Makalah Komputasi Paralel

“Project UTS Komputasi Paralel: Perkalian Matriks-Matriks dan Matriks-Vektor”

Disusun Oleh:

- Fakhri Perdana (1906352060)
- Richardy Lobo’ Sapan (1906373954)
- Sulthan Ali Pasha (1906304244)

Disusun dalam rangka memenuhi tugas

Mata Kuliah Komputasi Paralel

Dosen Pengampu: Alhadi Bustaman, Ph.D

Fakultas Matematika dan Ilmu Pengetahuan Alam

Universitas Indonesia

Maret, 2022

ABSTRAK

Matriks adalah kumpulan bilangan yang disusun secara baris atau kolom atau kedua-duanya dan di dalam suatu tanda kurung. Bilangan-bilangan yang membentuk suatu matriks disebut sebagai elemen-elemen matriks. Matriks digunakan untuk menyederhanakan penyampaian data, sehingga mudah untuk diolah.

Kata Kunci: Matrix, Sparse, Dense, Vektor.

DAFTAR ISI

BAB I

PENDAHULUAN

1. Latar Belakang

Dalam istilah Matematika, terdapat istilah Matriks. Matriks adalah kumpulan bilangan yang disusun secara baris atau kolom atau kedua-duanya dan di dalam suatu tanda kurung. Bilangan-bilangan yang membentuk suatu matriks disebut sebagai elemen-elemen matriks. Matriks digunakan untuk menyederhanakan penyampaian data, sehingga mudah untuk diolah. Notasi suatu matrik berukuran n x m ditulis dengan huruf besar dan dicetak tebal, misalnya An×m. Huruf n menyatakan jumlah baris, dan huruf m jumlah kolom. Matriks banyak digunakan untuk menyelesaikan berbagai permasalahan matematika, seperti dalam proses pemrograman untuk menyelesaikan suatu masalah.

2. Rumusan Masalah

Berdasarkan Latar Belakang tersebut, rumusan masalah yang akan digunakan dalam pembentukan makalah ini adalah:

1. Bagaimana cara melakukan perhitungan Matriks dalam Komputasi Paralel?
2. Bagaimana algoritma untuk melakukan perkalian matriks dense dan sparse?

3. Tujuan Penulisan

Tujuan yang ingin dicapai dari penulisan makalah ini adalah:

1. Memahami cara melakukan perkalian dalam matriks secara paralel
2. Memahami algoritma dalam melakukan perkalian matriks dense dan sparse

4. Manfaat Penulisan

Manfaat yang diharapkan didapat oleh penulisan makalah ini adalah:

1. Mendapatkan pengetahuan tentang cara melakukan perkalian secara paralel
2. Memperoleh pengetahuan mengenai algoritma untuk melakukan perkalian matriks dense dan sparse.

BAB II

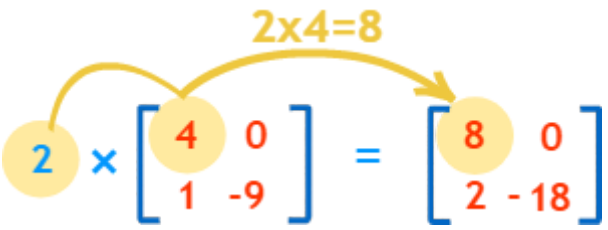
PEMBAHASAN

2.1 Konsep Dasar Perkalian Matriks

Matriks adalah susunan bilangan dalam bentuk segi empat seperti berikut:

$$\begin{bmatrix} 6 & 4 & 24 \\ 1 & -9 & 8 \end{bmatrix}$$

Untuk mengalikan matriks dengan satu angka itu mudah:



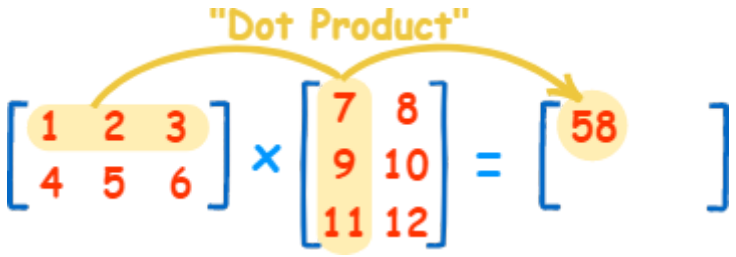
Ini adalah perhitungannya:

2×4=8	2×0=0
2×1=2	2×-9=-18

Kita menyebut angka ("2" dalam hal ini) skalar, jadi ini disebut "perkalian skalar".

Tetapi untuk mengalikan matriks dengan matriks lain kita perlu melakukan "Dot Product" dari baris dan kolom. Apa artinya? Mari kita lihat dengan sebuah contoh:

Untuk mencari jawaban untuk baris ke-1 dan kolom ke-1:

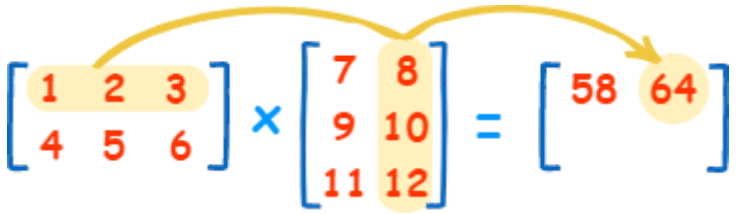


"Dot Product" adalah tempat kita mengalikan anggota yang cocok, lalu jumlahkan:

$$(1, 2, 3) \bullet (7, 9, 11) = 1\times7 + 2\times9 + 3\times11$$

$$= 58$$

Kita mencocokkan anggota pertama (1 dan 7), lalu mengalikannya. Demikian juga untuk anggota ke-2 (2 dan 9) dan anggota ke-3 (3 dan 11), dan akhirnya menjumlahkannya. Berikut contohnya untuk baris ke-1 dan kolom ke-2.



