Kompleksitas Algoritma Rank Sort dan Implementasi pada Parallel Programming Dengan Menggunakan OpenMP

Muhammad Indra N.S. - 23515019[1]
Program Magister Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]23515019@std.stei.itb.ac.id

***Abstract*—Salah satu permasalahan yang paling mendasar dalam pemrograman adalah proses pengurutan data. Untuk menyelesaikan permasalahan ini, ada banyak solusi yang yang dapat dilakukan, dikenal dengan sebutan algoritma pengurutan. Trend pemrograman sekarang juga telah berubah paradigma dari yang awalnya *single-core programming*, menjadi *multi-core programming*. Hal ini dilakukan demi meningkatkan performansi dalam pemrograman. Paper ini dibuat untuk membandingkan antara algoritma pengurutan dengan *single-core* dengan *multi-core precessing*.**

***Index Terms*—Algoritma, OpenMP, *parallel programming, Rank Sort*.**

I. PENDAHULUAN

Pengurutan adalah salah satu permasalahan yang tidak pernah habis di bahas dalam dunia pemrogramman. Solusi untuk penyelesaiannya pun telah banyak bermunculan. Solusi-solusi ini disebut dengan algoritma pengurutan. Beberapa algoritma pengurutan ada yang simpel dan intuitif sehingga mudah dipahami, ada juga yang sangat kompleks dan rumit tapi namun dapat menyelesaikan masalah dengan cepat. Setiap pengurutan data dapat terselesaikan, meski memakan waktu yang berbeda-beda tergantung algoritma mana yang digunakan.

Pada jaman sekarang, faktor untuk meningkatkan performansi dalam proses pengurutan data tidak hanya bergantung pada faktor algoritma saja, karena trend pemrograman sekarang ini sedang mengarah ke pemrograman *multi-core* yang sebelumnya masih berdasarkan pada *single-core*. Sehingga bermunculan algoritma-algoritma lain yang memang didisain agar lebih efisien ketika berjalan pada pemrograman paralel.

Salah satu algoritma pengurutan yang dapat digunakan pada pemrograman *single-core* dan dapat diefisienkan pada pemrograman *multi-core* adalah algoritma *Rank Sort.* Algoritma ini akan di uji pada *single-core programming* dengan bahasa pemrograman C++ pada umumnya. Sedangkan untuk pemrograman paralelnya akan digunakan API openMP untuk bahasa pemrograman C++. Setelah itu akan dibandingkan kompleksitas serta waktu eksekusinya untuk masing-masing pemrograman baik single maupun paralel.

II. LANDASAN TEORI

A. Algoritma Rank Sort

Pengertian algoritma adalah sebuah prosedur komputasi yang telah didefinisikan secara baik yang mengambil beberapa nilai, atau himpunan dari nilai-nilai, sebagai sebuah *input* dan menghasilkan beberapa nilai, atau himpunan nilai sebagai sebuah *output*.[1]

Algoritma adalah aturan tertentu (atau himpunan dari aturan-aturan) yang menspesifikasikan bagaimana menyelesaikan beberapa masalah; sebuah himpunan dari prosedur-prosedur yang memastikan untuk menemukan sebuah solusi tepat dari sebuah permasalahan.[3]

Algoritma adalah sebuah himpunan dari langkah-langkah yang dapat dikomputasi untuk mencapai sebuah hasil yang diinginkan.[2]

Algoritma adalah urutan langkah-langkah untuk memecahkan suatu persoalan, dengan memproses inputan menjadi keluaran.[4]

Sorting adalah sebuah operasi yang memisahkan item-item menjadi sebuah grup berdasarkan kriteria yang lebih spesifik.[3]

Sorting adalah pengurutan item-item dalam urutan yang telah ditentukan sebelumnya.[2]

Algoritma *Rank Sort* merupakan algoritma yang cara pengurutannya mencari ranking dari tiap angka yang ada dalam list, kemudian memindahkannya ke lokasi yang tepat.[5]

