PROPOSAL PENELITIAN

EVALUASI PERFORMA CONCURRENCY DAN PARALLELISM PADA BAHASA PEMROGRAMAN MODERN: STUDI KASUS GO, RUST, JAVA, DAN PYTHON

Disusun Oleh:

Isma'il Faruqy 2410511092

PROGRAM STUDI INFORMATIKA PROGRAM SARJANA

FAKULTAS ILMU KOMPUTER

UNIVERSITAS PEMBANGUNAN NASIONAL VETERAN JAKARTA

2025

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

BAB I
PENDAHULUAN

1.1 Latar Belakang

Perkembangan teknologi komputasi modern ditandai dengan meningkatnya kebutuhan akan sistem yang mampu memanfaatkan arsitektur multi-core processor secara optimal. Dalam konteks ini, konsep concurrency (eksekusi tugas secara bersamaan) dan parallelism (eksekusi simultan menggunakan beberapa inti prosesor) menjadi sangat penting untuk meningkatkan performa aplikasi. Keduanya telah menjadi fondasi pengembangan perangkat lunak berskala besar, mulai dari web server berperforma tinggi, big data processing systems, hingga komputasi ilmiah di bidang high-performance computing (HPC). Optimalisasi concurrency dan parallelism memungkinkan perangkat lunak untuk mengurangi waktu eksekusi, meningkatkan throughput, serta memanfaatkan sumber daya perangkat keras secara efisien (Lee & Aggarwal, 1987).

Berbagai bahasa pemrograman modern menawarkan pendekatan berbeda dalam mendukung concurrency dan parallelism. Go, yang dikembangkan Google, dikenal dengan goroutines dan channels yang ringan serta terintegrasi dengan runtime scheduler, sehingga memudahkan pengembang menulis kode konkuren dengan overhead minimal (Castro et al., 2019). Rust, sebagai bahasa sistem modern, mengusung ownership model yang menjamin keamanan memori sekaligus menghindari data race, serta mendukung asynchronous programming melalui pustaka seperti Tokio (Klabnik & Nichols, 2018). Java, yang telah lama menjadi standar industri, memiliki ekosistem matang melalui ExecutorService, Fork/Join Framework, dan Parallel Streams, dengan dukungan just-in-time compilation (JIT)

serta garbage collector yang terus berevolusi (Lea, 2000). Sementara itu, Python, meskipun populer untuk komputasi ilmiah, memiliki keterbatasan mendasar berupa Global Interpreter Lock (GIL) yang membatasi eksekusi paralel berbasis threads pada CPU-bound tasks. Upaya mitigasi biasanya dilakukan dengan modul multiprocessing atau pustaka eksternal seperti NumPy yang memanfaatkan BLAS (Van der Walt & Aivazis, 2011). Perkembangan terbaru melalui PEP 703 bahkan membuka arah baru dengan opsi build Python tanpa GIL, yang berpotensi meningkatkan performa paralel di masa depan (Gross, 2023).

Sejumlah penelitian terdahulu telah membandingkan performa concurrency dan parallelism pada beberapa bahasa pemrograman. Penelitian oleh Abhinav et al., (2020) mengevaluasi Go dan Java menggunakan matrix multiplication serta PageRank, dan menemukan bahwa Go unggul dalam efisiensi goroutine dan kecepatan kompilasi, sementara Java lebih baik dalam eksekusi matriks besar berkat optimasi ExecutorService. Sementara itu, Costanza et al., (2019) membandingkan Go, Java, dan C++17 pada pipeline bioinformatika (elPrep) dan menemukan bahwa Go memberikan keseimbangan terbaik antara runtime dan konsumsi memori, Java lebih cepat namun boros memori, sedangkan C++17 mengalami overhead signifikan karena reference counting.