$$(1, 2, 3) \bullet (8, 10, 12) = 1\times8 + 2\times10 + 3\times12$$

$$= 64$$

Kita dapat melakukan hal yang sama untuk baris ke-2 dan kolom ke-1:

$$(4, 5, 6) \bullet (7, 9, 11) = 4\times7 + 5\times9 + 6\times11$$

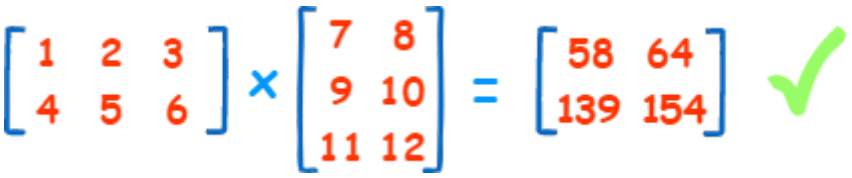
$$= 139$$

Dan untuk baris ke-2 dan kolom ke-2:

$$(4, 5, 6) \bullet (8, 10, 12) = 4\times8 + 5\times10 + 6\times12$$

$$= 154$$

Dan kita mendapatkan:



Saat kita melakukan perkalian:

- Banyaknya kolom matriks ke-1 harus sama dengan jumlah baris matriks ke-2.
- Dan hasilnya akan memiliki jumlah baris yang sama dengan matriks ke-1, dan jumlah kolom yang sama dengan matriks ke-2.

Matriks yang sebagian besar berisi nilai nol disebut sparse, berbeda dengan matriks yang sebagian besar nilainya bukan nol, disebut padat.

Matriks sparse besar umum terjadi pada umumnya dan khususnya dalam pembelajaran mesin terapan, seperti dalam data yang berisi hitungan, pengkodean data yang memetakan kategori ke hitungan, dan bahkan di seluruh subbidang pembelajaran mesin seperti pemrosesan bahasa alami.

Secara komputasi mahal untuk mewakili dan bekerja dengan matriks sparse seolah-olah mereka padat, dan banyak peningkatan kinerja dapat dicapai dengan menggunakan representasi dan operasi yang secara khusus menangani sparitas matriks.

Dense Matrix

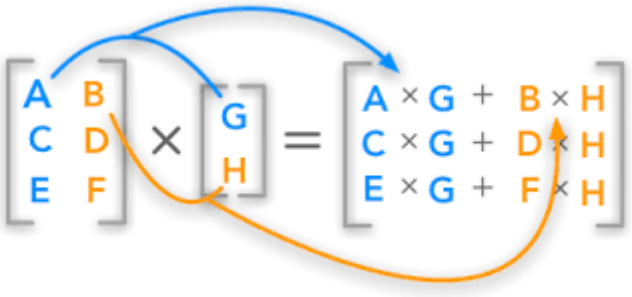
1	2	31	2	9	7	34	22	11	5
11	92	4	3	2	2	3	3	2	1
3	9	13	8	21	17	4	2	1	4
8	32	1	2	34	18	7	78	10	7
9	22	3	9	8	71	12	22	17	3
13	21	21	9	2	47	1	81	21	9
21	12	53	12	91	24	81	8	91	2
61	8	33	82	19	87	16	3	1	55
54	4	78	24	18	11	4	2	99	5
13	22	32	42	9	15	9	22	1	21

Sparse Matrix

1	.	3	.	9	.	3	.	.	.
11	.	4	.	.	.	.	.	2	1
.	.	1	.	.	.	4	.	1	.
8	.	.	.	3	1	.	.	.	.
.	.	.	9	.	.	1	.	17	.
13	21	.	9	2	47	1	81	21	9
.	.	.	.	.	.	.	.	.	.
.	.	.	.	19	8	16	.	.	55
54	4	.	.	.	11	.	.	.	.
.	.	2	.	.	.	.	22	.	21

2.2 Perkalian Matriks Dense

2.2.1 Perkalian Matriks-Vektor



- Program Sekuensial

```
10  int matvec(float **matrix, float *vector, float *result,
11  int size_i, int size_j) {
12      int i;
13      int j;
14      for(i=0;i<size_i;i++) {
15          result[i]=0;
16          for(j=0;j<size_j;j++) {
17              result[i] += matrix[i][j]*vector[j];
18          }
19      }
20      return 0;
21  }
22
```

Di sini kita dapat melihat bahwa kode ini mengimplementasikan perkalian vektor-matriks yang melakukan penjumlahan dari perkalian parsial vektor dan baris matriks. Mari kita bahas di sini pola akses memori yang berfokus pada baris 14 sampai 19. Pertama-tama lihat baris 17, di mana kita telah mendapatkan pola akses utama ke variabel array result, matrix dan vector. Akses dilakukan dengan indeks loop berikut: result dengan i, matrix dengan i dan j, dan vector dengan j. Jadi, antara matriks dan vektor kita memiliki akses memori yang serupa di dimensi terakhir yang diindeks oleh j. Karena j adalah indeks dari innermost loop, pola akses baris-utama ini sempurna untuk kode C karena mengeksploitasi lokalitas memori. Akses memori lain yang terlibat adalah result[i], di mana i adalah indeks dari outmost loop. Karena dalam kasus ini ada akses baris-utama ke array satu dimensi, kita juga tidak akan memiliki masalah lokalitas di sini.

- Program Paralel

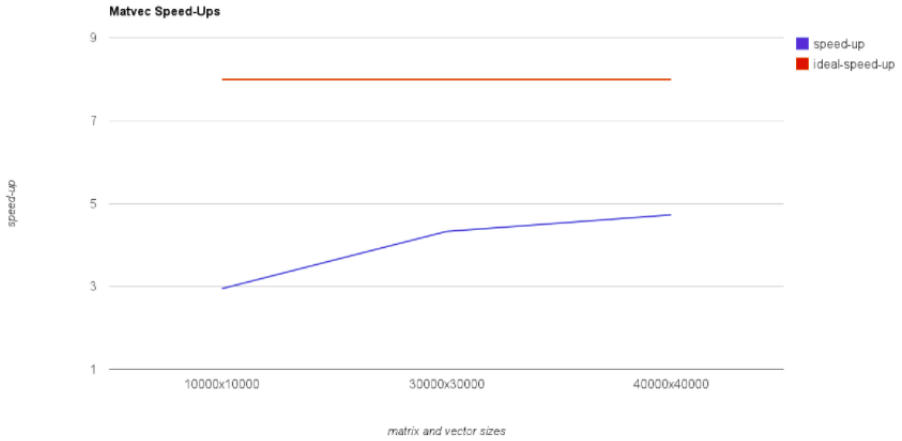
```
10  int matvec(float** matrix, float* vector, float* result,
11  int size_i, int size_j)
12  {
13     int i,j;
14  #pragma omp parallel shared(matrix,result,vector) private(i,j)
15     {
16  #pragma omp for  schedule(static)
17     for (i=0; i<size_i; i=i+1){
18        result[i]=0.;
19        for (j=0; j<size_j; j=j+1){
20           result[i]=(result[i])+((matrix[i][j])*(vector[j]));
21        }
22     }
23     }
24     return 0;
25  }
26
```