Pada *Rank Sort*, banyaknya bilangan yang lebih kecil daripada tiap bilangan yang dipilih dijumlahkan. Jumlah ini digunakan sebagai dasar pemilihan posisi bilangan yang akan dipilih selanjutnya pada list, yaitu "*rank*"-nya pada list.[6]

B. OpenMP

OpenMP adalah sebuah *Application Programming Interface* (API) untuk *shared-memory* yang memiliki fitur untuk memfasilitasi kebutuhan akan paralel programming.[7]

OpenMP bukanlah bahasa pemrograman. OpenMP dapat dikatakan seperti sebuah notasi yang dapat ditambahkan pada bahasa C++ untuk menjadikan pekerjaan yang ada di *shared* kedalam beberapa *thread*

yang kemudian akan di eksekusi pada prosesor atau *core* yang berbeda. Peletakkan fitur-fitur openMP yang tepat kedalam program yang sequential akan membuat beberapa aplikasi mendapatkan keuntungan dengan adanya *shared-memory* pada arsitektur paralel.

Ide dari openMP adalah sebuah entitas yang sedang berjalan dapat mengeksekusi alur dari instruksi secara independen. OpenMP dibangun dalam sebuah pekerjaan besar yang mendukung spesifikasi program untuk dieksekusi oleh beberapa *thread* yang saling mendukung. Sistem operasi akan akan membuat proses untuk mengeksekusi program; sistem operasi akan mengalokasikan beberapa sumber daya untuk proses tersebut, termasuk *memory* dan *register* untuk nilai objek tertentu. Jika *multiple thread* berkolaborasi untuk mengeksekusi program, maka *thread-thread* ini akan melakukan sharing terhadap sumber daya yang ada termasuk *address space* di program tersebut.

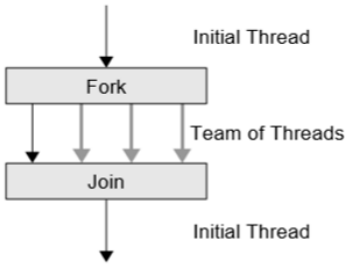
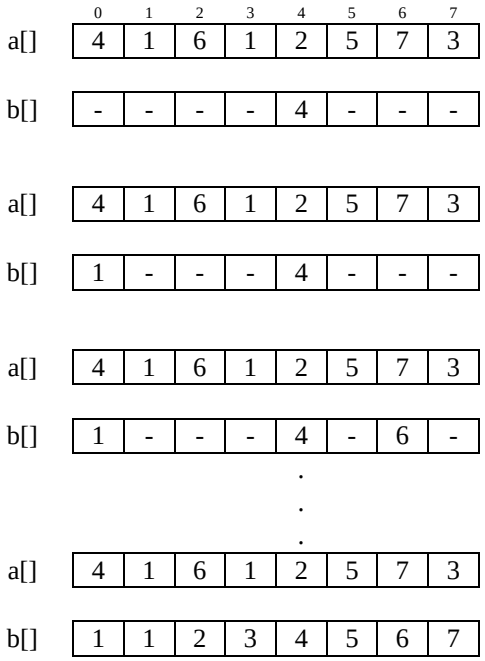


Fig. 1: *Fork-Join programming model* yang digunakan pada openMP. [7]

Pemodelan pemrograman yang digunakan pada openMP adalah pemodelan *Fork-Join*, seperti yang digambarkan pada *Fig. 1*. Berdasarkan pendekatan model ini, program dimulai sebagai pengeksekusian dengan *single thread*, seperti program sekuensial. *Thread* yang menjalankan program ini disebut dengan *Initial Thread*. Sewaktu *construct* dari openMP bertemu dengan sebuah *thread* ketika mengeksekusi program, maka akan membentuk *team of thread* (ini adalah *Fork*), yang mana *Initial Thread* akan menjadi *master* dari *thread* dan berkolaborasi dengan *team of thread* lainnya untuk mengeksekusi program secara dinamis didalam *construct* ini. Pada akhir *construct*, hanya *initial thread*, atau *master* dari *thread* yang masih aktif dan berjalan, sedangkan *thread-thread* dari *team of thread* berhenti (ini adalah *Join*).