Selain itu, penelitian tentang matrix-matrix multiplication menunjukkan bahwa kernel komputasi ini merupakan beban kerja representatif untuk mengukur kemampuan parallelism. Michailidis dan Margaritis (2012) menekankan bahwa optimasi algoritmik seperti blocking dan loop interchange sangat berpengaruh terhadap performa pada multi-core. Temuan lebih mutakhir oleh Akoushideh dan

Shahbahrami (2022) menunjukkan bahwa kombinasi SIMD, OpenMP, dan OpenCL dapat meningkatkan performa hingga 32 kali lipat dibanding implementasi serial pada CPU. Namun, mayoritas studi tersebut lebih berfokus pada model pemrograman paralel (misalnya OpenMP atau OpenCL) daripada membandingkan langsung bahasa pemrograman tingkat tinggi.

Dari literatur tersebut dapat dilihat bahwa studi yang ada masih terbatas pada kasus tertentu, baik berupa algoritma spesifik atau aplikasi domain tertentu, dan umumnya hanya membandingkan sebagian kecil bahasa pemrograman. Belum banyak kajian yang melakukan evaluasi menyeluruh terhadap performa concurrency dan parallelism pada beberapa bahasa modern sekaligus dengan metodologi benchmarking yang terstandar. Mengingat perbedaan mendasar dalam model eksekusi, runtime, serta strategi pengelolaan memori antar bahasa, diperlukan analisis komprehensif untuk memahami kelebihan dan keterbatasan masing-masing. Evaluasi lintas bahasa dengan mengukur metrik seperti waktu eksekusi, pemanfaatan CPU, konsumsi memori, serta skalabilitas pada berbagai jumlah inti prosesor akan memberikan gambaran lebih objektif mengenai performa concurrency dan parallelism.

Oleh karena itu, penelitian ini dilakukan untuk mengevaluasi dan membandingkan performa concurrency dan parallelism pada empat bahasa pemrograman modern, yaitu Go, Rust, Java, dan Python. Studi ini menggunakan benchmark tasks representatif seperti matrix multiplication dan parallel sorting untuk menguji skenario komputasi intensif, serta pola producer-consumer untuk mengevaluasi concurrency pada beban kerja ringan. Dengan mengukur metrik

kunci pada lingkungan multi-core processor, penelitian ini diharapkan dapat memberikan kontribusi ilmiah berupa pemahaman yang lebih mendalam mengenai performa concurrency dan parallelism pada bahasa pemrograman modern, sekaligus memberikan wawasan praktis bagi pengembang, peneliti, maupun industri dalam memilih bahasa yang sesuai untuk aplikasi berbasis komputasi paralel.

1.2 Rumusan Masalah

Berdasarkan latar belakang yang telah dipaparkan, maka permasalahan penelitian ini dapat dirumuskan sebagai berikut:

1. Bagaimana perbedaan performa concurrency dan parallelism pada bahasa pemrograman Go, Rust, Java, dan Python ketika diuji menggunakan benchmark tasks representatif seperti matrix multiplication, parallel sorting, dan pola producer–consumer?
2. Sejauh mana masing-masing bahasa pemrograman mampu memanfaatkan prosesor multi-inti dalam hal skalabilitas, waktu eksekusi, penggunaan CPU, dan konsumsi memori?
3. Faktor apa saja yang mempengaruhi kelebihan dan keterbatasan setiap bahasa dalam mendukung concurrency dan parallelism, ditinjau dari aspek model eksekusi, manajemen memori, serta mekanisme runtime?
4. Bahasa pemrograman mana yang menunjukkan efisiensi terbaik untuk skenario komputasi intensif dan beban kerja konkuren, serta bagaimana rekomendasi pemanfaatannya pada konteks pengembangan perangkat lunak modern?

1.3 Batasan Masalah

Agar penelitian ini lebih terarah dan fokus, maka ruang lingkup penelitian dibatasi sebagai berikut:

1.3.1 Bahasa Pemrograman yang Diteliti

Penelitian hanya mencakup empat bahasa pemrograman modern, yaitu Go, Rust, Java, dan Python. Bahasa lain seperti C/C++, C#, atau Erlang tidak termasuk dalam lingkup penelitian ini.