Perhatikan bahwa variabel matrix dan vector bersifat read-only. Dengan demikian, kita dapat membuatnya shared karena tidak akan muncul race conditions pada variabel-variabel tersebut selama eksekusi paralel. Sekarang perhatikan variabel result, yang merupakan perhitungan penting dalam algoritma ini. Variabel ini selalu diindeks oleh i, indeks outermost loop. Jadi jika kita mendistribusikan iterasi dalam loop ini, menjadikan i private dan membuat result shared, thread tidak akan pernah menulis di posisi yang sama dengan result.

Perbandingan

Setelah memiliki kode paralel, kita coba menjalankannya dan melakukan pengukuran waktu untuk ukuran matriks yang berbeda. Kita seharusnya memperoleh kinerja yang baik dari kode ini. Dengan mesin dengan 8 thread, kita peroleh hasil berikut.

MATRIX AND VECTOR SIZE	SEQUENTIAL TIME (SEC.)	PARALLEL TIME(SEC.)	SPEED-UP
10000×10000	0.10	0.03	2.95
30000×30000	1.01	0.23	4.33
40000×40000	1.88	0.39	4.73



Hasil percobaan yang ditunjukkan di atas menggunakan 4 core dengan hyperthreading, yang membuat total 8 thread, dengan 2 thread per core. Speed-up ideal adalah sekitar 8. Tampak dari perhitungan running time bahwa program paralel jelas memiliki waktu yang lebih cepat dibandingkan yang sekuensial.

2.2.2 Perkalian Matriks-Matriks

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

Program ini adalah program perkalian matriks sederhana untuk OpenMP

- Program Sekuensial

```c
int main()
{
    /* Call function di sini*/
    return 0;
}

int alg_matmul2D(int m, int n, int p, float **a, float **b,
 float **c)
{
    int i, j, k ;
    for (i=0; i<m; i++)
    {
        for (j=0; j<n; j++)
        {
            a[i][j]=0;
            for (k=0; k<p; k++)
                a[i][j] += b[i][k]*c[k][j] ;
        }
    }
    return 0;
}
```

Code ini memperkenalkan lebih banyak kompleksitas daripada kode yang dianalisis pada makalah sebelumnya tentang komputasi paralel PI dan komputasi paralel produk matriks-vektor. Masalah utama datang dari akses memori yang tidak mengeksploitasi lokalitas data. Lebih khusus lagi, perhitungan PI hanya menggunakan variabel skalar, tidak ada penggunaan array besar. Produk matriks-vektor menggunakan array yang diakses dengan mempertahankan lokalitas memori. Di sini, di produk matriks-matriks, kita menangani array dua dimensi yang tidak diakses dengan menjaga lokalitas data, dan dengan demikian menunjukkan masalah memori berikut yang berdampak pada kinerja.

Dalam kode yang disajikan di atas, kita melihat bahwa kita sedang mengimplementasikan produk matriks dasar yang mengakumulasikan produk baris dan kolom ke dalam matriks hasil a. Saat kita mengalikan baris dan kolom, masalah utama pada kode ini muncul. Bagaimana kita bisa melakukan akses memori yang dioptimalkan pada saat yang sama untuk matriks b dan c? Asumsikan bahwa matriks disimpan dalam memori dalam urutan baris-utama. Kita mengakses matriks a dengan baris dan matriks b dengan kolom, jadi kita menggunakan akses memori antagonis di sini. Poin kuncinya adalah jika kita mengoptimalkan satu akses memori matriks, kita mendapatkan akses memori yang buruk di matriks lainnya. Hal yang sama berlaku untuk penyimpanan kolom-utama.

- Program Paralel

Terlepas dari masalah memori dan lokalitas, bagaimana kita bisa mendapatkan kinerja yang lebih baik dari kode ini? Jawabannya adalah membuatnya paralel. Di sini kita dapat melihat implementasi paralel menggunakan standar OpenMP:

```
int alg_matmul2D(int m, int n, int p, float** a, float** b,
 float** c)
{
   int i,j,k;
#pragma omp parallel shared(a,b,c) private(i,j,k)
   {
#pragma omp for  schedule(static)
   for (i=0; i<m; i=i+1){
      for (j=0; j<n; j=j+1){
         a[i][j]=0.;
         for (k=0; k<p; k=k+1){
            a[i][j]=(a[i][j])+((b[i][k])*(c[k][j]));
         }
      }
   }
   }
   return 0;
}
```

Pada dasarnya, kita telah memparalelkan loop terluar yang mendorong akses ke matriks hasil a di dimensi pertama. Jadi, terdapat pembagian kerja di antara thread sedemikian rupa sehingga thread yang berbeda akan menghitung baris yang berbeda dari matriks hasil a. Perhatikan bahwa thread yang berbeda akan menulis bagian yang berbeda dari hasil dalam larik a, jadi kita tidak mendapatkan masalah selama eksekusi paralel. Perhatikan bahwa akses ke matriks b dan c bersifat read-only dan juga tidak menimbulkan masalah. Dalam strategi paralelisasi, kita membagikan semua matriks tugas dan semuanya akan berfungsi dengan baik; sebagai tambahan, semua indeks loop diprivatisasi karena setiap thread perlu melakukan iterasi loopnya sendiri.

