Analisis Kinerja Genetic Algorithm yang diakselerasi untuk Travelling Salesman Problem pada Platform Multicore CPU dan CUDA

Alexander Thomas Kurniawan Pratomo, Henry Novianus Palit, Darian Gunamardi.
Program Studi Informatika Fakultas Teknologi Industri Universitas Kristen Petra
Jl. Siwalankerto 121 – 131 Surabaya 60236
Telp. (031) – 2983455, Fax. (031) - 8417658
E-Mail: a.thomas.kurniawan@gmail.com, hnpalit@petra.ac.id, dariang@petra.ac.id

ABSTRAK

Kemajuan di bidang teknologi membawa berbagai tantangan dan kesempatan baru. Salah satunya adalah kesempatan untuk mempercepat suatu algoritma dimana algoritma ini biasanya memakan waktu sangat lama. *Genetic Algorithm* adalah salah satu algoritma yang dapat dipercepat dengan memanfaatkan perkembangan tersebut. Salah satu car mempercepat algoritma ini dapat melakukan pengubahan parameter, tetapi tidak semua parameter yang diubah ini dapat mempertahankan kualitas hasil dari algoritma dibandingkan dengan metode sebelumnya yang belum diakselerasi. Penelitian ini menggunakan metode *parallel computing* menggunakan teknologi *Multicore CPU* dan *GPU* untuk mempercepat algoritma tanpa mengubah parameter. Pada penelitian ini peneliti ingin menganalisa perubahan kinerja dari metode *Genetic Algorithm* ketika dilakukan paralelisasi. Penelitian ini menggunakan *CUDA* untuk melakukan paralelisasi yang menggunakan *GPU* dan menggunakan *OpenMP API* untuk melakukan paralelisasi yang menggunakan teknologi tersebut. Teknologi yang digunakan adalah *platform CUDA* dan *API OpenMP* untuk masing-masing GPU dan CPU. Selain teknologi tersebut, pemilihan bagian yang diparalelisasi juga mempengaruhi secara besar bagaimana performa dari algoritma. Berdasarkan hasil pengujian yang telah dilakukan, metode *Genetic Algorithm* dapat dipercepat dengan paralelisasi menggunakan *OpenMP* pada *CPU* dan *CUDA* pada *GPU*.

Kata Kunci: *Genetic Algorithm*, *Parallel Computing*, *Compute Unified Device Architecture*, *Graphics Processing Unit*, *Multicore Central Processing Unit*, *OpenMP*

ABSTRACT

Advancement in technology brings about both new challenges and opportunities. One of which is the opportunity to accelerate an algorithm which usually took a very long time to finish. Genetic algorithm is one such algorithm that can be accelerated using these advancements. Certain ways to accelerate this algorithm is done by tuning the parameters, but these methods are usually unable to retain the quality that is obtained from previous, non-accelerated method. This research applies parallel computing using the multi-core technology of CPU and GPU to accelerate genetic algorithm without any changes to the parameters. The purpose of the research is to analyze the differences in performance of the algorithm upon being accelerated with the technologies. The technologies used are CUDA platform and OpenMP API for GPU and CPU respectively. Aside from the technology itself, choosing the algorithm segments on which the implementation is done will also greatly affect the performance of the algorithm. According the testing results, Genetic Algorithm can be accelerated with parallelization using either OpenMP with CPU or CUDA with GPU.

Keywords: *Genetic Algorithm*, *Parallel Computing*, *Compute Unified Device Architecture*, *Graphics Processing Unit*, *Multicore Central Processing Unit*, *OpenMP*

1. PENDAHULUAN

Travelling Salesman Problem (TSP) adalah masalah tentang pengaturan perjalanan dimana seseorang harus pengunjungi tiap titik yang ada tepat sekali dan kembali ke kota awal. Bentuk lain dari TSP yang memiliki masalah yang mirip adalah penjadwalan kru, pengaturan rute kendaraan, dan peletakan komponen pada *Printed Circuit Board*. Solusi dari TSP dapat didapatkan dengan melakukan pendekatan menggunakan Genetic Algorithm. Tetapi waktu yang dibutuhkan untuk melakukan pendekatan ini bertambah secara eksponensial ketika jumlah kota yang ada bertambah.

Genetic Algorithm adalah algoritma yang digunakan untuk optimisasi yang terinpirasi oleh proses biologis. Proses biologis yang diambil adalah proses evolusi, persilangan dan eliminasi individu yang tidak dapat bertahan dalam alam tersebut. Individu tersebut dinilai berdasarkan DNA apakah dia bertahan atau tidak. Dari persilangan dan proses eliminasi tersebut, masing-masing individu memiliki kesempatan berevolusi ketika dia disilangkan dengan individu lain, dengan harapan individu yang baru tersebut dapat bertahan lebih baik di alam. Tetapi proses evolusi ini dapat memakan waktu karena kompleksnya sebuah individu maupun jumlah dari individu yang masif.

Tetapi proses ini dapat dipercepat dengan penggunaan tipe komputasi paralel. Komputasi paralel adalah proses komputasi dimana beberapa data dihitung secara bersama-sama menggunakan beberapa *core* yang tersedia di dalam *Central Processing Unit* (CPU) yang muncul karena clock pada *CPU* tidak dapat menjadi lebih tinggi lagi. Selain itu *Graphics Processing Units* (GPU) juga dapat dimanfaatkan untuk *general purpose computing on graphics processing units* (GPGPU). Tetapi, proses mengubah suatu program serial menjadi paralel tidak semudah membalikkan telapak tangan. Suatu program tersebut harus dilakukan profiling untuk menentukan titik-titik panas suatu program. Tetapi titik panas tersebut tidak semuanya dapat dilakukan paralelisasi. Selain itu, kombinasi paralelisasi dari beberapa titik panas tersebut juga dapat mempengaruhi kecepatan suatu program.

