Unjuk Kerja *Selection Sort Hybrid*

Eduardus Hardika Sandy Atmaja[1], Kartono Pinaryanto[2]
[1,2]Program Studi Informatika, Fakultas Sains dan Teknologi,
Universitas Sanata Dharma
Kampus III, Paingan, Maguwoharjo, Depok, Sleman, Yogyakarta, Indonesia
Email: [1]edo@usd.ac.id, [2]kartono@usd.ac.id

Masuk: 23 Oktober 2019; Direvisi: 27 Maret 2020; Diterima: 18 April 2020

Abstract. *Sorting is the most basic and important process in data processing. The sorting process on large data causes large computation. Some existing sorting algorithms need to be improved to further improve their performance. This study tried to develop an existing selection sort algorithm, into a selection sort hybrid algorithm that is expected to have better performance. Selection sort hybrid algorithm is an algorithm that combines both minimum and maximum searching techniques. It can find minimum and maximum values in the same time to sort from the both side of the data. Since it can be done separately, multithreading is used to do this job. So the sorting process can be done simultaneously. Several tests using different amounts of data have been conducted to compare the performance of the algorithms. The result is selection sort hybrid algorithm more efficient than the origin selection sort. Henceforth, the result obtained from the research can be used for various purposes related to data processing in informatics area.*
Keywords*: sorting, selection sort, selection sort hybrid, computing*

Abstrak. *Sorting atau pengurutan adalah proses yang paling mendasar dan penting dalam pemrosesan data. Proses pengurutan pada data yang besar menyebabkan komputasi menjadi tinggi. Maka beberapa algoritme pengurutan perlu ditingkatkan kinerjanya. Penelitian ini mencoba mengembangkan algoritme selection sort, menjadi algoritme selection sort hybrid yang diharapkan memiliki kinerja yang lebih baik. Algoritme selection sort hybrid adalah algoritme yang menggabungkan teknik pencarian minimum dan maksimum. Algoritme tersebut dapat menemukan nilai minimum dan maksimum dalam waktu yang bersamaan untuk mengurutkan data dari kedua sisinya. Karena dapat dikerjakan secara terpisah, maka teknik multithreading digunakan untuk melakukan pekerjaan ini. Jadi proses pengurutan data bisa dilakukan secara simultan. Beberapa pengujian menggunakan jumlah data yang berbeda telah dilakukan untuk membandingkan kinerja kedua algoritme ini. Hasilnya adalah algoritme selection sort hybrid lebih efisien daripada algoritme selection sort untuk semua kasus yang diberikan. Diharapkan hasil yang diperoleh dari penelitian ini dapat digunakan untuk berbagai keperluan terkait dengan pengolahan data di bidang informatika.*
Kata kunci*: Pengurutan, selection sort, selection sort hybrid, komputasi*

1. Pendahuluan

Perkembangan teknologi yang semakin cepat menyebabkan proses pertukaran data yang semakin cepat dan dalam skala yang besar. Pengolahan data pada skala besar membutuhkan biaya dan waktu yang cukup tinggi sehingga perlu adanya teknik komputasi yang handal. Penggunaan teknik komputasi yang handal diharapkan dapat meningkatkan proses pengolahan data menjadi lebih efektif dan efisien sehingga akan menghemat waktu dalam proses komputasi.

Salah satu proses yang umum dilakukan pada proses pengolahan data adalah *sorting* atau pengurutan. *Sorting* adalah proses untuk mengurutkan data berdasarkan suatu urutan tertentu seperti urutan angka atau urutan alfabet [1]. Terdapat banyak algoritme pengurutan data yang

telah diimplementasikan. Salah satu jenis algoritme *sorting* yang banyak dipakai adalah *selection sort*. *Selection sort* merupakan proses pengurutan data dengan menggunakan prinsip minimum atau maksimum [2]. *Selection sort* dengan minimum atau maksimum menghasilkan perhitungan kompleksitas waktu asimptotik yaitu n^2 sehingga waktu komputasi *selection sort* tidak efisien. Perlu adanya modifikasi algoritme untuk meningkatkan kinerja dari algoritme tersebut.