Perbandingan

Matrix Size	Sequential Time (Sec.)	Parallel Time(Sec.)	Speed-Up
250×250	0.022967	0.011298	2.032838
500×500	0.172636	0.058255	2.963454
1000×1000	7.618695	2.086783	3.650928
2000×2000	66.446484	19.163739	3.467303

Hasil percobaan yang ditunjukkan di atas menggunakan 4 core dengan hyperthreading, yang membuat total 8 thread, dengan 2 thread per core. Speed-up ideal adalah sekitar 5 dan didapatkan kecepatan maksimal mendekati 3,5, yang memaparkan masalah memori produk matriks-matriks dan dampaknya terhadap kinerja. Hasil ini memberi kita alasan yang baik untuk menggunakan paralelisme karena kita akan mendapatkan waktu eksekusi yang lebih baik, meningkatkan produktivitas, mengurangi biaya, dan mengurangi waktu yang terbuang.

2.3 Perkalian Matriks Sparse

Matriks yang sebagian besar berisi nilai nol disebut sparse, berbeda dengan matriks yang sebagian besar nilainya bukan nol, disebut padat.

Secara komputasi, sangat merugikan untuk mewakili dan bekerja dengan matriks sparse seolah-olah mereka adalah matriks padat, dan banyak peningkatan kinerja dapat dicapai dengan menggunakan representasi dan operasi yang secara khusus menangani matriks sparse. Keunikan utama pada matriks sparse adalah eksploitasinya dapat menghasilkan penghematan komputasi yang sangat besar dan karena banyak masalah matriks berukuran besar yang terjadi dalam praktisnya itu berbentuk sparse.

Sparsitas suatu matriks dapat dikuantifikasi dengan skor, yaitu banyaknya nilai nol dalam matriks dibagi dengan jumlah total elemen dalam matriks. Sebagai contoh:

```
sparsity = count zero elements / total elements
```

Below is an example of a small 3 x 6 sparse matrix.

```
     1, 0, 0, 1, 0, 0
A = (0, 0, 2, 0, 0, 1)
     0, 0, 0, 2, 0, 0
```

Contoh memiliki 13 nilai nol dari 18 elemen dalam matriks, memberikan matriks ini skor sparsity 0,722 atau sekitar 72%.

Space Complexity

Matriks yang sangat besar membutuhkan banyak memori, dan beberapa matriks yang sangat besar yang ingin kita kerjakan adalah sparse.

Contoh matriks yang sangat besar yang terlalu besar untuk disimpan di memori adalah matriks link yang menunjukkan tautan dari satu situs web ke situs web lainnya.

Contoh matriks Sparse yang lebih kecil mungkin berupa matriks kemunculan kata atau istilah untuk kata-kata dalam satu buku dibandingkan semua kata yang dikenal dalam bahasa Inggris.

Dalam kedua kasus, matriks yang terkandung jarang dengan lebih banyak nilai nol daripada nilai data. Masalah dengan mewakili matriks jarang ini sebagai matriks padat adalah bahwa memori diperlukan dan harus dialokasikan untuk setiap nilai nol 32-bit atau bahkan 64-bit dalam matriks.

Ini jelas merupakan pemborosan sumber daya memori karena nilai nol tersebut tidak mengandung informasi apa pun.

Time Complexity

Dengan asumsi matriks sparse yang sangat besar dapat dimasukkan ke dalam memori, kita akan ingin melakukan operasi pada matriks ini.

Sederhananya, jika matriks sebagian besar berisi nilai nol, yaitu tidak ada data, maka melakukan operasi di seluruh matriks ini mungkin memakan waktu lama di

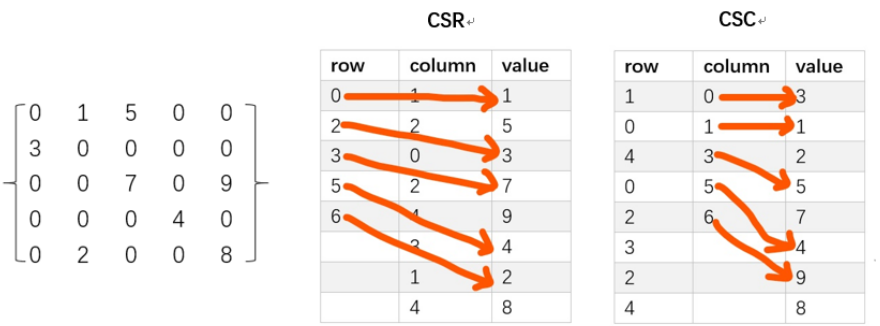
mana sebagian besar komputasi yang dilakukan akan melibatkan penambahan atau perkalian nilai nol bersama-sama.

Solusi untuk mewakili dan bekerja dengan matriks sparse adalah dengan menggunakan struktur data alternatif untuk mewakili data sparse. Nilai nol dapat diabaikan dan hanya data atau nilai bukan nol dalam matriks sparse yang perlu disimpan atau ditindaklanjuti. Ada beberapa struktur data yang dapat digunakan untuk membangun matriks sparse secara efisien; tiga contoh umum tercantum di bawah ini.

- Dictionary of Keys. Dictionary digunakan di mana indeks baris dan kolom dipetakan ke suatu nilai.
- List of Lists. Setiap baris matriks disimpan sebagai daftar, dengan setiap subdaftar berisi indeks kolom dan nilainya.
- Coordinate List. Daftar tupel disimpan dengan setiap tupel berisi indeks baris, indeks kolom, dan nilainya.

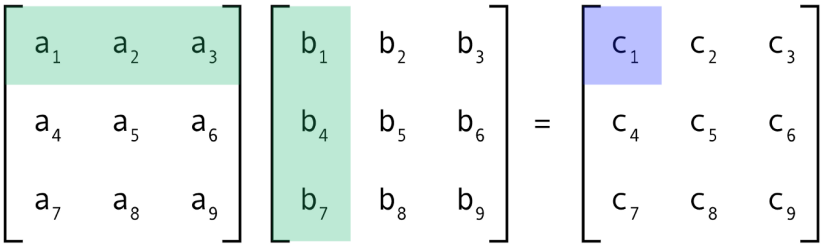
Ada juga struktur data yang lebih cocok untuk melakukan operasi yang efisien; dua contoh yang umum digunakan tercantum di bawah ini.