1.3.2 Jenis Benchmark

Benchmark yang digunakan terbatas pada tugas komputasi representatif, yaitu matrix multiplication, parallel sorting, serta pola producer–consumer. Benchmark lain seperti graf komputasi, simulasi numerik kompleks, atau GPU-based workloads tidak dibahas.

1.3.3 Lingkungan Pengujian

Eksperimen dilakukan pada lingkungan multi-core CPU dengan konfigurasi perangkat keras dan perangkat lunak yang dikontrol (misalnya versi kompiler, interpreter, runtime, serta sistem operasi). Pengujian pada GPU, cluster, atau arsitektur heterogen tidak termasuk dalam penelitian ini.

1.3.4 Implementasi Program

Implementasi benchmark di setiap bahasa menggunakan pustaka standar atau mekanisme bawaan bahasa. Optimalisasi tingkat lanjut dengan pustaka eksternal pihak ketiga (misalnya NumPy untuk Python atau BLAS untuk Java/Rust) hanya dipertimbangkan secara terbatas untuk perbandingan, tetapi tidak menjadi fokus utama.

1.3.5 Metrik yang Dianalisis

Metrik performa yang dianalisis meliputi waktu eksekusi, pemanfaatan CPU, konsumsi memori, serta skalabilitas terhadap jumlah inti prosesor. Metrik lain seperti energy consumption, ukuran kode, atau kompleksitas pengembangan tidak menjadi fokus utama penelitian ini.

1.4 Tujuan Penelitian

Penelitian ini bertujuan untuk mengevaluasi dan membandingkan performa concurrency dan parallelism pada beberapa bahasa pemrograman modern, yaitu Go, Rust, Java, dan Python, dengan menggunakan benchmark representatif pada lingkungan prosesor multi-inti.

Secara lebih rinci, penelitian ini memiliki tujuan khusus sebagai berikut:

1. Mengukur dan menganalisis perbedaan waktu eksekusi, pemanfaatan CPU, konsumsi memori, serta skalabilitas dari implementasi concurrency dan parallelism pada Go, Rust, Java, dan Python.
2. Mengevaluasi efisiensi setiap bahasa pemrograman dalam menyelesaikan benchmark tasks komputasi intensif seperti matrix multiplication dan parallel sorting, serta beban kerja konkuren seperti pola producer–consumer.
3. Mengidentifikasi faktor-faktor teknis yang memengaruhi kinerja setiap bahasa, termasuk model eksekusi, mekanisme runtime, dan strategi manajemen memori.

4. Memberikan rekomendasi pemilihan bahasa pemrograman yang sesuai untuk kebutuhan aplikasi yang menuntut concurrency dan parallelism tinggi, baik dalam konteks akademis maupun industri.

1.5 Manfaat Penelitian

1.5.1 Manfaat Teoritis

Penelitian ini diharapkan dapat menambah khazanah literatur akademis di bidang ilmu komputer, khususnya pada topik concurrency dan parallelism dalam bahasa pemrograman modern. Hasil penelitian dapat menjadi rujukan bagi peneliti lain yang ingin mengembangkan studi sejenis, serta memperkaya pemahaman mengenai perbedaan model eksekusi, strategi runtime, dan pengelolaan memori antar bahasa.

1.5.2 Manfaat Praktis

Hasil penelitian dapat memberikan panduan bagi pengembang perangkat lunak dalam memilih bahasa pemrograman yang sesuai dengan kebutuhan aplikasi, khususnya aplikasi yang menuntut concurrency tinggi maupun pemrosesan paralel berskala besar. Dengan demikian, penelitian ini dapat membantu meningkatkan efisiensi pengembangan dan performa aplikasi nyata.

1.6 Sistematika Penulisan

1. BAB I PENDAHULUAN

Bab ini menguraikan latar belakang penelitian yang menjelaskan urgensi evaluasi performa concurrency dan parallelism pada bahasa pemrograman

modern. Selanjutnya dipaparkan rumusan masalah, batasan masalah, tujuan penelitian, manfaat penelitian baik secara teoritis maupun praktis, serta sistematika penulisan yang menggambarkan struktur keseluruhan penelitian.