Dari sisi komputasi menggunakan *CPU*, salah satu cara yang digunakan adalah menggunakan *OpenMP*. *OpenMP* adalah sekumpulan *compiler directive*, *library routines*, dan *enviroment*

variables yang dapat digunakan untuk menunjukan paralelisasi tingkat tinggi dalam *Fortran* maupun *C/C++* [10]. *OpenMP* diharapkan dapat membagi beberapa bagian program yang dapat dilakukan paralelisasi. Suatu bagian program dapat diparallelisasi jika proses tersebut tidak membutuhkan suatu hasil yang dihasilkan oleh proses lainnya. Cara lain yang dapat digunakan adalah menggunakan *PThread* pada sistem operasi *Linux* dan *Win32* pada sistem operasi *Windows*. Kedua cara tersebut memiliki kelebihan dapat melakukan kontrol lebih detail pada proses yang diparallel daripada *OpenMP*, tetapi memiliki kekurangan harus melakukan perubahan bentuk program ketika ingin dijalankan sebagai pada salah satu *thread* saja dan harus melakukan *compile* ulang ketika ingin menyesuaikan jumlah *thread*. Sedangkan *OpenMP* dapat berjalan pada sistem *operasi Windows* maupun *Linux* dan tidak membutuhkan compile ulang ketika ingin dijalankan pada jumlah *thread* yang berbeda dan secara langsung menggunakan jumlah *thread* yang ada [4].

Di sisi lain, GPGPU dapat menggunakan *Compute Unified Device Architecture* (CUDA). *CUDA* adalah platform dan programming yang kembangkan oleh *Nvidia*. Tetapi tidak semua program tersebut dapat dijalankan didalam *GPU*. Sedangkan hanya sebagian program saja yang dapat diparalel yang dijalankan didalam *GPU* dan program yang hanya dapat dijalankan secara sekuensial tetap dijalankan dialam *CPU*. [8] Tetapi tidak semua titik panas yang didapatkan tersebut dapat diparalelisasi didalam *GPU*. Selain itu, walaupun titik panas tersebut dapat diparalelisasi didalam *GPU*, titik panas belum tentu dapat berjalan lebih cepat. Salah satu penyebabnya adalah overhead yang terjadi ketika ada data yang perlu dipindahkan dari *random access memory* (*RAM*) yang dimiliki oleh *CPU* ke *RAM* yang dimiliki oleh *GPU* maupun sebaliknya [11]

Pada penelitian ini *Genetic Algorithm* akan diparalelisasi pada platform *multicore CPU* dan *CUDA* untuk mendapatkan perbandingan performa dari kedua platform tersebut. Sehingga didapatkan tren kinerja dari kedua platform tersebut dan dapat diprediksi ketika bentuk genetik semakin kompleks.

2. TINJAUAN STUDI

2.1. *Parallel Computing*

Parallel Computing adalah bentuk dari komputasi dimana banyak kalkulasi dilakukan secara bersama-sama, setelah prinsip masalah yang besar dapat dibagi menjadi lebih kecil. Paralel computing menggunakan beberapa perangkat komputasi. Kemudian, masalah yang telah dibagi menjadi lebih kecil tersebut dieksekusi dimasing masing core [1].

2.2. OpenMP

OpenMP adalah Application Programming Interface (API) untuk parallisasi dengan memori yang dipakai bersamaan menggunakan C, C++, atau Fortran. API ini terdiri dari arahan compiler yang digunakan untuk menunjukan dan mengkontrol paralelisasi, yang dikuatkan oleh runtime function, dan enviroment variables. User diharuskan untuk melakukan indentifikasi terhadap paralelisasi yang memasukan struktur control yang tepat pada [9].

2.3. CUDA

Compute Unified Device Architecture (CUDA) adalah platorm parallel computing dan programming model yang dikembangkan oleh Nvidia untuk GPGPU. CUDA digunakan untuk melakukan paralelisasi program dengan cara menjalankannya pada GPU yang dibuat oleh Nvidia. Dengan CUDA, user dapat menjalankan bagian dari program yang membutuhkan komputasi intensif di dalam GPU sehingga program dapat berjalan lebih cepat. Bahkan dengan CUDA, beberapa GPU yang terdapat dalam satu sistem dapat digunakan secara bersamaan untuk menjalankan process yang sama [7].

2.4. Standard Template Library

Ivan Horton [3] mendefiniskan *Standard Template Lirbary* (STL) sebagai sekumpulan alat ampuh dan luas yang digunakan untuk mengorganisasi dan memproses data. STL dibuat dalam bentuk *template* sehingga data yang memenuhi beberapa syarat tertentu yang dapat diolah oleh STL ini. STL dapat dibagi menjadi 3 bagian secara konseptual, yaitu :

2.4.1.Containers

Containers mendefinisikan tempat penyimpanan yang berguna untuk menyimpan dan mengatur data. Bagian dari STL yang termasuk *containers* adalah *array*, *vector*, *stack*, *queue*, *deque*, *list*, *forward_list*, *set*, *unordered_set*, *map*, dan *unordered_map*.

2.4.2. Iterators

Iterators adalah objek yang berperilaku seperti penunjuk yang dan digunakan untuk mereferensikan urutan objek dalam tempat. Bagian dari STL yang termasuk *iterators* adalah *iterator*.

2.4.3. Algorithms

Algorithms mendefinisikan algoritma-algoritma yang dapat diaplikasikan dalam suatu set yang disimpan di tempat penyimpanan. Bagian dari STL yang termasuk *algorithms* adalah *algorithms*.