Berdasarkan permasalahan diatas dibutuhkan suatu modifikasi algoritme yang dapat mengatasi permasalahan waktu komputasi algoritme *selection sort*. Solusi yang ditawarkan pada penelitian ini adalah dengan menggabungkan algoritme *selection* minimum dan *selection* maksimum yang selanjutnya diberi nama algoritme *selection sort hybrid*. Algoritme *selection sort hybrid* adalah algoritme yang mengkombinasikan proses komputasi dengan melakukan pencarian nilai minimum dan nilai maksimum (bekerja dua arah) secara simultan dengan menggunakan *multithreading*. *Thread* sendiri merupakan sebuah rangkaian kode program yang dapat dijalankan secara independen oleh prosesor [3]. Ketika beberapa *thread* berjalan secara bersamaan dalam satu program utama maka dapat disebut sebagai *multithreading*. Alasan pemilihan algoritme *selection sort* adalah karena algoritme *selection sort* hanya melakukan pertukaran data satu kali saja dalam satu iterasi pengurutan data. Pertukaran dilakukan bila 1) telah ditemukan nilai minimum yang akan ditukarkan dengan indeks data yang belum terurut terdepan (teknik pencarian nilai minimum) 2) telah ditemukan nilai maksimum yang akan ditukarkan dengan indeks data yang belum terurut terbelakang (teknik pencarian nilai maksimum). Arti dari kondisi tersebut adalah dalam satu iterasi pengurutan kita dapat sekaligus mencari nilai minimum dan maksimum lalu menukarnya dengan indeks data yang belum terurut terdepan dan terbelakang tanpa khawatir adanya tabrakan pertukaran data. Selain itu, karena kondisi tersebut proses pengurutan data juga dapat dilakukan secara simultan dengan menggunakan *multithreading*. Penggunaan *multithreading* pada proses pengurutan data dua arah pada algoritme *selection sort hybrid* diharapkan dapat mengurangi waktu komputasi sehingga algoritme dapat berjalan dengan lebih cepat.

2. Tinjauan Pustaka

Terdapat banyak penelitian yang membahas mengenai algoritme pengurutan, beberapa diantaranya mencoba untuk membandingkan beberapa jenis algoritme pengurutan [4, 5, 6]. Hasilnya adalah setiap algoritme pengurutan memiliki kelebihan dan kekurangan dari segi kompleksitas waktu algoritme. Perbedaan kompleksitas waktu algoritme disebabkan oleh ketidakstabilan algoritme dalam menangani variasi jumlah data. Kelemahan algoritme yang dipakai untuk mendapatkan hasil yang lebih optimal belum diatasi.

Penelitian lain juga mencoba untuk membandingkan beberapa algoritme pengurutan sederhana dengan algoritme *advance* (algoritme lanjut) [7]. Hasilnya adalah algoritme *selection sort* merupakan algoritme yang paling stabil karena tidak terpengaruh oleh variasi input data. Namun, pengoptimalan algoritme *selection sort* belum dicoba.

Memodifikasi algoritme *selection sort* untuk mendapatkan hasil yang optimal juga telah dilakukan [8]. Modifikasi algoritme dengan cara mengurangi operasi pertukaran data pada setiap iterasi sehingga proses iterasi dapat diminimalkan sudah dilakukan. Tetapi, hanya salah satu metode *selection* minimum atau maksimum yang digunakan. Sementara, penggabungan kedua metode tersebut belum dilakukan. Padahal, penggabungan tersebut merupakan peluang untuk lebih meningkatkan kinerja algoritme *selection sort*.

Penelitian lain juga mencoba memodifikasi algoritme *selection sort* [9] dengan mengimplementasikan *queue* untuk menampung sementara data yang memiliki nilai yang sama sebelum diurutkan. Hal tersebut dilakukan untuk mengurangi iterasi ketika menukar data. Data-data yang ditampung *queue* tersebut selanjutnya diletakkan di tempat yang tepat dan tidak akan diproses pada iterasi berikutnya karena sudah dianggap terurut. Kelemahan dari algoritme yang dimodifikasi adalah tidak stabil bagi data yang bervariasi atau unik. Modifikasi yang dilakukan tidak ada fungsinya ketika semua data berbeda dan hasilnya akan sama dengan *selection sort* biasa.

Percobaan implementasi *stack* dalam *selection sort* telah dilakukan dalam penelitian lain untuk menyimpan data yang diprediksi berada di partisi yang tepat dalam sebuah *dataset* [10]. Hasil dari penelitian tersebut menunjukkan pengurangan waktu komputasi yang signifikan dibandingkan dengan *selection sort* biasa. Kelemahan dari algoritme ini adalah adanya penambahan *stack* yang menyebabkan algoritme membutuhkan ruang yang besar untuk menampung sejumlah banyak data sehingga tidak efisien dalam hal *space complexity*.