- Compressed Sparse Row. Matriks sparse direpresentasikan menggunakan tiga array satu dimensi untuk nilai bukan nol, luas baris, dan indeks kolom.
- Compressed Sparse Column.. Sama dengan metode Compressed Sparse Row kecuali indeks kolom dikompres dan dibaca terlebih dahulu sebelum indeks baris.



Compressed Sparse Row, juga disebut CSR, sering digunakan untuk mewakili matriks sparse dalam machine learning mengingat akses efisien dan perkalian matriks yang didukungnya.

2.3.1 Perkalian Matriks-Matriks



Pada kesempatan kali ini, kami akan memberi contoh mengenai perkalian Matriks-Matriks Sparse dengan penggabungan struktur csr dan csc

```
import numpy as np
from scipy.sparse import csc_matrix
from scipy.sparse import csr_matrix
```

Yang pertama dilakukan adalah memanggil modul yang kita butuhkan, pada kasus ini adalah modul matriks sparse dengan struktur csr dan csc.

```
baris_A = np.array([0, 0, 1, 2 ])
kolom_A = np.array([0, 1, 0, 1])
data_A = np.array([5, 9, 6, 7])

matrikscsc = csc_matrix((data_A, (baris_A, kolom_A)),
           shape = (3, 3))

print("Matriks csc: \n", matrikscsc.toarray())

baris_B = np.array([0, 1, 1, 2 ])
kolom_B = np.array([0, 0, 1, 0])
data_B = np.array([1, 8, 3, 4])

matrikscsr = csr_matrix((data_B, (baris_B, kolom_B)),
           shape = (3, 3))

print("Matriks csr:\n", matrikscsr.toarray())
```

Langkah selanjutnya adalah pembuatan matriks csr dan csc yang akan dihitung. Pada kesempatan ini akan menghitung matriks 3 x 3 untuk keduanya, dengan perintah baris dan kolom memberi penjelasan mengenai posisi nilai pada matriks 3 x 3, dan perintah data memberitahukan mengenai nilai yang akan dimasukkan pada matriks.

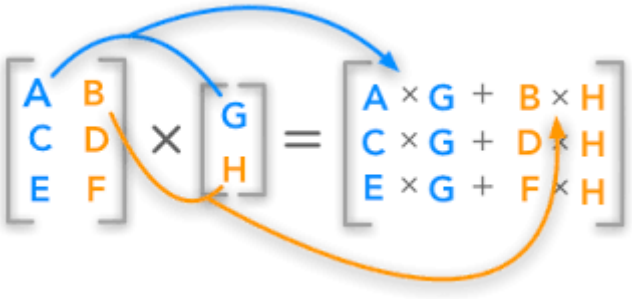
```
sparseMatrix = matrikscsc.multiply(matrikscsr)

print("Produk hasil csc dengan matriks csr:\n",
  sparseMatrix.toarray() )

sparseMatrix = matrikscsr.multiply(matrikscsc)

print("Produk hasil csr dengan matriks csc:\n",
  sparseMatrix.toarray() )
```

Langkah terakhir adalah melakukan operasi perkalian, baik antara matriks csr dengan matriks csc, ataupun kebalikannya.

```
Matriks csc:
 [[5 9 0]
 [6 0 0]
 [0 7 0]]
Matriks csr:
 [[1 0 0]
 [8 3 0]
 [4 0 0]]
Produk hasil csc dengan matriks csr:
 [[ 5  0  0]
 [48  0  0]
 [ 0  0  0]]
Produk hasil csr dengan matriks csc:
 [[ 5  0  0]
 [48  0  0]
 [ 0  0  0]]
```

Berikut merupakan *Output* yang kita dapatkan dari program yang telah kita *Input* diatas.

2.3.2 Perkalian Matriks Vektor



Disini akan dijelaskan mengenai Perkalian Matriks Vektor menggunakan Python.

Berikut adalah programnnya :

```
import numpy as np
from scipy.sparse import csr_matrix
```

Masukkan terlebih dahulu package yang dibutuhkan

```
row_A = np.array([0, 0, 1, 2 ])
col_A = np.array([0, 1, 0, 1])
data_A = np.array([4, 3, 8, 9])

csrMatrix_A = csr_matrix((data_A, (row_A, col_A)),
            shape = (3, 3))


print("Sparse Matrix : \n", csrMatrix_A.toarray())
```

lalu kita buat sparse matriksnya

```
row_B = np.array([0])
col_B = np.array([0])
data_B = np.array([7])

Vector_B = csr_matrix((data_B, (row_B, col_B)
            shape = (3, 1))


print("Vector :\n", Vector_B.toarray())
```

kemudian masukkan vectornya

```
sparseMatrix_AB = csrMatrix_A.multiply(Vector_B)

print("Product Sparse Matrix:\n",sparseMatrix_AB.toarray() )
```

selanjutnya kita kalikan, didapat outputnya seperti berikut :

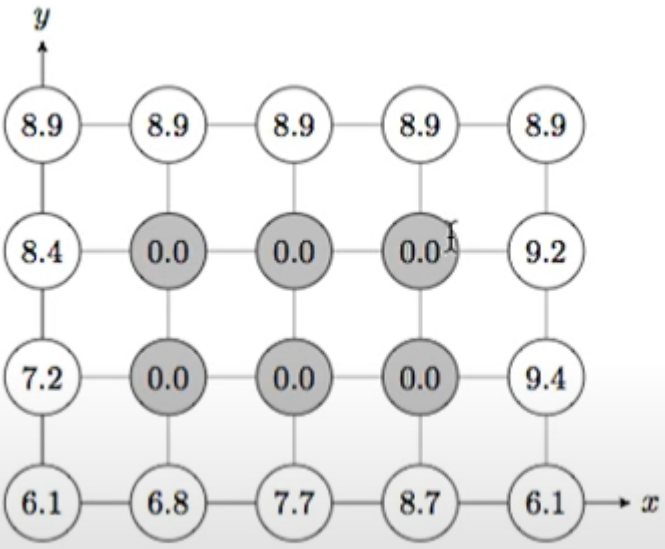
```
Sparse Matrix :
 [[4 3 0]
 [8 0 0]
 [0 9 0]]
Vector :
 [[7]
 [0]
 [0]]
Product Sparse Matrix:
 [[28 21  0]
 [ 0  0  0]
 [ 0  0  0]]
```

2.4 Tugas Pribadi

2.4.1 Tugas Pribadi Fakhri Perdana - Solve Laplace Equation using Jacobi Method

Pada tugas pribadi saya ini, saya akan menggambarkan secara mudah dan simpel bagaimana menyelesaikan Laplace Equation dengan Jacobi Method. Program ini dijalankan menggunakan OpenMP-Like Python atau bisa disebut sebagai OpenMP versi Python.