2.4.4. Numerics

Numerics mendefinisikan fungsi-fungsi pengelolahan angka, termasuk pemrosesan angka dari set pada elemen. Bagian tempat menyimpan dan mengatur data. Bagian ini juga termasuk fungsi fungsi tambahan untuk *random number generation*. Bagian dari STL yang termasuk *numerics* adalah *complex*, *cmath*, *valarray*, *numeric*, *random*, *ratio*.

2.5. Thrust

Nvidia [9] mendefiniskan *Thrust* sebagai *template library* untuk CUDA yang didasarkan dari STL. Thrust menyediakan banyak primitif data parallel seperti scan, sort, dan reduce yang jika digabung dapat digunakan untuk implementasi algoritma compleks. STL dapat dibagi menjadi 3 bagian, yaitu :

2.5.1. Vectors

Pada *thrust* ada 2 jenis *vector*, yaitu *host_vector* dan *device_vector*. Seperti namanya, data pada *host_vector* disimpan pada memori *host* sedangkan *device_vector* tinggal pada memori device *GPU*. *Vector* container sama seperti *vector* pada STL. Selain itu, *host_vector* dan *device_vector* adalah kontainer generik

2.5.2. Iterators

Sama seperti STL, Thrust juga menyediakan iterator. Tetapi karena Thrust memiliki 2 jenis vector, maka jenis iterator pada Thurst juga menjadi 2 macam yaitu untuk *device* dan *host*. Selain itu untuk mengambil *pointer* yang akan digunakan untuk passing ke kernel CUDA dapat menggunakan *raw_pointer_cast*.

2.5.3. Algorithms

Thrust menyediakan banyak algoritma parallel. Banyak dari algoritma ini memiliki metode yang sama seperti pada STL, dan ketika ada pada STL, maka nama dari algoritmanya akan sama.

2.6. *Travelling Salesman Problem*

Federiko Greco [2] mendifinisikan *Travelling Salesman Problem* (*TSP*) sebagai representasi kelas masalah yang juga dikenal sebagai masalatah tentang optimasi suatu kombinasi. Dalam bentuk yang umum, peta suatu kota diberikan kepada "*salesman*" dan dia harus mengunjungi semua kota sekali.

2.7. *Genetic Algorithm*

Oliver Kramer [5] mendefinisikan *Genetic Algorithms* sebagai pendekatan pencarian heuristic yang dapat diaplikasikan untuk berbagai masalah optimisasi. *Genetic Algorithms* berbasiskan evolusi. Persilangan dan mendapatkan keturunan untuk berevolusi adalah prinsip utama dari evolusi. Beberapa proses dari *Genetic Algorithm* antara lain adalah:

2.7.1. Crossover

Crossover adalah operator yang membuat kombinasi dari genetik dari dua atau lebih individu. Motivasi dari operator tersebut adalah kedua bagian genetik tersebut mungkin memiliki bagian dari solusi yang terbaik dan ketika digabung akan melampaui orang tuanya.

2.7.1.1. Order 1 Crossover

Order 1 Crossover adalah salah satu metode yang dapat digunakan untuk *crossover* pada kasus TSP. Metode ini membuat individu turunannya dengan memilih sebagian dari genetiknya dari salah satu orang tua dan menjaga urutan elemen secara relatif dari orang tua lain.

2.7.2. Mutation

Mutation adalah operator yang mengubah solusi dengan mengganggu mereka. *Mutation* ini berdasarkan perubahan yang acak. Kemudian kekuatan dari gangguan ini disebut *mutation rate*. Ada tiga syarat untuk *mutation*, yaitu *reachability*, *unbiasedness*, dan *scalability*. *Reachability* adalah setiap poin dari ruang solusi harus bisa dicapai dari titik manapun. *Unbiasedness* adalah *mutation* tidak menyebabkan kecenderungan pencarian ke suatu arah. *Scalability* adalah kekuatan dari mutasi dapat beradaptasi.

2.7.3. Selection

Agar dapat menuju ke solusi yang optimal, anak yang memiliki solusi yang paling baik harus dipilih sebagai populasi orang tua untuk generasi selanjutnya. Proses ini berdasarkan nilai *fitness* di populasi. Dalam kasus *minimization*, nilai fitness kecil lebih diharapkan dan sebaliknya untuk *maximization* problem. Metode seleksi yang diuji dalam penelitian ini antara lain:

Tournament Selection

Tournament Selection ini dilakukan dengan memilih beberapa individu dipilih secara acak dan didalam individu yang terpilih itu akan dipilih individu yang memiliki *fitness* terbaik.

Roulette Wheel Selection

Roulette Wheel Selection atau yang juga dikenal *Fitness Proportional Selection* ini memilih suatu individu secara acak dengan distribusi *uniform*. Probabilitas suatu individu dipilih tergantung dengan *fitness* dari masing-masing individu.

2.8. *Amdahl's Law*

Menurut McCool [6], Amdahl berargumen bahwa bagian dari suatu waktu yang digunakan program (T_1) dibagi menjadi dua bagian, yaitu waktu yang digunakan program untuk menjalankan pekerjaan yang tidak dapat diparalelisasi, dan yang dapat diparalelisasi, masing masing dapat disebut W_{ser} *dan* W_{par}.