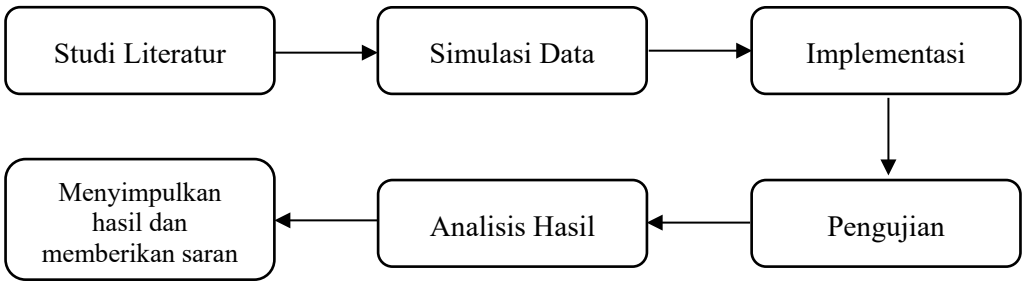
Penggunaan *multithreading* untuk melakukan pengurutan data juga pernah dilakukan untuk meningkatkan performa algoritme *quick sort* [11]. Alasan pemilihan algoritme tersebut adalah karena *quick sort* merupakan algoritme dengan prinsip *divide and conquer*. Prinsip tersebut bekerja dengan cara membagi *array* menjadi dua buah *sub-array* dan secara rekursif melakukan pengurutan terhadap kedua *sub-array* tersebut. Dengan kondisi tersebut setiap *sub-array* dalam rekursif dapat diurutkan secara paralel. Karena tidak dimungkinkan adanya tabrakan sehingga *multithreading* dapat diimplementasikan untuk algoritme ini. Hasil dari penelitian ini menunjukkan bahwa penggunaan *multithreading* pada algoritme *quick sort* memberikan hasil yang lebih baik daripada algoritme *quick sort* tanpa *multithreading*.

Berdasarkan beberapa uraian di atas, penelitian ini mencoba memperbaiki kekurangan pada penelitian sebelumnya dengan melakukan modifikasi algoritme *selection sort* menjadi algoritme *selection sort hybrid*. *Hybrid* yang dimaksud adalah menggabungkan metode *selection* minimum dan maksimum secara simultan dengan menggunakan *multithreading* yang bertujuan untuk mengurangi iterasi dalam pertukaran data. Hasil dari algoritme *selection sort hybrid* diharapkan dapat meningkatkan kinerja algoritme *selection sort*.

3. Metodologi Penelitian

3.1. Langkah-Langkah Penelitian

Metodologi penelitian yang dilakukan dalam penelitian ini adalah langkah-langkah kegiatan implementasi algoritme *selection sort hybrid* untuk mengurutkan data. Gambar 1 memperlihatkan bagan metode penelitian.



Gambar 1. Metodologi penelitian

Gambar 1 menunjukkan metodologi penelitian yang dilakukan dalam penelitian ini. Tahap pertama pada penelitian ini dimulai dengan studi literatur untuk mempelajari teori-teori yang relevan dalam menganalisis persoalan dan mempelajari bagaimana pemecahannya. Teori-teori yang dipelajari adalah topik yang berkaitan dengan algoritme *sorting* khususnya algoritme *selection sort* dan teknik *multithreading*. Tahap kedua adalah simulasi data untuk membuat beberapa *dataset* numerik yang dihasilkan secara otomatis dengan menerapkan mekanisme pembangkitan fungsi *random*. Pembangkitan bilangan *random* tersebut dilakukan dengan tujuan untuk mendapatkan data yang tidak terprediksi dan tidak dimanipulasi (*sampling*). Tahap ketiga adalah merancang algoritme *selection sort* dan *selection sort hybrid* dilanjutkan dengan implementasi ke dalam bahasa pemrograman *Java*. Setelah dilakukan implementasi, selanjutnya dilakukan tahap keempat yaitu pengujian terhadap kedua algoritme dengan menggunakan data-data yang telah disimulasikan sebelumnya. Tujuan dari tahap pengujian adalah membandingkan kinerja kedua algoritme dari sisi waktu komputasi. Tahap kelima ialah menganalisis hasil pengujian yang telah diperoleh untuk mengetahui apakah tujuan dari penelitian tercapai atau tidak. Hal ini ditentukan dari performa waktu komputasi kedua algoritme. Tahap terakhir yang

dilakukan adalah menarik kesimpulan dari penelitian yang telah dilakukan. Saran juga diberikan untuk memperbaiki kekurangan yang ada untuk diterapkan dalam penelitian selanjutnya.