III. PEMBAHASAN

Algoritma yang digunakan untuk proses *sorting* disini adalah algoritma *Rank Sort*, dimana angka yang nilainya lebih kecil dari angka yang dipilih dihitung. Hasil hitungan ini merupakan representasi posisi dari angka yang dipilih dalam list yang telah terurut. Sebagai contoh, misal terdapat n angka yang tersimpan dalam sebuah *array*, a[0] ... a[n-1]. Pertama-tama, a[0] dibaca dan dibandingkan dengan angka-angka yang lain, a[1], dan dilakukan pencatatan jumlah angka yang lebih kecil dari angka yang dipilih pertama kali, a[0]. Misalnya jumlahnya adalah x, maka x dijadikan sebagai indeks lokasi pada *array* akhir yang telah terurut. Angka a[0] disalin ke *array* akhir yang telah terurut, b[0] ... b[n-1], pada lokasi b[x], kemudian angka selanjutnya dibaca dan dibandingkan dengan angka-angka yang lain, a[0], a[2], ... a[n-1]. Begitu seterusnya sampai angka di indeks terakhir telah dibandingkan dan disalin ke *array* b.



Untuk membandingkan satu angka dengan n-1 angka yang lain dibutuhkan paling n-1 langkah jika dilakukan pada *sequential programming*. Untuk dapat melakukan langkah-langkah tersebut dengan seluruh n angka, dibutuhkan n(n-1) langkah. Untuk perhitungan kompleksitasnya, kita akan menggunakan notasi *Big-O*.

$T(n) = n(n-1)$
$T(n) = n^2 - n$
$O(n) = O(n^2) - O(n)$
$O(n) = O(\max(n^2, n))$
Jadi, $O(n) = O(n^2)$

Dengan kompleksitas $O(n^2)$, maka algoritma ini bisa dikategorikan algoritma yang polinomial. Untuk kode sekuensial aktualnya seperti di bawah ini:

```
for (i = 0; i < n; i++){
      x = 0;
      for (j = 0; j < n; j++){
            if ( a[i] > a[j] )
                  x++;
      b[x] = a[i];
}
```

Dengan dua loop `for` pada setiap iterasi untuk n langkah. Kode yang diberikan tersebut pada kenyataannya

tidak akan membuat *array* terurut dengan benar jika terdapat duplikasi angka, karena angka yang sama akan diletakkan di lokasi yang sama. Meskipun demikian, untuk memecahkan masalah yang muncul, kode tersebut dapat dengan mudah dimodifikasi, sehingga menjadi sebagai berikut:

```
for (i = 0; i < n; i++){
  x = 0;
  for (j = 0; j < n; j++){
   if ( a[i] > a[j] )
     x++;
   else if (a[i] == a[j] && j < i)
     x++;
  b[x] = a[i];
}
```

Untuk *parallel programming*, kompleksitas yang diperoleh tergantung jadi jumlah prosesor atau *core* yang ada pada komputer pada saat melakukan proses *compile*. Kompleksitas untuk pemrograman paralel yang ada disini adalah n langkah untuk setiap angka (n-1) dibagi dengan jumlah prosesor atau *core* p. Pengujian kompleksitas disini di bagi menjadi dua *case*, *Worst Case* dimana jumlah prosesor atau *core* lebih kecil dari jumlah data. Maka akan di dapat kompleksitas dengan notasi *Big-O*,

$T(n) = n(n-1) / p$
$T(n) = (n^2 – n) / p$
$O(n) = O(n^2/p) – O(n/p)$
$O(n) = O(\max(n^2/p, n/p))$
Jadi, $O(n) = O(n^2/p)$

Untuk *Best Case* dimana jumlah prosesor atau *core* minimal sama dengan jumlah data. Maka akan di dapat kompleksitas dengan notasi *Big-O*,