Tujuan dari Program ini adalah untuk mencari nilai nol pada gambar di bawah ini.



The values of the initial conditions and boundary nodes for solving Laplace's equation.

Yang mana hasil dari nilai nol itu adalah Solver dari Laplace Equation.gambar ini hanya sebagai ilustrasi, untuk programnya saya akan menggunakan input yang berbeda dengan mengubah node menjadi matrix.

dengan nilai nolnya dapat dicari dengan Rumus

$$u_{i,j}^{k+1} = \frac{u_{i,j+1}^{k} + u_{i+1,j}^{k} + u_{i,j-1}^{k} + u_{i-1,j}^{k}}{4}.$$

berikut adalah programnya

```
import pymp
from time import perf_counter

nx = 100
ny = 100


sol = pymp.shared.array((nx,ny))
soln = pymp.shared.array((nx,ny))

for j in range(0,ny-1):
    sol[0,j] = 10.0
    sol[nx-1,j] = 1.0


for i in range(0,nx-1):
    sol[i,0] = 0.0
    sol[i,ny-1] = 0.0
```

pertama - tama kita masukkan terlebih dahulu input datanya, yaitu matrix ukuran berapa (M_nx, M_ny), setelah itu kita masukkan nilai pada batas luar matrix, dan nilai nol pada bagian dalam matrix, seperti gambar yang sebelumnya.

setelah itu kita hitung

```
print(sol)
# Iterasi
start_time = perf_counter()
with pymp.Parallel(2) as p:
  for kloop in range(1,1000):
    soln = sol.copy()

    for i in p.range(1,nx-1):
      for j in p.range (1,ny-1):
        sol[i,j] = 0.25 * (soln[i,j-1] + soln[i,j+1] + soln[i-1,j] + soln[i+1,j])


end_time = perf_counter()
print(sol)

print('Elapsed wall clock time = %g seconds.' % (end_time-start_time) )
```

```
[[10. 10. 10. ... 10. 10. 10.]
 [10.  0.  0. ...  0.  0. 10.]
 [10.  0.  0. ...  0.  0. 10.]
 ...
 [10.  0.  0. ...  0.  0. 10.]
 [10.  0.  0. ...  0.  0. 10.]
 [10. 10. 10. ... 10. 10. 10.]]
[[10.         10.         10.         ... 10.         10.
  10.        ]
 [10.          9.98643348  9.97288813 ...  0.          0.
  10.        ]
 [10.          9.97288813  9.94581879 ...  0.          0.
  10.        ]
 ...
 [10.          0.          0.         ...  9.94581879  9.97288813
  10.        ]
 [10.          0.          0.         ...  9.97288813  9.98643348
  10.        ]
 [10.         10.         10.         ... 10.         10.
  10.        ]]
Elapsed wall clock time = 6.67107 seconds.
```

untuk 2 thread, membutuhkan waktu 6.67 detik

```
# Iterasi
start_time = perf_counter()
with pymp.Parallel(4) as p:
  for kloop in range(1,1000):
    soln = sol.copy()

    for i in p.range(1,nx-1):
      for j in p.range (1,ny-1):
        sol[i,j] = 0.25 * (soln[i,j-1] + soln[i,j+1] + soln[i-1,j] + soln[i+1,j])


end_time = perf_counter()
print('Elapsed wall clock time = %g seconds.' % (end_time-start_time) )
```

```
Elapsed wall clock time = 3.59785 seconds.
```

untuk 4 thread, membutuhkan waktu 3,56 detik

```
# Iterasi
start_time = perf_counter()
with pymp.Parallel(8) as p:
  for kloop in range(1,1000):
    soln = sol.copy()

    for i in p.range(1,nx-1):
      for j in p.range (1,ny-1):
        sol[i,j] = 0.25 * (soln[i,j-1] + soln[i,j+1] + soln[i-1,j] + soln[i+1,j])


end_time = perf_counter()
print('Elapsed wall clock time = %g seconds.' % (end_time-start_time) )
```

```
Elapsed wall clock time = 2.17031 seconds.
```

untuk 8 thread membutuhkan waktu 2,17 detik.

2.4.2 Tugas Pribadi Richardy Lobo' Sapan - Gaussian Blur dengan PyMP

Untuk tugas pribadi ini, saya akan memperlihatkan proyek tentang image convolution, dalam hal ini, akan diperlihatkan tentang Gaussian Blurring. versi paralel diimplementasikan menggunakan PyMP, model paralelisasi fork-join lainnya, yang menyediakan fungsionalitas seperti OpenMP dengan Python.ersi paralel diimplementasikan menggunakan PyMP, model paralelisasi fork-join lainnya, yang menyediakan fungsionalitas seperti OpenMP dengan Python. PyMP ini mendukung sebagian besar fungsi yang didukung oleh OpenMP.

Pada image processing, Gaussian Blur (juga dikenal Gaussian Smoothing) adalah hasil blur gambar dengan fungsi Gaussian. Penggunaannya banyak pada software graphics, terutama dalam hal mengurangi image noise dan mengurangi detail. Gaussian blurring merupakan proses konvolusi. Proses konvolusi, secara dasar menghitung untuk setiap pixel, nilai baru dengan menambahkan nilai berbobot dari pixel tetangga. representasi matematikanya sebagai berikut.

$$R = \sum_{i=-1}^{1} \sum_{j=-1}^{1} P_{x+i,y+j} \; S_{1+i,1+j}$$

di mana P adalah matriks gambar, S adalah mask atau kernel yang diaplikasikan, dan R adalah matriks hasil. Kernel yang biasa digunakan untuk Gaussian Blurring ditunjukkan di bawah.