3.2. Simulasi Data

Data yang digunakan pada penelitian ini adalah data numerik yang dibentuk secara acak dengan rentang nilai 0 – 10.000. Data simulasi yang dibentuk berjumlah 100 *dataset* untuk 100 kali pengujian. Jumlah data yang digunakan pada setiap pengujian berbeda-beda mulai dari 10.000 sampai dengan 1.000.000 dengan perbedaan kelipatan 10.000 untuk masing-masing *dataset*. Data simulasi yang dihasilkan selanjutnya digunakan untuk menguji algoritme *selection sort* dan *selection sort hybrid*.

3.3. Desain Algoritme

Algoritme *selection sort hybrid* dibuat dengan mengembangkan algoritme *selection sort* yang telah ada. Berikut ini adalah algoritme *selection sort* dengan pendekatan pencarian minimum [12]:

Algoritme 1. S*election sort* dengan pendekatan pencarian minimum

```
1) Baca data ke dalam array
2) Untuk iterasi = 0 sampai N-2 lakukan langkah 3 sampai 6
3)     Tentukan minIndex = iterasi;
4)     Untuk elemen = iterasi + 1 sampai N-1 lakukan langkah 5
5)     Test apakah array[elemen] < array[minIndex ]
              Jika ya, minIndex = elemen
6)     Tukarkan nilai array[iterasi] dengan array[minIndex]
```

Selection sort pada Algoritme 1 dengan pendekatan pencarian minimum melakukan pengurutan data dengan cara mencari elemen terkecil dari baris data lalu ditukarkan dengan indeks elemen paling depan yang belum terurut. Proses tersebut diulang hingga semua elemen menjadi terurut. Hasilnya berupa rangkaian data yang terurut secara *ascending* (menaik). Apabila ingin mengurutkan data secara *descending* (menurun) maka penukaran elemen terkecil dilakukan dengan indeks paling belakang yang belum terurut. Berikut ini adalah algoritme *selection sort* dengan pendekatan pencarian maksimum [12]:

Algoritme 2. S*election sort* dengan pendekatan pencarian maksimum

```
1) Baca data ke dalam array
2) Untuk iterasi = N-2 sampai 0 lakukan langkah 3 sampai 6
3)     Tentukan maxIndex = iterasi;
4)     Untuk elemen = iterasi - 1 sampai 0 lakukan langkah 5
5)     Test apakah array[elemen] > array[maxIndex]
              Jika ya, maxIndex = elemen
6)     Tukarkan nilai array[iterasi] dengan array[maxIndex]
```

Selection sort pada Algoritme 2 dengan pendekatan pencarian maksimum melakukan pengurutan data dengan cara mencari elemen terbesar dari baris data lalu ditukarkan dengan indeks elemen paling belakang yang belum terurut. Proses tersebut diulang hingga semua elemen menjadi terurut. Hasilnya berupa rangkaian data yang terurut secara *ascending* (menaik). Apabila ingin mengurutkan data secara *descending* (menurun) maka penukaran elemen terbesar dilakukan dengan indeks paling depan yang belum terurut.

Rancangan algoritme *selection sort hybrid* dibuat dengan menggabungkan konsep maksimum dan minimum pada algoritme *selection sort* biasa. Proses pencarian maksimum dan minimum digunakan untuk mengurutkan data dari depan (minimum) dan dari belakang (maksimum). Proses pencarian data maksimum dan data minimum dilakukan secara simultan dengan menggunakan *multithreading* sehingga proses pengurutan akan menjadi lebih cepat. Berikut ini adalah rancangan algoritme *selection sort hybrid*.