2. BAB II TINJAUAN PUSTAKA

Bab ini menyajikan landasan teori yang mencakup konsep dasar concurrency dan parallelism, model pemrograman paralel dan konkuren (thread-based, message-passing, asynchronous, data-parallel, dan hybrid model), hukum Amdahl dan Gustafson sebagai dasar teoritis pengukuran performa sistem paralel, karakteristik empat bahasa pemrograman yang diteliti (Go, Rust, Java, dan Python), serta benchmark komputasi paralel yang digunakan dalam penelitian. Bab ini juga mengulas penelitian-penelitian terdahulu yang relevan sebagai dasar komparasi dan pengembangan metodologi penelitian.

3. BAB III METODE PENELITIAN

Bab ini menjelaskan jenis dan pendekatan penelitian yang digunakan, yaitu penelitian eksperimental dengan pendekatan kuantitatif. Kemudian dipaparkan kerangka kerja penelitian yang terdiri dari lima tahapan utama: studi literatur, perancangan eksperimen, implementasi benchmark, pelaksanaan pengujian, dan analisis data. Bab ini juga menjelaskan tempat penelitian beserta spesifikasi perangkat keras dan perangkat lunak yang digunakan, serta teknik analisis data yang meliputi analisis kuantitatif deskriptif, analisis perbandingan (speedup dan efficiency), analisis skalabilitas berdasarkan hukum Amdahl dan Gustafson, serta visualisasi dan interpretasi hasil.

4. BAB IV HASIL DAN PEMBAHASAN (untuk laporan akhir)

Bab ini akan menyajikan data hasil pengujian performa concurrency dan parallelism pada empat bahasa pemrograman yang diteliti. Hasil eksperimen akan disajikan dalam bentuk tabel, grafik, dan visualisasi lainnya. Pembahasan akan menganalisis perbandingan performa antar bahasa, skalabilitas sistem terhadap penambahan jumlah thread, serta interpretasi hasil berdasarkan teori yang telah dipaparkan pada Bab II.

5. BAB V KESIMPULAN DAN SARAN (untuk laporan akhir)

Bab ini akan menyimpulkan hasil penelitian secara menyeluruh dengan menjawab rumusan masalah yang telah ditetapkan. Kesimpulan akan mencakup perbandingan performa keempat bahasa pemrograman, identifikasi faktor-faktor yang memengaruhi kinerja masing-masing bahasa, serta rekomendasi pemilihan bahasa pemrograman untuk berbagai skenario aplikasi. Selain itu, bab ini juga akan menyajikan saran untuk pengembangan penelitian selanjutnya, termasuk kemungkinan perluasan cakupan benchmark, pengujian pada arsitektur perangkat keras berbeda, atau eksplorasi teknik optimasi lanjutan.

6. DAFTAR PUSTAKA

Bagian ini memuat seluruh referensi yang digunakan dalam penelitian, meliputi buku, artikel jurnal ilmiah, prosiding konferensi, dan sumber online yang relevan. Penulisan daftar pustaka mengikuti standar sitasi yang berlaku.

7. LAMPIRAN (untuk laporan akhir)

Lampiran akan memuat informasi pendukung seperti source code implementasi benchmark untuk keempat bahasa pemrograman, hasil log pengujian lengkap, tabel data mentah hasil eksperimen, serta dokumentasi teknis lainnya yang diperlukan untuk verifikasi dan replikasi penelitian.

BAB II
TINJAUAN PUSTAKA

2.1 Landasan Teori

2.1.1 Konsep Dasar *Concurrency* dan *Parallelism*

2.1.1.1 *Concurrency*

Concurrency merupakan kemampuan sistem untuk menjalankan beberapa proses secara tumpang tindih dalam satu rentang waktu, dengan fokus utama pada koordinasi eksekusi dan manajemen sumber daya. Roscoe (2005) mendefinisikan concurrency sebagai interaksi antara sejumlah proses yang berjalan bersamaan melalui komunikasi sinkron, di mana keteraturan dan keamanan komunikasi menjadi lebih penting dibanding kecepatan eksekusi. Pendekatan *Communicating Sequential Processes (CSP)* yang ia kembangkan menyoroti potensi *nondeterminism*, *deadlock*, dan *livelock* sebagai konsekuensi alami dari sistem konkuren.

