algorithm*. Hal ini disebabkan oleh pengimplementasian algoritma Cannon yang kurang sesuai. Algoritma Cannon dibuat untuk digunakan sebagai algoritma terdistribusi (dilakukan pada komputer yang terdiri dari berbagai prosesor). Dengan mengimplementasikan algoritma Cannon secara paralel dan mengatur beberapa faktor seperti jumlah prosesor yang digunakan, algoritma Cannon dapat menjadi lebih efisien dari *Naive algorithm*.

D. Source Code Program

Implementasi ketiga algoritma ini dalam bahasa pemrograman C dapat dilihat di github.com/Fariz06/Tugas-5-PMC.

V. Kesimpulan

Kesimpulannya pusing tujuh keliling

References

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.

[6] K. Srikanth, "Cannon's algorithm for distributed matrix multiplication," OpenGenus IQ, 13-Oct-2018. [Online]. Available: https://iq.opengenus.org/cannon-algorithm-distributed-matrix-multiplication/. [Accessed: 07-May-2022]

[7] A. Gupta and V. Kumar, “Scalability of Parallel Algorithms for Matrix Multiplication,“ 1993 International Conference on Parallel Processing - ICPP’93, 1993, pp. 115-123, doi: 10.1109/ICPP.1993.160.