Algoritme 3. S*election sort hybrid*

1) Baca data ke dalam array 2) Inisialisasi nilai startIndexMin menjadi indeks awal dan endIndexMin menjadi indeks tengah 3) Inisialisasi nilai startIndexMax menjadi indeks tengah+1 dan endIndexMax menjadi indeks akhir 4) Buat dua buah object thread untuk melakukan pengurutan minimum dan maksimum	
5) Selama startIndexMin kurang dari sama dengan endIndexMin lakukan langkah 6 sampai 9 6) Tentukan minIndex = startIndexMin 7) Untuk elemen = startIndexMin+1 sampai endIndexMax lakukan langkah 8 8) Test apakah array[elemen] <= array[minIndex] Jika ya, minIndex = elemen 9) Tukar nilai array[startIndexMin] dengan array[minIndex]	5)Selama endIndexMax lebih besar sama dengan startIndexMax lakukan langkah 6 sampai 9 6)Tentukan maxIndex = endIndexMax 7)Untuk elemen = endIndexMax-1 sampai startIndexMin lakukan langkah 8 8)Test apakah array[elemen] > array[maxIndex] Jika ya, maxIndex = elemen 9)Tukarkan nilai array[endIndexMax] dengan array[maxIndex]

Pada Algoritme 3, langkah 5-9 menunjukkan bahwa pencarian dan pertukaran elemen minimum dan maksimum dilakukan secara terpisah pada dua *thread* yang berbeda namun dalam waktu yang simultan. Hal tersebut menandakan bahwa proses pencarian dilakukan dua arah yaitu menukar elemen minimum dengan elemen pada indeks paling depan yang belum terurut dan menukar elemen maksimum dengan elemen pada indeks paling belakang yang belum terurut. Proses tersebut diulang hingga semua elemen menjadi terurut. Hasilnya berupa rangkaian data yang terurut secara *ascending* (menaik). Apabila ingin mengurutkan data secara *descending* (menurun) maka penukaran elemen terkecil dilakukan dengan indeks paling belakang yang belum terurut dan penukaran elemen terbesar dilakukan dengan indeks paling depan yang belum terurut. Karena adanya mekanisme pengurutan dua arah yang dilakukan dalam waktu yang bersamaan maka proses pengurutan menjadi lebih cepat. Pengurutan dengan menggunakan *multithreading* menjadikan algoritme *selection sort hybrid* berbeda dari pendahulunya. Mekanisme ini dipilih supaya beberapa pekerjaan yang tidak saling bersinggungan dapat dilakukan segera dalam waktu yang simultan untuk menghemat waktu pengurutan.

3.4. Ilustrasi Algoritme

Untuk lebih memperjelas jalannya algoritme, berikut ini dijelaskan langkah-langkah pengurutan data dengan menggunakan algoritme *selection sort hybrid* secara *ascending*. Harap diperhatikan bahwa proses pengurutan dengan teknik minimum dan maksimum dilakukan secara terpisah melalui dua *thread* yang berbeda namun digambarkan bersamaan karena kedua *thread* berjalan secara simultan. Dimisalkan terdapat 10 data acak yaitu 20, 15, 8, 26, 5, 11, 31, 16, 3, 7.

20	15	8	26	5	11	31	16	3	7
0	1	2	3	4	5	6	7	8	9

Gambar 2. Proses Pengurutan Iterasi 1

Gambar 2 memperlihatkan data awal yang akan diurutkan. Langkah pertama adalah mencari elemen minimum dan maksimum dari data mulai dari *index* 0 (*beginIndex*) sampai *index* 9 (*endIndex*), maka didapat *minIndex* = 8 dengan elemen data 3 dan *maxIndex* = 6 dengan elemen data 31. Lalu tukarkan data pada *minIndex* dengan data pada *beginIndex* dan tukarkan pula data pada *maxIndex* dengan data pada *endIndex* sehingga dihasilkan urutan yang baru seperti yang diperlihatkan pada Gambar 3. Elemen data yang telah diarsir menandakan bahwa elemen tersebut sudah terurut sehingga tidak akan diproses kembali.