Dalam konteks modern, Yuan dan Yang (2020) menggambarkan concurrency sebagai eksekusi asinkron dari banyak operasi yang berinteraksi melalui *message passing* dalam sistem terdistribusi. Mereka menunjukkan bahwa tantangan utama concurrency bukan hanya pada performa, tetapi pada kebenaran dan keandalan interaksi antar proses yang dapat berjalan dalam berbagai urutan (*interleaving*).

Abhinav et al. (2020) menegaskan concurrency sebagai kemampuan program untuk menjalankan beberapa tugas secara

interleaved, bukan simultan, untuk memaksimalkan efisiensi penggunaan prosesor. Castro et al. (2019) memperluas hal ini dengan melihat concurrency sebagai *structured communication* antar proses, mendorong keamanan dan keteraturan interaksi melalui *session types*. Costanza et al. (2019) kemudian membedakan concurrency dari parallelism: concurrency berurusan dengan pengaturan banyak tugas secara bersamaan, sedangkan parallelism fokus pada eksekusi simultan untuk meningkatkan kecepatan.

Dengan demikian, concurrency dapat dipahami sebagai fondasi logis dari sistem paralel dan terdistribusi, yang menekankan koordinasi, komunikasi, dan kontrol antar proses agar sistem tetap efisien, aman, dan konsisten.

2.1.1.2 *Parallelism*

Parallelism merupakan konsep eksekusi simultan dari beberapa proses atau instruksi pada waktu yang sama untuk meningkatkan kinerja komputasi. Berbeda dari concurrency yang menitikberatkan pada koordinasi tugas, parallelism berfokus pada pembagian beban kerja ke beberapa unit pemrosesan secara bersamaan. Michailidis dan Margaritis (2012) menunjukkan bahwa parallelism menjadi dasar utama dalam pemanfaatan prosesor multi-core, terutama untuk *matrix computation* dan operasi numerik intensif. Melalui pendekatan seperti OpenMP, Intel TBB, dan

Cilk++, mereka membuktikan bahwa pemrosesan paralel secara langsung meningkatkan throughput sistem dan efisiensi eksekusi.

Pendekatan ini diperkuat oleh Akoushideh dan Shahbahrami (2022) yang mengevaluasi berbagai model pemrograman paralel pada CPU, seperti SIMD, OpenMP, Hybrid OpenMP-SIMD, dan OpenCL. Hasil eksperimen mereka menunjukkan peningkatan performa hingga 32 kali lebih cepat dibandingkan implementasi sekuensial, membuktikan bahwa parallelism efektif untuk beban kerja yang dapat dipartisi secara independen. Penelitian ini juga menegaskan perbedaan mendasar antara *data-level parallelism* (misalnya SIMD) dan *thread-level parallelism* (misalnya OpenMP), yang keduanya berperan penting dalam memaksimalkan potensi multi-core modern.

Secara teoritis, batas percepatan parallelism dijelaskan oleh Amdahl's Law (1967), yang menyatakan bahwa peningkatan jumlah prosesor tidak akan menghasilkan *speedup* linear jika sebagian kode tetap berjalan secara sekuensial. Sebaliknya, Gustafson's Law (1988) menegaskan bahwa efisiensi parallelism dapat terus meningkat seiring bertambahnya ukuran workload, karena bagian paralel dari program mendominasi total eksekusi. Kedua teori ini menjelaskan bahwa parallelism tidak hanya bergantung pada jumlah inti pemrosesan, tetapi juga pada proporsi kode yang dapat dijalankan secara simultan.