3. ANALISA DAN DESAIN SISTEM

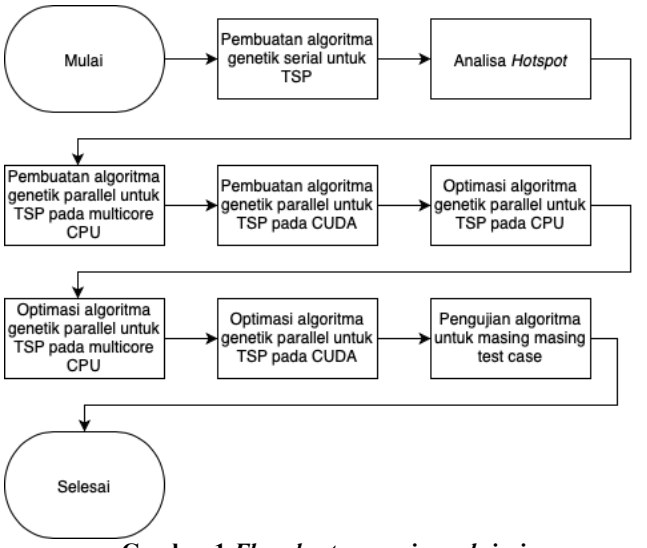
3.1. Analisa

Langkah pertama yang dilakukan adalah melakukan pembuatan algoritma genetik versi serial untuk TSP. *Encoding* yang digunakan adalah permutasi, dimana isi gen dari adalah kota yang dilewati. Sehingga genetik individu tersebut berisi urutan kota yang dilewati oleh individu tersebut. Kemudian masing-masing individu tersebut memiliki nilai *fitness* yang dievaluasi dengan menghitung jumlah jarak yang dilewati oleh individu tersebut. Semakin kecil nilai *fitness* dari individu tersebut semakin bagus individu tersebut.

Kemudian dilakukan analisa *Hotspot* menggunakan *VTune Amplifier* untuk mendapatkan bagian mana dari program yang paling membutuhkan waktu. Dari hasil analisa hotspot tersebut dapat ditentukan bagian mana saja yang dilakukan paralelisasi. Dari hasil paralelisasi tersebut akan dikermbangkan juga cara untuk memanfaatkan GPU agar paralelisasi dapat berjalan lebih efisien. Hasil *profiling* disertakan pada Tabel 1. Hasil yang disertakan disini hanyalah 10 teratas untuk memudahkan pembacaan. Walaupun hanya 10 function yang ditampilkan, tetapi banyak proses lain yang menurut hipotesa saya juga akan memperkecil *running time*. Selain itu yang saya sertakan disini hanyalah yang bisa saya kontrol *running time*-nya.

Tabel 1. ***Tabel Daftar Running Time***

Nama function	*runnning time*
XOCross	75,37
Mutation	72,99
CalculateGenesFitness	62,81
XORankedPairing	31,19
XOCutting	28,42
XOGeneChecking	21,91
SelectionElitism	20,62
GenerateGene	4,03
std::stable_sort	1,14
XOJoinGene	0,87



Gambar 1 ***Flowchart pengerjaan skripsi***

Pada Gambar 1 dapat dilihat alur pengerjaan skripsi yang akan dilakukan hingga selesai.

3.2. Desain

Subbab ini akan membahas mengenai desain alur dari sistem baik versi *serial* maupun *parallel* yang digunakan oleh program.

3.1.1. Garis Besar Kerja Sistem Versi Serial

Sistem terbagi menjadi dua bagian yaitu *precomputation* dan *iteration*. Proses yang terjadi pada bagian *precomputation* hanya dilakukan sekali. Sistem menyimpan dua bagian data, yaitu data dari titik titik yang harus dilewati dan populasi yang sedang diproses saat itu. Proses yang dilakukan pada tahap *precomputation* meliputi *Prepare Data Set*, dan *Initialize Population*. Kemudian proses yang dilakukan pada tiap *iteration* adalah *Cross Over*, *Mutation*, *Reserve Elite Gene*, dan *Elimination*. *Stop condition* dapat berupa waktu maupun nilai *fitness.*

3.1.1.1. Prepare Data Set

Proses ini adalah salah satu bagian yang terdapat pada proses *Prepare Data Set*. *Dataset* ini memiliki bentuk *TSPLib*. *TSPLib* ini terbagi menjadi data, yaitu *header* dan *data*. *Header* ini berisi *metadata* yang dimiliki oleh *dataset* ini. Bagian kedua adalah daftar titik-titik (*nodes*) yang harus dikunjungi. Pertama dimulai dengan melakukan pengambilan semua titik titik yang harus dilewati yang terdapat dalam file yang memiliki format *TSPLIB*. Kemudian melakukan dari titik titik tersebut dibuatlah daftar hasil perhitungan jarak dari titik satu ke titik lainnya. Daftar perhitungan jarak ini berbentuk *array* satu dimensi untuk mempercepat akses.

Load nodes coordinates

Proses ini adalah salah satu bagian yang terdapat pada proses *Prepare Data Set*. Dengan posisi file telah terbuka, langkah pertama adalah menjadikan keadaan pertama belum memasuki bagian *node*. Langkah kedua adalah mengambil kata pertama dari baris tersebut. Jika sudah memasuki bagian koordinat dan kata pertama bukan "*EOF*", maka akan membaca koordinat dari titik tersebut. Jika yang dibaca adalah kata "*DIMENSION*", maka ukuran daftar maka ukuran dari daftar titik dan matriks jarak akan disesuaikan dan juga menentukan ukuran populasi. Jika yang dibaca adalah kata "*NODE_COORD_SECTION*", maka akan memasuki mode koordinat.

Calculate distance to each node

Proses ini adalah salah satu bagian yang terdapat pada proses *Prepare Data Set*. Jarak dari tiap titik disimpan dalam bentuk matriks. Setiap isi pada matriks dilakukan perhitungan untuk menentukan isi dari matriks. Kota asal ditentukan dari indeks matriks dibagi dengan jumlah kota dan kota tujuan ditentukan dari indeks matriks dimodulus dengan jumlah kota. Jarak titik satu ke titik lain dihitung dengan menggunakan *eucledian distance* atau mencari akar pangkat dari pemangkatan dua selisih koordinat sumbu x kedua titik ditambah pemangkatan dua selisih koordinat sumbu y kedua titik.