3	15	8	26	5	11	7	16	20	31
0	1	2	3	4	5	6	7	8	9

Gambar 3. Proses Pengurutan Iterasi 2

Gambar 3 memperlihatkan iterasi kedua, rentang pengurutan diperkecil dengan menambah *beginIndex* dengan 1 dan mengurangi *endIndex* dengan 1. Lalu proses pengurutan berlanjut dengan mencari elemen minimum dan maksimum dari data mulai dari *index* 1 (*beginIndex*) sampai *index* 8 (*endIndex*), maka didapat *minIndex* = 4 dengan elemen data 5 dan *maxIndex* = 3 dengan elemen data 26. Lalu tukarkan data pada *minIndex* dengan data pada *beginIndex* dan tukarkan pula data pada *maxIndex* dengan data pada *endIndex* sehingga dihasilkan urutan yang baru seperti yang diperlihatkan pada Gambar 4.

3	5	8	20	15	11	7	16	26	31
0	1	2	3	4	5	6	7	8	9

Gambar 4. Proses Pengurutan Iterasi 3

Gambar 4 memperlihatkan iterasi ketiga, rentang pengurutan diperkecil dengan menambah *beginIndex* dengan 1 dan mengurangi *endIndex* dengan 1. Lalu proses pengurutan berlanjut dengan mencari elemen minimum dan maksimum dari data mulai dari *index* 2 (*beginIndex*) sampai *index* 7 (*endIndex*), maka didapat *minIndex* = 6 dengan elemen data 7 dan *maxIndex* = 3 dengan elemen data 20. Lalu tukarkan data pada *minIndex* dengan data pada *beginIndex* dan tukarkan pula data pada *maxIndex* dengan data pada *endIndex* sehingga dihasilkan urutan yang baru seperti yang diperlihatkan pada Gambar 5.

3	5	7	16	15	11	8	20	26	31
0	1	2	3	4	5	6	7	8	9

Gambar 5. Proses Pengurutan Iterasi 4

Gambar 5 memperlihatkan iterasi keempat, rentang pengurutan diperkecil dengan menambah *beginIndex* dengan 1 dan mengurangi *endIndex* dengan 1. Lalu proses pengurutan berlanjut dengan mencari elemen minimum dan maksimum dari data mulai dari *index* 3 (*beginIndex*) sampai *index* 6 (*endIndex*), maka didapat *minIndex* = 6 dengan elemen data 8 dan *maxIndex* = 3 dengan elemen data 16. Lalu tukarkan data pada *minIndex* dengan data pada *beginIndex* dan tukarkan pula data pada *maxIndex* dengan data pada *endIndex* sehingga dihasilkan urutan yang baru seperti yang diperlihatkan pada Gambar 6.

3	5	7	8	15	11	16	20	26	31
0	1	2	3	4	5	6	7	8	9

Gambar 6. Proses Pengurutan Iterasi 5

Gambar 6 memperlihatkan iterasi kelima, rentang pengurutan diperkecil dengan menambah *beginIndex* dengan 1 dan mengurangi *endIndex* dengan 1. Lalu proses pengurutan berlanjut dengan mencari elemen minimum dan maksimum dari data mulai dari *index* 4 (*beginIndex*) sampai *index* 5 (*endIndex*), maka didapat *minIndex* = 5 dengan elemen data 11 dan *maxIndex* = 4 dengan elemen data 15. Lalu tukarkan data pada *minIndex* dengan data pada *beginIndex* dan tukarkan pula data pada *maxIndex* dengan data pada *endIndex* sehingga dihasilkan urutan yang baru seperti yang diperlihatkan pada Gambar 7.

3	5	7	8	11	15	16	20	26	31
0	1	2	3	4	5	6	7	8	9

Gambar 7. Proses Pengurutan Iterasi 6

Gambar 7 memperlihatkan kondisi data pada iterasi keenam. Karena semua data sudah selesai dikerjakan maka iterasi berhenti dan elemen data sudah terurut secara menaik (*ascending*).

Implementasi dilakukan dengan menggunakan bahasa pemrograman *Java* dengan spesifikasi komputer, prosesor *Intel Core* i3 M 350 2,27 GHz, RAM 6 GB dan dijalankan di sistem operasi *Windows* 7 32-*bit*.