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Ide proyek ini adalah melakukan operasi konvolusi untuk setiap piksel secara paralel dan membandingkannya dengan eksekusi serial yang sama. Kita pertama-tama menerapkan Gaussian blur pada gambar untuk menghilangkan noise yang tidak perlu. Langkah ini dapat dilakukan dengan mask over konvolusi di atas gambar. Kita menggunakan total 4 thread untuk memeriksa seluruh gambar, yaitu gambar dibagi menjadi empat bagian dan setiap thread bekerja pada satu bagian secara paralel. Di PyMP, ini dapat dicapai dengan menggunakan:

```
with pymp.Parallel(2) as p1:
    with pymp.Parallel(2) as p2:
        for i in p1.range(0, img.width):
            for j in p2.range(0, img.length):
            #convolution operations
```

Berikut kita akan melihat bagaimana bagian ini ditambahkan pada program serial. Sebelum itu, kita perlu meng-install module pymp-pypi dan Pillow terlebih dahulu untuk dapat menjalankan program.

```
[5] !pip install pymp-pypi
```

```
Requirement already satisfied: pymp-pypi in /usr/local/lib/python3.7/dist-packages (0.5.0)
```

```
[6] !pip install Pillow
```

```
Requirement already satisfied: Pillow in /usr/local/lib/python3.7/dist-packages (7.1.2)
```

- Program serial

Import dahulu library dan package yang akan digunakan, diantaranya ada numpy, scipy, PIL (Pillow), imageio, pymp, dan datetime.

```
[34] import numpy as np
     from numpy import array
     from scipy import misc
     from PIL import Image
     import imageio
     import pymp
     import datetime
```

Setelah itu, kita buat program serialnya sebagai berikut.

```
a = datetime.datetime.now()
pymp.config.nested = True
face = imageio.imread('coin,png')
print (face.shape)
convx = array([[1/16, 2/16, 1/16],
    [2/16, 4/16, 2/16],
    [1/16, 2/16, 1/16]])
l=face.shape[0]
b=face.shape[1]
padded = np.zeros((l+2,b+2))
for i in range(0,l):
  for j in range(0,b):
    padded[i+1][j+1]=face[i][j]


res = np.zeros((l,b), dtype='uint8')
i=None
j=None

for i in range(1,l+1):
  for j in range(1,b+1):
    res[i-1][j-1] = (convx[0][0]*padded[i-1][j-1] + convx[0][1]*padded[i-1][j]+convx[0][2]*padded[i-1][j+1]+
        convx[1][0]*padded[i][j-1]+convx[1][1]*padded[i][j] + convx[1][2]*padded[i][j+1]+
        convx[2][0]*padded[i+1][j-1]+ convx[2][1]*padded[i+1][j] +convx[2][2]*padded[i+1][j+1])
```

Terakhir, print output berupa gambar dan waktu eksekusi.

- Program paralel

Sama seperti sebelumnya, kita import dahulu library dan package yang dibutuhkan.

```
import numpy as np
from numpy import array
from scipy import misc
from PIL import Image
import imageio
import pymp
import datetime
```

Setelah itu, dengan merujuk pada program serial sebelumnya, dan juga merujuk pada cara memparalelisasi operasi konvolusi yang juga sudah diperlihatkan di atas, kita bisa memperoleh program sebagai berikut.

```
a = datetime.datetime.now()
pymp.config.nested = True
face = imageio.imread('coin.jpg')
print (face.shape)
convx = array([[1/16, 2/16, 1/16],
    [2/16, 4/16, 2/16],
    [1/16, 2/16, 1/16]])
l=face.shape[0]
b=face.shape[1]
#padded = np.zeros((l+2,b+2))
padded = pymp.shared.array((l+2,b+2), dtype='uint8')
i=None
j=None
with pymp.Parallel(2) as p1:
  with pymp.Parallel(2) as p2:
    for i in p1.range(0,l):
      for j in p2.range(0,b):
        padded[i+1][j+1]=face[i][j]


res = pymp.shared.array((l,b), dtype='uint8')
i=None
j=None
with pymp.Parallel(2) as p1:
  with pymp.Parallel(2) as p2:
    for i in p1.range(1,l+1):
      for j in p2.range(1,b+1):
        res[i-1][j-1] = (convx[0][0]*padded[i-1][j-1] + convx[0][1]*padded[i-1][j]+convx[0][2]*padded[i-1][j+1]+
            convx[1][0]*padded[i][j-1]+convx[1][1]*padded[i][j] + convx[1][2]*padded[i][j+1]+
            convx[2][0]*padded[i+1][j-1]+ convx[2][1]*padded[i+1][j] +convx[2][2]*padded[i+1][j+1])
```

Terakhir, print output dari program sebagai berikut.

```
img = Image.fromarray(res)
img.save('my.png')
img.show()
b = datetime.datetime.now()
print(b-a)
```

Output dan perbandingan

Kedua program memberikan hasil yang sama sebagai berikut, yaitu hasil gambar yang blur.

Sebelum (kri) dan Sesudah (kanan) Gaussian Blur

Walau kedua hasil output yang diperoleh sama, tetapi cara kerja mereka sangat berbeda. Pada program serial, gaussian blur dilakukan pada gambar sebagai suatu kesatuan, sedangkan pada program paralel, gaussian blur dilakukan dengan membagi gambar menjadi 4 thread (sesuai yang digunakan oleh penulis). Dampak dari paralelisasi ini adalah waktu eksekusi dari gaussian blur yang lebih cepat setelah diparalelisasi, seperti ditunjukkan di bawah.

Untuk image .png size 246 x 300

execution time (Serial): 6.22s

execution time (Paralel): 3.345s

Oleh karena itu diperoleh speedup 6.22/3.345 = 1.8594978

Jika kita buat untuk input gambar dengan ukuran berbeda sebagai berikut, maka akan kita peroleh hasil sebagai berikut.