$T(n) = n(n-1) / p$
$T(n) = n – 1$
$O(n) = O(n) – O(1)$
$O(n) = O(\max(n, 1))$
Jadi, $O(n) = O(n)$

Dengan kompleksitas untuk *worst case*nya O(n2), maka algoritma ini bisa dikategorikan algoritma yang polinomial. Sedangkan kompleksitas untuk *best case*nya O(n), maka algoritma ini bisa dikategorikan algoritma yang linier. Untuk kode paralelnya di tulis dalam notasi `forall`, sehingga menjadi seperti di bawah ini:

```
forall (i = 0; i < n; i++){
  x = 0;
  for (j = 0; j < n; j++){
   if ( a[i] > a[j] )
     x++;
   else if (a[i] == a[j] && j < i)
     x++;
  b[x] = a[i];
}
```

IV. IMPLEMENTASI

Bahasa yang digunakan dalam pengerjaan program ini dengan menggunakan bahasa C++. Sedangkan untuk paralel C++ dengan API dari openMP. Skenario pengukuran kinerja pada program Rank Sort ini dilakukan dengan jumlah data yang bervariasi, mulai dari 10, 50, 100, 500, 1000, 5000, 10000, 50000, sampai 100000 data yang harus diurutkan. Untuk data yang akan diurutkan didapat secara *random* dengan menggunakan fungsi rand(). Persiapan yang dilakukan untuk pengukuran kinerja program adalah sebagai berikut:

1. Pengukuran dilakukan dengan spesifikasi *hardware* sebagai berikut:
   Prosesor : Intel(R) Core (TM i5-4200U CPU @ 1,60 GHz
   Jumlah *core* : 4 CPUs
   atau prosesor
   Memori : 3,8 GiB RAM
2. Pengukuran dilakukan pada sistem operasi berbasis Linux, dalam hal ini adalah Ubuntu 15.10 (32-bit). Dengan menggunakan *editor* Geany dalam menuliskan kode program. Disamping itu, *compiler* untuk mengkompilasi kode program dilakukan oleh g++ (GNU Licence) versi 5.2.1.
3. Pengukuran waktu menggunakan fungsi clock_gettime().

Pada sekuensial program, tidak perlu mengatur atau menambahkan apa pun ketika akan melakukan percobaan ini. Hanya menambahkan beberapa baris *code* untuk melakukan penghitungan waktu eksekusi, yang dalam hal ini tidak perlu ditampilkan keseluruhan *code*nya. *Code* program yang digunakan pada sekuensial ini menjadi seperti prosedur di bawah ini,

```
void sorting_single(int *a, int n){
  int i, j, count;
  int *temp = new int[n];
  for (i = 0; i < n; i++){
    count = 0;
    for (j = 0; j < n; j++){
      if (a[j] < a[i])
        count++;
      else if (a[j] == a[i] && j < i)
        count++;
    }
    temp[count] = a[i];
  }
  memcpy(a, temp, n * sizeof(int));
  delete[] temp;
}
```

Yang dilakukan dalam openMP adalah membagi *task-task* yang ada kedalam sejumlah *thread*. *Thread-thread* disini dapat diatur sendiri jumlahnya dengan memanggil fungsi **omp_set_num_threads().** Dalam percobaan kali ini, jumlah *thread* yang akan digunakan dibatasi oleh jumlah *core* yang ada di CPU, dengan menggunakan fungsi **omp_get_num_procs().** Ketika proses pengurutan akan dilakukan, digunakan kata kunci **#pragma omp parallel** untuk membagi proses pengurutan kedalam

thread yang ada. Dalam openMP juga dapat dilakukan pengaturan *variable* mana saja yang akan di *share*, atau yang mana yang akan di atur sebagai *private*. Jika *variable* itu di atur sebagai *private*, maka tiap *thread* akan memiliki *variable* itu untuk masing-masing. Disini ada sebuah *variable* yang di atur sebagai *private* yang bernama count, sehingga ditambahkan kata kunci **private(count)**. Setelah itu digunakan kata kunci **#pragma omp parallel for** sehingga *loop* yang ada di bagi kembali kedalam *thread-thread* baru.