Dengan demikian, parallelism merupakan aspek fisik dari eksekusi sistem yang bertujuan meningkatkan kecepatan komputasi melalui pemrosesan simultan, sedangkan concurrency berperan sebagai fondasi logis yang mengatur bagaimana tugas-tugas tersebut dapat berjalan berdampingan secara terkoordinasi dan efisien.

2.1.2 Model Pemograman Paralel dan Konkuren

Seiring perkembangan perangkat keras, berbagai model pemrograman paralel dan konkuren telah dikembangkan untuk mengelola kompleksitas eksekusi. Pendekatan modern sering kali berfokus pada penggunaan pola-pola desain yang terstruktur (*structured patterns*) untuk menciptakan program yang efisien dan mudah dikelola, yang dikenal sebagai *algorithmic skeletons* (McCool et al., 2012).

2.1.2.1 Thread-Based Model

Model ini merupakan pendekatan paling umum dalam sistem berbasis *shared memory*, di mana beberapa thread dijalankan secara paralel dalam satu proses dan berbagi ruang memori yang sama. Java dan C++ menggunakan pendekatan ini secara luas. Dalam Java, framework seperti ExecutorService dan Fork/Join Framework memungkinkan manajemen thread yang efisien dengan *work-stealing* untuk memaksimalkan pemanfaatan core (Lea, 2000). Pendekatan ini memudahkan pembagian tugas secara paralel, namun menimbulkan potensi *race condition* dan *deadlock* apabila sinkronisasi tidak diatur dengan baik.

Michailidis dan Margaritis (2012) menunjukkan bahwa performa *thread-based parallelism* sangat bergantung pada strategi pembagian beban dan efisiensi sinkronisasi antar thread. Meskipun efektif untuk komputasi intensif, model ini kurang cocok untuk sistem terdistribusi karena overhead komunikasi meningkat secara signifikan.

2.1.2.2 Message-Passing Model

Berbeda dengan model berbagi memori, *message-passing model* beroperasi berdasarkan komunikasi eksplisit antar proses atau aktor melalui pesan. Go dan Erlang merupakan contoh bahasa yang mengadopsi pendekatan ini secara langsung. Go menggunakan goroutines sebagai unit konkuren ringan yang berkomunikasi lewat channels, sesuai prinsip *communicating sequential processes (CSP)* yang diperkenalkan oleh Roscoe (2005).

Castro et al. (2019) mengembangkan pendekatan ini lebih jauh melalui *role-parametric session types*, yang menjamin keamanan dan keteraturan komunikasi antar proses. Keunggulan utama model ini adalah skalabilitas dan keamanan terhadap *race condition*, karena setiap proses memiliki ruang memori terisolasi. Namun, biaya komunikasi antar entitas bisa meningkat pada sistem dengan jumlah proses besar.

2.1.2.3 Asynchronous/Reactive Model

Model ini mengandalkan mekanisme eksekusi non-blok (*non-blocking execution*) di mana tugas tidak perlu menunggu operasi I/O selesai sebelum melanjutkan ke tugas berikutnya. Pendekatan ini umum pada bahasa modern seperti Rust dan Python. Rust menggunakan *async/await* serta pustaka seperti Tokio untuk mendukung *asynchronous task scheduling* yang aman dari *data race* (Klabnik & Nichols, 2018).

Python, melalui pustaka asyncio, menerapkan *event loop* untuk menangani banyak tugas I/O-bound secara konkuren, meskipun eksekusi paralelnya dibatasi oleh Global Interpreter Lock (GIL). Gross (2023) melalui PEP 703 mengusulkan penghapusan GIL sebagai upaya membuka dukungan penuh terhadap *true parallelism*. Model ini unggul dalam efisiensi I/O-bound workloads, namun memiliki keterbatasan dalam komputasi CPU-bound.