Image Size	*Serial(in seconds)*	*Parallel(in seconds)*	*Speed up*
512X512	3.14	1.77	1.77
768X1024	9.22	5.21	1.77
2048X3072	72.30	41.08	1.76
3464X5200	214.34	122.48	1.75

2.4.3 Tugas Pribadi Sulthan Ali Pasha - Integral Riemann dengan MPI

Pada tugas pribadi ini, saya akan menunjukkan cara perhitungan integral riemann dari fungsi y=sin(x) dan juga pembuatan grafiknya dengan menggunakan MPI.

```
from mpi4py import MPI
import numpy
import scipy
import matplotlib.pyplot as pyplot
from matplotlib.patches import Rectangle
from scipy import integrate
import timeit
```

Langkah awal adalah kita memasukkan modul-modul yang kita butuhkan.

```
mulai = timeit.default_timer()
```

Lalu untuk melakukan perhitungan waktu yang dibutuhkan, kita definisikan untuk mulai perhitungan waktu.

```
def f(x) :
    y = numpy.cos(x)
    return y
```

Dilanjutkan dengan memasukkan fungsi yang ingin kita lakukan perhitungan, pada kasus ini fungsinya adalah f(x)=y=sin(x)

```
def main() :
    ukuran = MPI.COMM_WORLD.Get_size()
    rank = MPI.COMM_WORLD.Get_rank()
```

Selanjutnya, kita masukkan variabel MPI yang kita butuhkan. Untuk kesempatan kali ini kita membutuhkan variabel *size* dan *rank*. Variabel *size* dibutuhkan untuk menghitung total ukuran dari grafik integral riemann yang akan kita hitung, dan *rank* dibutuhkan untuk membagi jumlah proses, bergantung terhadap berapa prosesor yang kita gunakan.

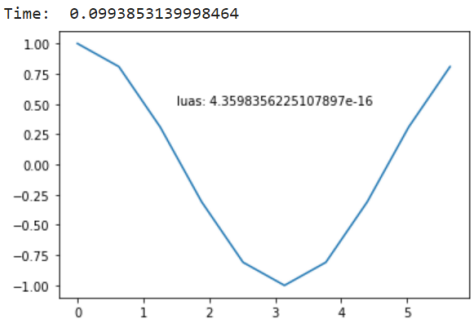
```
bb = 0.0
ba = 2.0 * scipy.pi

samplesPerRank = 10

lebar = (ba-bb) / (samplesPerRank*ukuran)

rankmulai = bb + lebar*samplesPerRank*rank
rankselesai   = rankmulai + lebar*samplesPerRank

koordinat = numpy.empty([samplesPerRank, 2])

area = numpy.zeros(1)


if rank == 0:
    seluruhkoordinat = numpy.empty([samplesPerRank*ukuran, 2])

else:
    seluruhkoordinat  = None
```

Setelahnya mulai dilakukan pendefinisian untuk persiapan perhitungan. Dimulai dengan mendefinisikan batas atas dan batas bawah fungsi, lalu berapa sampel yang akan digunakan tiap *rank*. Lalu didefinisikan untuk memulai perhitungan. Diciptakan juga array pada rank 0 untuk menampung seluruh hasil.

```
    for i in range(0, samplesPerRank) :
        x = rankmulai + i*lebar
        y = f(x)
        koordinat[i] = [x, y]
        area = area + lebar*y

    MPI.COMM_WORLD.Gather(koordinat, seluruhkoordinat, root=0)


    if rank == 0:
        ax = pyplot.figure().add_subplot(1,1,1)

        pyplot.plot(*zip(*seluruhkoordinat))

        for i in range(0, samplesPerRank*ukuran) :
            tinggi = seluruhkoordinat[i][1]

        luasarea = integrate.quad(f, bb, ba)[0]

        ax.text(1.5, 0.5, "luas: " + str(luasarea) , fontsize=10)

        pyplot.savefig("plot")

if __name__ == "__main__":
    main()
selesai = timeit.default_timer()
print('Time: ', selesai - mulai)
```

Lalu definisikan untuk koordinat dan area yang akan digunakan. Setelah itu akan dikumpulkan semua hasil, untuk memulai pembuatan grafik. Setelah itu dimulai pembuatan grafik, dan perhitungan luas areanya. Setelah itu munculkan luas area pada grafik pada tempat yang ditentukan. Setelah itu simpan seluruhnya, dan lakukan eksekusi. Setelah seluruhnya selesai, selesaikan pula timer yang sudah dimulai pada

awal program agar dapat kita ketahui waktu yang digunakan untuk keseluruhan proses.



Berikut merupakan hasil dari program yang telah dijalankan.

BAB III

PENUTUP

1. Kesimpulan

Matriks sparse adalah matriks dengan sebagian besar entrinya adalah nol. Sebagai konsekuensi dari sifatnya, mereka dapat direpresentasikan dan disimpan secara efisien dengan hanya menyimpan nilai bukan nol dan posisinya di dalam matriks. Selain itu, operasi seperti perkalian matriks dapat diimplementasikan lebih efisien untuk matriks sparse.

Berbeda dengan matriks dense, operasi pada matriks sparse lebih hemat memori, karena matriks sparse dapat direpresentasikan dalam bentuk lain, di mana elemen nol tidak diperhitungkan sehingga tidak ada operasi yang terbuang percuma. Matriks dense tidak memiliki sifat ini, oleh karena itu operasi pada matriks dense lebih memakan memori.

Terdapat berbagai metode yang dapat kita gunakan dalam melakukan perkalian matriks. Perhitungan dapat dilakukan secara sequential maupun parallel. Perbedaan dari kedua metode tersebut adalah bahwa metode paralel lebih cepat dalam proses hingga menemukan hasil yang dibutuhkan.

DAFTAR PUSTAKA

Matrix Multiplication in NumPy - GeeksforGeeks. (2020, August 29). GeeksforGeeks. https://www.geeksforgeeks.org/matrix-multiplication-in-numpy/?ref=rp

Implementation of COO and CSR Based on Array Form for Sparse Matrix. (2019). Programmer.ink. https://programmer.ink/think/implementation-of-coo-and-csr-based-on-array-form-for-sparse-matrix.html

Team, C. (2014, June 4). *Parallel matrix matrix multiplication*. Codee. https://www.codee.com/parallel-matrix-matrix-multiplication/

Team, C. (2014, May 28). *Parallel computation of matrix-vector product*. Codee. https://www.codee.com/parallel-computation-of-matrix-vector-product/