3.1.1.2. Initialize Population

Setelah data set yang harus diselesaikan telah dibuat, langkah selanjutnya adalah membuat individu-individu yang bentuk genetiknya adalah solusi dari TSP. Panjang dari genetik dari masing-masing individu dan jumlah dari individu sendiri bergantung pada jumlah kota yang dilewati. Langkah pertama pada proses ini adalah mempersiapkan tempat untuk menyimpan individu dan masing-masing genetiknya. Kemudian genetik dari masing masing individu diisi dengan urutan kota secara random. Setelah masing masing individu telah diisi, dihitung nilai *fitness* untuk tiap individu.

Generate Unvisisted Node List

Proses ini adalah salah satu bagian yang terdapat pada proses *Initialize Population*. Langkah pertama yang dilakukan adalah persiapan tempat untuk daftar kota yang belum dikunjungi. Kemudian tempat tersebut diisi oleh angka dari nol sampai jumlah kota dikurangi satu.

Generate Each Individual Randomly

Proses ini adalah salah satu bagian yang terdapat pada proses *Initialize Population*. Langkah pertama yang memilih satu titik secara acak dari daftar dan memasukannya ke individu. Kemudian titik tersebut dipindah ke tempat dimana titik tersebut tidak pernah tertunjuk ketika *range* dari nilai yang diacak dikurangi.

Calculate Fitness

Proses ini adalah salah satu bagian yang terdapat pada proses *Initialize Population*. Langkah pertama yang dilakukan adalah melakukan mengubah fitness individu tersebut menjadi nol. Kemudian menambahkan titik titik jarak dari nilai gen satu menjuju gen sebelahnya.

3.1.1.3. Crossover

Proses pertama yang dilakukan oleh bagian *iteration* adalah *crossover*. Hal pertama adalah memilih *parent* yang akan dilakukan *crossover* menggunakan metode *Roulette Wheel*. Metode *crossover* yang dilakukan adalah *Order 1 Crossver*. Setelah jumlah *parent* yang terpilih terpenuhi, proses selanjutnya adalah menenentukan titik potong untuk masing-masing pasangan secara acak. Dari titik potong yang telah ditentukan, dilakukan pencatatan gen mana yang termasuk didalam titik potong tersebut. Pencatatan ini dilakukan agar tidak selalu melakukan pengecekan untuk setiap gen. Setelah semua individu *anak* telah terbentuk, langkah selanjutnya adalah penggabungan kedua populasi tersebut untuk proses selanjutnya.

Generate Parent Pairing

Proses ini adalah salah satu bagian yang terdapat pada proses *Crossover*. Langkah pertama yang dilakukan adalah melakukan *Roulette Wheel* untuk *parent* pertama. Kemudian melakukan *Roulette Wheel* untuk *parent* kedua dengan memastikan *parent* pertama tidak pernah terpilih. Algoritma dari proses ini dapat dilihat pada

Generate Cutting Point

Proses ini adalah salah satu bagian yang terdapat pada proses *Crossover*. Fungsi ini mengambil dua titik potong untuk tiap pasangan dan mencatatnya ada suatu memo. Titik potong pertama dipastikan tidak akan lebih besar dari titik potong kedua.

Fill Child Gene with Parent Gene

Proses ini adalah salah satu bagian yang terdapat pada proses *Crossover*. Fungsi ini berguna untuk memindahkan genetik dari *child* dengan genetik dari *parent*. *Parent* yang dipilih sesuai dari pasangan yang ditentukan sebelumnya.

Memoing Gene

Proses ini adalah salah satu bagian yang terdapat pada proses *Crossover*. Fungsi ini berguna untuk mencatat nilai genetik apa saja termasuk dalam kedua titik potong yang telah ditentukan sebelumnya. Bagian ini dilakukan agar tidak melakukan pengecekan secara terus menerus ketika ada gen yang akan dimasukan.

Exchange Parent Gene

Proses ini adalah salah satu bagian yang terdapat pada proses *Crossover*. Proses ini berfungi untuk menukar urutan gen agar sesuai dengan urutan gen pada *parent* satunya. Dengan memanfaatkan memo yang dibuat selanjutnya sebagai catatan gen apa saja yang dibekukan, proses ini dapat berjalan lebih cepat.

Combine Parent and Child Population

Proses ini adalah bagian terakhir yang terdapat pada proses *Crossover*. Langkah pertama yang dilakukan adalah persiapan tempat untuk kedua populasi tersebut. Kemudian tempat tersebut diisi oleh populasi *parent* dan diikuti oleh populasi *child*.

3.1.1.4. Mutation

Langkah pertama pada proses ini adalah pembuatan angka acak dari 0 sampai 100. Ketika angka yang dibuat kurang dari atau sama dengan *mutation rate*, maka akan dipilih 2 gen secara acak. Untuk 2 gen yang terpilih tersebut, letak masing-masing gen ditukar. Proses tersebut dilakukan untuk semua individu yang ada pada populasi.

3.1.1.5. Reserve Elite Individual

Sebelum dilakukan *reserve* individu, untuk setiap individu dihitung masing-masing *fitness*-nya. *Fitness* yang telah dihitung sebelumnya digunakan untuk menentukan peringkat dari masing masing individu. Untuk beberapa individu yang peringkat tertinggi, akan dicatat untuk masuk sebagai kandidat generasi selanjutnya. Selain itu agar individu yang sama tidak terpilih lagi, maka individu-individu tersebut juga dikeluarkan dari daftar populasi yang diuji.