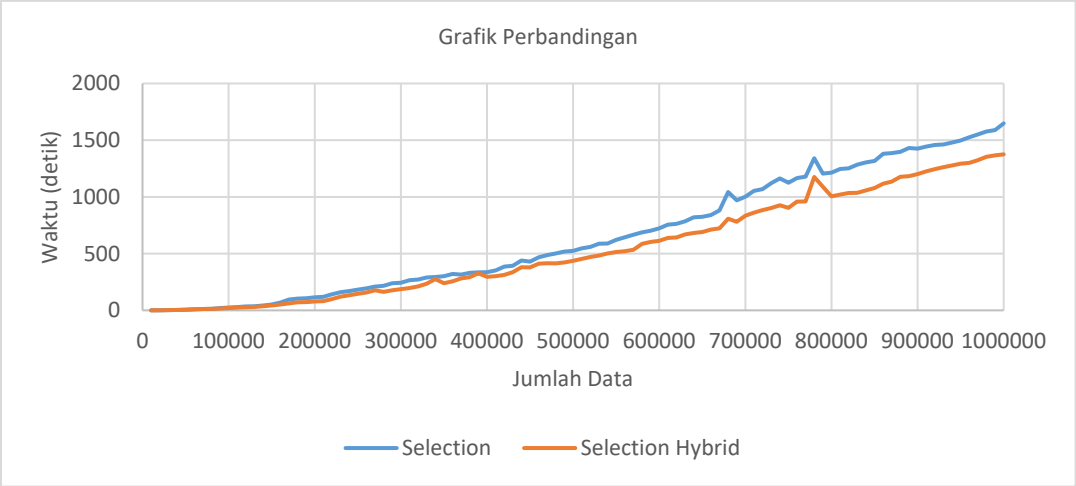
4. Hasil dan Diskusi

Pengujian dilakukan sebanyak 100 kali dengan menggunakan 100 *dataset* yang berbeda seperti yang telah di sebutkan sebelumnya. 100 kali pengujian dilakukan untuk melihat perbedaan waktu komputasi antara data dengan jumlah sedikit hingga banyak. Hal tersebut dilakukan untuk melihat algoritme mana yang lebih baik. Dari 100 pengujian tersebut diambil 20 hasil pengujian yang memperlihatkan performa terbaik dari *selection sort hybrid*. 20 pengujian tersebut ditampilkan pada Tabel 1.

Tabel 1. Hasil 20 pengujian

No	Jumlah Data	Waktu Pengurutan (detik)		Selisih Waktu (detik)
		Selection	*Selection Hybrid*	
1	680000	1042,9842970	807,1526406	235,8316564
2	730000	1120,8621070	901,9071904	218,9549166
3	740000	1163,4534733	925,9247560	237,5287173
4	750000	1125,7906044	904,3170821	221,4735223
5	770000	1179,2511646	960,3952794	218,8558852
6	810000	1245,8634738	1020,6826071	225,1808667
7	830000	1284,8624393	1035,9134483	248,9489910
8	840000	1304,4938387	1057,7873049	246,7065338
9	850000	1316,8418256	1078,3386530	238,5031726
10	860000	1380,5778091	1116,4322089	264,1456002
11	870000	1385,9941337	1135,2906427	250,7034910
12	880000	1397,4258379	1177,3595880	220,0662499
13	890000	1431,8280940	1182,3385892	249,4895048
14	900000	1425,8519330	1200,7130238	225,1389092
15	910000	1443,544166	1225,604838	217,939328
16	960000	1525,5476457	1300,3392958	225,2083499
17	970000	1551,2168892	1324,6206291	226,5962601
18	980000	1576,7110282	1354,3041140	222,4069142
19	990000	1589,3936462	1366,7332913	222,6603549
20	1000000	1648,3414592	1375,3316012	273,0098580

Tabel 1 memperlihatkan bahwa waktu yang dibutuhkan *selection sort hybrid* lebih cepat dibandingkan dengan *selection sort* biasa untuk setiap *dataset* yang sama. Selisih waktu menunjukkan perbedaan waktu selesai pengurutan antara kedua algoritme. Selisih waktu yang besar adalah 273,0098580 detik dengan jumlah data 1.000.000 dan selisih waktu terkecil 217,939328 dengan jumlah data 910.000.



Gambar 8. Grafik Perbandingan S*election* Sort dan S*election Sort Hybrid*

Gambar 8 memperlihatkan perbandingan algoritme *selection sort* dan *selection sort hybrid* dari segi waktu komputasi dan jumlah data untuk seluruh *dataset*. Terlihat bahwa waktu pengurutan dengan algoritme *selection sort hybrid* selalu berada di bawah algoritme *selection sort* biasa. Hal tersebut menunjukkan bahwa algoritme *selection sort hybrid* lebih cepat dari algoritme *selection sort* biasa pada semua kasus.