Code program yang digunakan pada paralel openMP ini menjadi seperti prosedur di bawah ini,

```
void sorting_parallel(int *a, int n){
  int i, j, count;
  int *temp = new int[n];
  omp_set_num_threads(
     omp_get_num_procs());
  #pragma omp parallel private (count)
  for (i = 0; i < n; i++){
    count = 0;
    #pragma omp parallel for
    for (j = 0; j < n; j++){
      if (a[j] < a[i])
        count++;
      else if (a[j] == a[i] && j < i)
        count++;
    }
    temp[count] = a[i];
  }
  memcpy(a, temp, n * sizeof(int));
  delete[] temp;
}
```

Pada tiap program tersebut, pengukuran waktu dilakukan dengan menggunakan fungsi clock_gettime(), dimana fungsi ini dipanggil untuk mendapatkan waktu saat sebelum algoritma dijalankan, dan setelah algoritma selesai. Disamping itu, skenario program dilakukan dengan melakukan pengurutan mulai dari 10, 50, 100, 500, 1000, 5000, 10000, 50000, sampai 100000 angka. Setiap pengurutan data dilakukan pengulangan dengan data yang sama sebanyak 5 kali. Hal ini dilakukan agar hasil data yang diambil bisa lebih valid sehingga diambil dari hasil rata-rata waktu pengujian dari masing-masing program.

Proses kompilasi pada sekuensial program dilakukan dengan command :

muinnusa@muinnusa-S451LN:~/Documents/code/uas$ g++ sorting_serial.cpp –o sorting_serial

Loop ke-	1	2	3	4	5
10	0,00033	0,00000	0,00001	0,00000	0,00001
50	0,00010	0,00005	0,00005	0,00005	0,00005
100	0,00020	0,00019	0,00020	0,00021	0,00023
500	0,00297	0,00230	0,00350	0,00141	0,00225
1000	0,01374	0,01239	0,01140	0,01427	0,01334
5000	0,10640	0,11025	0,11054	0,10519	0,13931
10000	0,39190	0,39340	0,39818	0,39483	0,39052
50000	9,88921	9,81492	9,98937	9,48148	9,34293
100000	39,27441	39,01293	39,19835	39,91348	39,14032
500000	980,33450	980,41029	980,23231	980,40129	980,31024

Fig. 2: Hasil eksekusi waktu dari pemrograman sekuensial dalam satuan detik.

Setelah itu dilakukan running program dengan command :

muinnusa@muinnusa-S451LN:~/Documents/code/uas$ ./sorting_serial

Dari kompilasi dan running program sekuensial di atas, maka didapatlah hasil waktu eksekusi seperti pada Fig. 2.

Sedangkan untuk yang openMP, proses kompilasi pada paralel program dilakukan dengan command :

muinnusa@muinnusa-S451LN:~/Documents/code/uas$ g++ sorting_paralel.cpp –o sorting_ paralel -fopenmp

Loop ke-	1	2	3	4	5
10	0,00001	0,00000	0,00001	0,00000	0,00001
50	0,00006	0,00005	0,00005	0,00005	0,00005
100	0,00020	0,00019	0,00020	0,00021	0,00023
500	0,00475	0,00442	0,00428	0,00441	0,00332
1000	0,01571	0,01512	0,01851	0,01427	0,01734
5000	0,17694	0,17148	0,18243	0,17848	0,17622
10000	0,65117	0,65335	0,65525	0,65525	0,65115
50000	16,44365	16,44235	16,47468	16,48257	16,48427
100000	67,12773	67,13415	67,12713	67,12735	67,12114
500000	1656,91014	1656,91251	1656,98458	1656,95737	1656,91825

Fig. 3: Hasil eksekusi waktu dari pemrograman paralel dengan openMP dalam satuan detik.