3.1.1.6. Selection

Setelah individu yang termasuk elit telah di simpan untuk generasi selanjutnya, langkah selanjutnya adalah melakukan eliminasi individu yang tersisa. Metode seleksi yang ada antara lain:

Roulette Wheel Selection

Untuk metode ini, langkah pertama yang dilakukan adalah menentukan *force* dengan nilai acal. Kemudian *force* tersebut akan dikurangi dengan weight dari masing masing individu satu per satu. Jika *force* tersebut habis jika dikurangkan dengan weight suatu individu, maka individu tersebut akan ditambahkan dalam daftar kandidat dan dihapus dari daftar populasi yang diuji. Proses ini dilakukan hingga jumlah kandidat sama dengan jumlah populasi awal.

Tournament Selection

Untuk metode ini, langkah pertama yang dilakukan adalah memilih beberapa individu secara acak. Individu yang memiliki nilai weight terbaik akan ditambahkan dalam daftar kandidat dan dihapus dari daftar populasi yang diuji. Proses ini dilakukan hingga jumlah kandidat sama dengan jumlah populasi awal.

Partitioned Roulette Wheel Selection

Pada metode *roulette wheel selection*, pemilihan individu selanjutnya tidak dapat dilakukan sebelum pemilihan individu saat itu selesai karena ada kemungkinan untuk individu yang sama terpilih lagi jika *roulette wheel* dilakukan secara bersamaan. Demikian juga pada metode *tournament selection*, karena yang dipilih adalah individu yang memiliki nilai *fitness* yang paling besar. Jika *tournament selection* ini dijalankan bersamaan, maka kejadian seperti metode sebelumnya juga akan terjadi. Oleh karena itu digunakanlah metode ini agar tidak ada individu yang sama, maka individu yang dievaluasi dipilih secara urut agar tidak ada individu yang dievaluasi lebih dari satu kali. Kemudian indivitu tersebut dilakukan *roulette wheel* untuk menjaga agar tetap acak.

Pada metode ini, langkah pertama yang dilakukan adalah memilih beberapa individu yang bersebelahan. Kemudian ditentukan *force* dengan nilai acak. Kemudian *force* tersebut akan dikurangi dengan beban dari masing masing individu satu per satu. Jika *force* tersebut habis jika dikurangkan dengan beban suatu individu, maka individu tersebut akan ditambahkan dalam daftar kandidat dan dihapus dari daftar populasi yang diuji. Proses ini dilakukan hingga jumlah kandidat sama dengan jumlah populasi awal. Individu selanjutnya yang dipilih dimulai dari individu yang berada disamping individu terakhir.

3.1.2. Garis Besar Kerja Sistem Versi Parallel

Paralelisasi untuk metode ini dilakukan dalam beberapa tingkat karena tidak semuanya dapat diparalelisasi dalam satu tingkat. Pada proses:

a. *Calculate distance to each node*,
b. *Generate unvisited node list* (*GPU*),
c. *Calculate fitness (GPU)*,
d. *Fill child gene with parent gene*,
e. Memoing gene,
f. *Combine parent and child population*,

paralelisasi dilakukan pada tingkat element. Sedangkan pada proses:

a. *Generate unvisited node list* (*CPU*),
b. *Generate each individual randomly,*
c. *Calculate fitness* (*CPU*),
d. *Generate parent pairing*,
e. *Generate cutting point*,
g. *Exchange parent gene*,
h. *Mutation,*
i. *Partitioned roulette wheel,*

dilakukan pada tingkat pasangan / individu.

4. PENGUJIAN SISTEM

Pada bab ini dibahas tentang pengujian sistem terhadap program yang diimplementasikan. Pengujian dilakukan terhadap aspek kuantitatif berupa kecepatan, dan nilai *fitness*. Dimana fokus dari penelitian ini adalah perbandingan kecepatan dari program versi serial dan parallel. Ada tiga test case yang diuji, yaitu ca4663[12], fi10639[13], dan it16862[14]. Test case ca4663 merupakan titik titik sebanyak 4663 yang diambil dari World TSP pada negara Kanada. Test case fi10639 merupakan titik titik sebanyak 10639 yang diambil dari *World TSP* pada negara Filandia. Test case it16862 merupakan titik titik sebanyak 16862 yang diambil dari *World TSP* pada negara Italia. Tiap 10 menit, dilakukan pengambilan data untuk mengetahui progress dari program. Data yang diambil berupa nilai fitness paling tinggi, paling rendah, rata-rata populasi, dan jumlah generasi pada saat itu. Pada masing-masing platform diuji, program di hentikan setelah 12 jam. Hal tersebut dilakukan karena dengan waktu 12 jam tersebut masing masing kurva fitness sudah cukup terbentuk dan rata-rata generasi yang dihasilkan sudah cukup stabil. Pada *platform CPU* yang memiliki huruf S pada akhir namanya memiliki arti dia berjalan pada *single-core*. Sedangkan yang tidak memiliki huruf S memiliki arti berjalan pada *multi-core*.

Konfigurasi untuk genetic algorithm yang dipakai sebagai berikut:

- Mutation Rate : 5%,
- Elitism Rate : 5%,
- Jumlah individu : jumlah kota * 0.25,
- Ukuran turnamen : 3.

Parameter tersebut dipilih karena parameter tersebut merupakan parameter yang standar digunakan untuk *genetic algorithm* dan dapat berjalan pada semua *platform* yang diuji. Selain itu konfigurasi dari pemanggilan kernel sebagai berikut:

- Ukuran Block : jumlah maksimum *thread* per *SM* / 32,
- Ukuran Grid : 32 * jumlah *SM*.