2.1.2.4 Data-Parallel Model

Model ini menekankan pada pembagian data ke beberapa unit pemrosesan untuk dieksekusi secara bersamaan. Pendekatan ini sering digunakan dalam aplikasi ilmiah dan *high-performance computing* (HPC). (Michailidis & Margaritis, 2012) serta Akoushideh dan Shahbahrami (2022) menunjukkan bahwa model seperti SIMD, OpenMP, dan OpenCL mampu meningkatkan

performa signifikan pada operasi *matrix-matrix multiplication*—dengan *speedup* hingga 32 kali lipat pada CPU multi-core.

Dalam model ini, operasi identik dilakukan pada elemen data berbeda secara serentak, sehingga cocok untuk workload numerik besar. Kelemahannya terletak pada fleksibilitas yang rendah terhadap tugas yang tidak homogen, karena efisiensinya bergantung pada keseimbangan pembagian data antar core.

2.1.2.5 Hybrid Model

Seiring perkembangan perangkat keras, banyak sistem modern menerapkan hybrid parallelism, yaitu kombinasi antara thread-based dan data-parallel model. Contohnya adalah integrasi OpenMP dengan SIMD (OpenMP-SIMD) yang diuji oleh Akoushideh dan Shahbahrami (2022), di mana penggabungan *thread-level parallelism* dan *data-level parallelism* mampu menghasilkan performa optimal. Pendekatan hibrida juga umum dalam arsitektur multi-core dan GPU modern, di mana concurrency digunakan untuk koordinasi antar thread, sedangkan parallelism digunakan untuk mempercepat eksekusi di tingkat data.

2.1.3 Hukum Amdahl dan Gustafson

Kinerja sistem paralel secara teoritis dibatasi oleh proporsi kode yang dapat dijalankan secara bersamaan. Dua hukum paling berpengaruh yang menjelaskan batas dan potensi tersebut adalah Hukum Amdahl (Amdahl’s Law) dan Hukum Gustafson (Gustafson’s Law).

2.1.3.1 Hukum Amdahl

Gene Amdahl (1967) mengemukakan bahwa percepatan (*speedup*) maksimum dari sistem paralel dibatasi oleh bagian program yang tetap berjalan secara sekuensial. Jika proporsi bagian sekuensial dari program dilambangkan dengan (f), dan jumlah prosesor yang digunakan adalah (p), maka percepatan teoretis maksimum dapat dihitung dengan persamaan:

$$S(p) = \frac{1}{f + \frac{(1-f)}{p}}$$

Dari persamaan tersebut, dapat dilihat bahwa meskipun jumlah prosesor (p) ditingkatkan secara tak terbatas, *speedup* maksimum tetap dibatasi oleh bagian sekuensial (f). Artinya, tidak semua bagian program dapat diparalelisasi sepenuhnya, sehingga efisiensi sistem akan menurun seiring bertambahnya jumlah prosesor. Hukum ini menekankan pentingnya meminimalkan bagian sekuensial dalam algoritma agar peningkatan jumlah core memberikan manfaat yang signifikan.

Sebagai contoh, Michailidis dan Margaritis (2012) menunjukkan bahwa meskipun teknik seperti *loop interchange* dan *blocking* dapat meningkatkan performa matrix multiplication, bottleneck tetap terjadi pada tahap sinkronisasi dan memori, sesuai dengan prediksi Amdahl's Law.

2.1.3.2 Hukum Gustafson

John Gustafson (1988) mengusulkan reinterpretasi terhadap Amdahl's Law dengan asumsi bahwa ukuran masalah (*problem size*) dapat diperbesar seiring bertambahnya jumlah prosesor. Hukum ini menekankan bahwa efisiensi parallelism seharusnya tidak diukur dengan waktu tetap, tetapi dengan skala pekerjaan yang meningkat. Persamaan yang diajukan Gustafson adalah:

$$S(p) = p - f \times (p - 1)$$

Di mana (f) kembali melambangkan proporsi waktu eksekusi sekuensial. Berbeda dari Amdahl, Gustafson berpendapat bahwa dengan memperbesar ukuran workload, bagian paralel program akan semakin dominan, sehingga efisiensi sistem bisa tetap tinggi meskipun jumlah prosesor bertambah.