Penelitian ini telah berhasil mengimplementasikan algoritme *selection sort hybrid* dengan menggabungkan algoritme *selection sort* dengan pencarian minimum dan maksimum dengan menggunakan mekanisme *multithreading*. Pada semua pengujian yang telah dilakukan didapatkan bahwa algoritme *selection sort hybrid* lebih efisien dibandingkan dengan algoritme *selection sort* biasa. Gambar 8 juga memperlihatkan bahwa semakin besar data yang diurutkan maka selisih waktu eksekusi kedua algoritme akan relatif meningkat, dimana algoritme *selection sort hybrid* lebih unggul dari algoritme *selection sort* biasa. Hal tersebut dapat terjadi karena adanya mekanisme pengurutan data dari depan dan dari belakang yang dilakukan secara simultan, artinya waktu yang digunakan untuk pertukaran data akan berkurang setengah dari algoritme *selection sort* biasa. Sedangkan waktu yang dibutuhkan untuk melakukan pencarian nilai minimum dan maksimum akan sama dengan *selection sort* biasa. Maka keunggulan utama dari algoritme *selection sort hybrid* terletak pada pengurangan proses pertukaran data untuk mengurangi iterasi algoritme yang berdampak pada berkurangnya waktu eksekusi.

5. Simpulan dan Saran

Berdasarkan uraian yang dibahas dalam penelitian ini dapat disimpulkan bahwa algoritme *selection sort hybrid* lebih efisien dibandingkan dengan algoritme *selection sort* biasa. Hal tersebut dapat terjadi karena terdapat proses pengurutan data dua arah yaitu dari depan (minimum) dan dari belakang (maksimum) yang dilakukan secara simultan dengan menggunakan *multithreading*. Saran yang dapat diberikan berdasarkan hasil yang didapat dalam penelitian ini yaitu perlu diterapkan untuk pengurutan data obyek atau pada kasus nyata sehingga manfaatnya dapat lebih diperoleh untuk menyelesaikan kasus sehari-hari.

Referensi

[1] K. Jouper dan H. Nordin, "Performance Analysis of Multithreaded Sorting Algorithms," Thesis, Blekinge Institute of Technology, 2015.

[2] M. T. Goodrich, R. Tamassia dan M. H. Goldwasser, Data Structure & Algorithms in Java Sixth Edition, United States of America: John Wiley & Sons, Inc., 2014.

[3] P. Hyde, Java Thread Programming, Indianapolis: SAMS, 1999.

[4] V. P. Kulalvaimozhi, M. Muthulakshmi, R. Mariselvi, G. S. Devi, C. Rajalakshmi dan C. Durai, Performance analysis of sorting algorithm, *International Journal of Computer Science and Mobile Computing (IJCSMC)*, Vol 4(1), pp. 291-306, 2015.

[5] K. S. Al-Kharabsheh, I. M. AlTurani, A. M. I. AlTurani dan N. I. Zanoon, Review on sorting algorithms a comparative study, *International Journal of Computer Science and Security (IJCSS)*, Vol. 7(3), pp. 120-126, 2013.

[6] D. Rajagopal dan K. Thilakavalli, Different sorting algorithm's comparison based upon the time complexity, *International Journal of u- and e- Service, Science and Technology*, Vol. 9(8), pp. 287-296, 2016.

[7] N. Akhter, M. Idrees dan F. Rehman, Sorting algorithms – a comparative study, *International Journal of Computer Science and Information Security (IJCSIS),* Vol. 14(12), pp. 930-936, 2016.

[8] J. Alnihoud and R. Mansi, An enhancement of major *sorting* algorithms, *The International Arab Journal of Information Technology*, Vol. 7(1), pp. 55-61, 2010.

[9] J. B. Hayfron-Acquah dan O. Appiah, Improved selection sort algorithm, *International Journal of Computer Applications*, Vol. 110(5), pp. 29-33, 2015.

[10] M. Khairullah, Enhancing worst sorting algorithms, *International Journal of Advanced Science and Technology*, Vol. 56, pp. 13-26, 2013.

[11] S. Majumdar, I. Jain and A. Gawade, Parallel quick sort using thread pool pattern, *International Journal of Computer Applications*, Vol. 136(7), pp. 36-41, 2016.

[12] H. M. Deitel and P. J. Deitel, Java: How to Program 8th Edition, Pearson: London, 2009.