Ada tiga komputer yang digunakan untuk pengujian, komputer pertama memiliki spesifikasi sebagai berikut:

- CPU : Ryzen 5 1600, 6 inti dengan *clock* 3.2 GHz,
- RAM : 16 GB *DDR4* dengan *clock* 3200 Mhz,
- GPU : RTX 2060, 30 *SM* dengan masing-masing *SM* memiliki 64 *CUDA core* dengan *clock* 1365 MHz,

Kemudian komputer kedua dengan spesifikasi berikut:

- CPU : Xeon E5 2630 v4, 10 inti dengan *clock* 2.2 GHz * 2,
- RAM : 128 GB *DDR4* dengan *clock* 2133 Mhz,
- GPU : Tesla P4, 20 *SM* dengan masing-masing *SM* memiliki 128 *CUDA core* dengan *clock* 1114 MHz.

Terakhir untuk komputer ketiga memiliki spesifikasi sebagai berikut:

- CPU : Ryzen Threadripper 2990 WX, 32 inti dengan *clock* 3 GHz,
- RAM : 64 GB *DDR4* dengan clock 2400 Mhz,
- GPU : RTX 2080 Ti, 68 SM dengan masing-masing *SM* memiliki 64 *CUDA core* dengan *clock* 1350 MHz, dan GTX 1070 Ti, 19 *SM* dengan masing-masing *SM* memiliki 128 *CUDA core* dengan *clock* 1607 MHz.

Untuk komputer ketiga hanya dilakukan pengujian pada *test case* it16862, karena pada *test case* tersebut performa pada kedua komputer yang lain mulai mencapai batas.

4.1. Pengujian pada Test Case ca4663

Pada *test case* ini, hasil paling baik diraih oleh platform CUDA dengan GPU RTX 2060 dengan nilai *fitness* 1.527.875,8. Selain itu GPU RTX 2060 juga memiliki rata-rata generasi yang dihasilkan dalam 10 menit sebanyak 6.781,3. Kemudian hasil paling buruk diraih oleh *plaform CPU* dengan *CPU* E5-2630v4 ketika berjalan dalam *mode single-core* dengan nilai *fitness* 4.372.899,0 dan jumlah rata-rata generasi yang dihasilkan dalam 10 menit sebanyak 63,7. Data yang didapatkan pada akhir terdapat pada Tabel 2 untuk rata-rata generasi dan Tabel 3 untuk data *fitness.*

Tabel 2. *Rata-Rata Generasi untuk Test Case ca4663*

Platform	Rata-rata generasi tiap menit
E5-2630v4S (CPU)	63,7
R5-1600S (CPU)	81,8
R5-1600 (CPU)	603,4
E5-2630v4 (CPU)	940
Tesla P4 (CUDA)	3.561,7
RTX 2060 (CUDA)	6.781,3

Tabel 3. *Data Fitness Test Case ca4663*

Plaform	Minimum	Rata-rata	Maksimum
E5-2630v4S	4.372.899,0	4.468.292,1	4.835.286,1
R5-1600S	3.673.959,5	3.756.221,1	4.117.880,8
R5-1600	2.111.284,3	2.268.885,6	2.732.384,0
E5-2630v4	1.967.163,1	2.127.867,7	2.510.704,3
Tesla P4	1.564.348,6	1.656.947,6	1.822.434,8
RTX 2060	1.527.875,8	1.685.845,1	1.892.503,0

4.2. Pengujian pada *Test Case fi10649*

Pada *test case* ini, hasil paling baik diraih oleh platform CUDA dengan GPU RTX 2060 dengan nilai *fitness* 888.951,5. Selain itu GPU RTX 2060 juga memiliki rata-rata generasi yang dihasilkan dalam 10 menit sebanyak 1.224,7. Kemudian hasil paling buruk diraih oleh *plaform CPU* dengan *CPU* E5-2630v4 ketika berjalan dalam *mode single-core* dengan nilai *fitness* 15.752.507,8 dan jumlah rata-rata generasi yang dihasilkan dalam 10 menit sebanyak 12.0. Data yang didapatkan pada akhir program terdapat pada Tabel 4 untuk rata-rata generasi dan Tabel 5 untuk data *fitness.*

Tabel 4. *Rata-Rata Generasi untuk Test Case fi10649*

Platform	Rata-rata generasi tiap menit
E5-2630v4S (CPU)	8,0
R5-1600S (CPU)	15,0
R5-1600 (CPU)	109,5
E5-2630v4 (CPU)	192
Tesla P4 (CUDA)	476,2
RTX 2060 (CUDA)	1.224,7

Tabel 5. *Data Fitness Test Case fi10649*

Plaform	Minimum	Rata-rata	Maksimum
E5-2630v4S	17.449.339,1	17.513.051,3	17.599.316,3
R5-1600S	14.154.926,6	14.194.222,1	14.252.572,3
R5-1600	3.030.184,6	3.052.983,9	3.116.257.0
E5-2630v4	1.910.552,7	1.931.680,6	1.995.457,2
Tesla P4	1.096.557,4	1.116.583,2	1.145.332,1
RTX 2060	888.951,5	903.758,6	927.015,0

4.3. Pengujian pada *Test Case it16862*

Pada *test case* ini, hasil paling baik diraih oleh platform *CUDA* dengan *GPU* RTX 2080 Ti dengan nilai *fitness* 1.208.298. Selain itu *GPU* RTX 2080 Ti juga memiliki rata-rata generasi yang dihasilkan tiap menit sebanyak 879,0. Kemudian hasil paling buruk diraih oleh *plaform CPU* dengan *CPU* E5-2630v4 ketika berjalan dalam *mode single-core* dengan nilai *fitness* 35.162.753,9 dan jumlah rata-rata generasi yang dihasilkan dalam 10 menit sebanyak 4,9. Pada *test case* ini, walaupun pada *platform* TR-2990WX dan Tesla P4 jumlah generasi cukup dekat, tetapi karena perbedaan *random engine* yang dipakai pada kedua *platform*, pada Tesla P4 dapat melampaui nilai *fitness* dari TR-2990WX. Data yang didapatkan pada akhir program terdapat pada Tabel 6 untuk rata-rata generasi dan Tabel 7 untuk data *fitness.*