Setelah itu dilakukan running program dengan command :

muinnusa@muinnusa-S451LN:~/Documents/code/uas$ ./sorting_ paralel

Dari kompilasi dan running program paralel di atas, maka didapatlah hasil waktu eksekusi seperti pada Fig. 3.

Tabel 1: Tabel hasil rata-rata dari percobaan sekuensial dan paralel openMP dalam satuan detik

Banyaknya Data	Sekuensial	Parallel openMP
10	0,00001	0,00007
50	0,00005	0,00006
100	0,00021	0,00021
500	0,00424	0,00249
1000	0,01619	0,01303
5000	0,17711	0,11434
10000	0,65323	0,39377
50000	16,46550	9,70358
100000	67,12750	39,30790
500000	1656,93657	980,33773

Dari hasil eksekusi kedua program dapat di ambil rata-rata untuk tiap jumlah data yang telah diurutkan seperti yang terlihat pada Tabel 1.

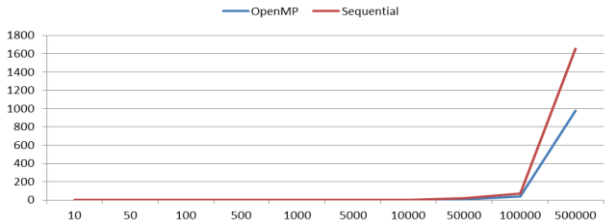


Fig. 4: Diagram rata-rata waktu eksekusi

Berdasarkan data pada Tabel 1 dapat dibuat sebuah diagram yang membandingkan waktu eksekusi dari program yang sekuensial serta program paralel dengan openMP.

V. KESIMPULAN

Kesimpulan yang didapat dari pengujian diatas adalah Algoritma Rank Sort algoritma yang dapat digunakan baik secara sekuensial maupun paralel. Meski jika digunakan pada sekuensial akan menghasilkan kompleksitas algortima yang polinomial, $O(n^2)$. Sedangkan jika akan digunakan dengan menggunakan pemprograman paralel, kompleksitas akan sangat tergantung dengan jumlah prosesor atau core yang ada dalam komputer. Dalam percobaan yang dilakukan ini, ada keterbatasan dalam hardware yang hanya memiliki 4 core saja, sehingga kompleksitas dari algoritmanya menjadi $O(n^2/p)$.

Meskipun begitu, waktu eksekusi yang didapat lebih cepat ketika jumlah data menjadi lebih banyak. Hal ini membuktikan bahwa pemrograman paralel ini berjalan dengan benar.

Untuk kedepannya diharapkan dapat dibuktikan bahwa algoritma Rank Sort ini dapat menjadikan kompleksitas menjadi O(n) dengan n prosesor. Bahkan dapat menjadi O(1) jika menggunakan komputer dengan n^2 prosesor.

REFERENCES

[1] Cormen, et all. "*Introduction to Algorithm 3rd edition*". The MIT Press. 2009.

[2] Paul E. Black, in Dictionary of Algorithms and Data Structures [online], Vreda Pieterse and Paul E. Black, eds. 10 January 2007. (accessed 20 Desember 2015 22:34) Available from: http://www.nist.gov/dads/HTML/algorithm.html

[3] http://www.webster-dictionary.org/ (accessed 20 Desember 2015 22:34)

[4] http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2014-2015/Pengantar%20Strategi%20Algoritma%20(2015).ppt (accessed 20 Desember 2015 22:40)

[5] http://web.mst.edu/~ercal/387/Suppl/class.9.txt (accessed 20 Desember 2015 23:17)

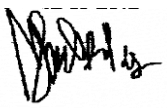
[6] http://grid.cs.gsu.edu/~cscyip/csc4310/Slides9.pdf (accessed 20 Desember 2015 23:42)

[7] Chapman, Barbara. "*Using OpenMP : portable shared memory parallel programming"*. The MIT Press. 2008.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 21 Desember 2015



Muhammad Indra N.S.
23515019