Tabel 6. *Rata-Rata Generasi untuk Test Case it16862*

Platform	Rata-rata generasi tiap menit
E5-2630v4S (CPU)	4,9
TR-2990WXS (CPU)	5,1
R5-1600S (CPU)	5,7
R5-1600 (CPU)	41,0
E5-2630v4 (CPU)	78
Tesla P4 (CUDA)	111,8
TR-2990WX (CPU)	113,1
GTX 1070 Ti (CUDA)	217,7
RTX 2060 (CUDA)	336,3
GTX 2080 Ti (CUDA)	879,0

Tabel 7. ***Data Fitness Test Case it16862***

Platform	Minimum	Rata-rata	Maksimum
E5-2630v4S	35.162.753,9	35.291.258,7	35.463.585,1
TR-2990WXS	34.423.275,3	34.549.554,5	34.706.707,7
R5-1600S	33.315.569,9	33.451.954,6	33.622.007,5
R5-1600	13.966.778,9	14.006.352,0	14.091.529,7
E5-2630v4	7.750.217,7	7.774.997,7	7.846.569,6
TR-2990WX	5.382.960,2	5.314.011,7	5.382.960,2
Tesla P4	4.178.787,6	4.199.267,2	4.228.963,7
GTX 1070Ti	2.501.517,8	2.518.557,8	2.545.618,7
RTX 2060	1.900.081,5	1.921.036,1	1.949.108,7
GTX 2080 i	1.185.113,7	1.208.297,9	1.238.566,8

5. KESIMPULAN DAN SARAN

5.1. Kesimpulan

Dengan melakukan penelitian ini, dapat disimpulkan bahwa paralelisasi *Genetic Algorithm* memiliki hasil yang cukup memuaskan dalam beberapa hal, yaitu sebagai berikut:

- Peningkatan jumlah generasi yang dihasilkan tiap 10 menit dapat meningkat sebanyak 59,3 kali hingga 106,5 kali menggunakan GPU RTX 2060 dan 19,7 hingga 55,9 kali menggunakan Tesla P4.
- Peningkatan jumlah generasi yang dihasilkan tiap 10 menit dapat meningkat sebanyak 7,2 kali hingga 7,4 kali dengan memanfaatkan semua *thread* yang dimiliki R5 1600 dan 19,7 hingga 55,9 kali kali dengan memanfaatkan semua *thread* yang dimiliki *E5-2630 v4*.
- Dengan memanfaatkan pembuatan *trendline* pada *Microsoft Excel* bisa didapatkan rumus *trendline* generasi yang dihasilkan $y = 2051{,}7e^{-1{,}344x}$, untuk *E5-2630 v4*, $y = 2896{,}9e^{-1{,}242x}$ untuk *R5 1600*, $y = 18307e^{-1{,}731x}$ untuk Tesla P4, $y = 28396e^{-1{,}502x}$ untuk RTX 2060,
- Berdasarkan *Amdahl's Law*, proporsi waktu eksekusi yang dapat diparalelisasi paling sedikit 0,92. Sehingga, metode *Genetic Algorithm* untuk *Travelling Salesman Problem* dapat diparalelisasi dengan efektif.

5.2. Saran

Peneliti mengharapkan bahwa pada penelitian berikutnya, parameter dari *Genetic Algorithm* ini dapat di *tuning* agar mendapatkan hasil yang lebih baik. Selain itu juga dapat menyelesaikan kasus *TSP* yang jauh lebih besar.

REFERENSI

[1] Barney, B. URI=https://computing.llnl.gov/tutorials/parallel_comp/

[2] Greco, F. 2008. *Traveling salesman problem*. SL: Sciyo.com.

[3] Horton, I. 2015. Using the C standard template libraries. New York: Apress.

[4] Intel. 2015, January 01. Threading Models for High-Performance Computing: Pthreads or OpenMP? URI=https://software.intel.com/en-us/articles/threading-models-for-high-performance-computing-pthreads-or-openmpNvidia.

[5] Kramer, O. 2017. Genetic algorithm essentials. Cham: Springer.

[6] McCool, M. D., Robison, A. D., & Reinders, J. 2012. *Structured parallel programming: patterns for efficient computation*. Waltham: Morgan Kaufmann.

[7] Nvidia. CUDA Zone. URI=https://developer.nvidia.com/cuda-zone

[8] Nvidia. 2019, November. Thrust Quick Star Guide. URI= https://docs.nvidia.com/pdf/Thrust_Quick_Start_Guide.pdf

[9] Oiso, M., Matsumura, Y., Yasuda, T., & Ohkura, K. 2011. Implementation Method of Genetic Algorithms to the CUDA Environment using Data Parallelization. Journal of Japan Society for Fuzzy Theory and Intelligent Informatics, 23(1), 18-28. doi:10.3156/jsoft.23.18

[10] OpenMP ARB. OpenMP FAQ. URI=https://www.openmp.org/about/openmp-faq/

[11] Werkhoven, B. V., Maassen, J., Seinstra, F., & Bal, H. 2014. Performance Models for CPU-GPU Data Transfers. 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. doi:10.1109/ccgrid.2014.16

[12] University of Waterloo. 2001. CA4663 [Data file]. Retrieved from University of Waterloo. URI=http://www.math.uwaterloo.ca/tsp/world/ca4663.tsp

[13] University of Waterloo. 2001. FI10639 [Data file]. Retrieved from University of Waterloo. URI=http://www.math.uwaterloo.ca/tsp/world/fi10639.tsp

[14] University of Waterloo. 2001. IT16862 [Data file]. Retrieved from University of Waterloo. URI=http://www.math.uwaterloo.ca/tsp/world/it16862.tsp