Pendekatan ini lebih realistis untuk aplikasi *high-performance computing (HPC)* modern yang terus memperbesar ukuran data atau kompleksitas perhitungan agar dapat memanfaatkan semua core yang tersedia. Studi oleh Akoushideh dan Shahbahrami (2022) memperkuat hal ini, di mana peningkatan ukuran matriks justru meningkatkan *speedup ratio* secara signifikan karena bagian paralel mendominasi keseluruhan komputasi.

2.1.3.3 Implikasi terhadap Penelitian

Kedua hukum ini memberikan dasar teoritis dalam mengevaluasi efisiensi sistem konkuren dan paralel. Hukum Amdahl

menekankan batasan fundamental dari parallelism, sedangkan Hukum Gustafson menyoroti potensi skalabilitas dengan peningkatan workload. Dalam konteks penelitian ini, kedua hukum digunakan sebagai acuan analisis hasil pengujian performa pada bahasa Go, Rust, Java, dan Python, untuk menilai sejauh mana masing-masing bahasa mampu menskalakan kinerja seiring bertambahnya jumlah inti prosesor dan kompleksitas tugas.

2.1.4 Bahasa Pemograman

Setiap bahasa pemrograman modern memiliki pendekatan dan filosofi tersendiri dalam mendukung *concurrency* dan *parallelism*. Perbedaan tersebut mencakup model eksekusi, mekanisme sinkronisasi, serta strategi pengelolaan memori dan runtime. Empat bahasa yang menjadi fokus penelitian ini Go, Rust, Java, dan Python mewakili paradigma dan ekosistem yang beragam dalam desain sistem konkuren dan paralel.

2.1.4.1 Go

Go, dikembangkan oleh Google pada tahun 2007, dirancang dengan fokus pada *scalable concurrent programming* (Nurwina Quirante et al., 2023). Bahasa ini mengimplementasikan model Communicating Sequential Processes (CSP) (Roscoe, 2005), yang mengizinkan banyak *goroutines* (unit konkuren ringan) berinteraksi melalui channels. *Goroutines* dijadwalkan oleh runtime Go menggunakan *work-stealing scheduler* yang efisien, memungkinkan

ribuan proses berjalan serentak dengan overhead minimal (Castro et al., 2019).

Abhinav et al. (2020) menunjukkan bahwa Go unggul dalam manajemen *lightweight concurrency*, terutama untuk beban kerja dengan banyak tugas kecil yang berjalan bersamaan. Go juga memiliki mekanisme sinkronisasi yang aman terhadap *race condition* melalui *channel communication* dan *select statements*. Namun, keterbatasannya muncul pada *pure parallel computation*, karena eksekusi paralel Go tetap bergantung pada runtime dan *garbage collector* yang kadang menambah latensi pada *CPU-bound tasks*.

2.1.4.2 Rust

Rust adalah bahasa pemrograman sistem yang dikembangkan Tim Rust yang disponsori oleh Mozilla dengan tujuan menggabungkan performa tingkat rendah seperti C/C++ dengan keamanan memori yang kuat (Jung et al., 2021; Klabnik & Nichols, 2018). Keunggulan utama Rust dalam konteks concurrency terletak pada ownership model dan sistem tipe statis yang mencegah *data race* secara kompilasi.

Rust mendukung dua paradigma utama: *thread-based concurrency* dan *asynchronous programming*. Melalui pustaka seperti Tokio dan async/await, Rust memungkinkan penjadwalan asinkron tanpa overhead besar (Klabnik & Nichols, 2018).

Pendekatan ini membuat Rust efisien untuk sistem *I/O-bound* maupun *compute-bound*, dengan performa mendekati C++. Rust tidak menggunakan *garbage collector*, sehingga manajemen memori bersifat deterministik, menjadikannya unggul untuk aplikasi *high-performance* dan *real-time systems*.

Namun, kompleksitas sintaks dan aturan kepemilikan (borrowing rules) dapat memperlambat pengembangan bagi pemrogram baru. Walaupun demikian, Rust terus berkembang sebagai kandidat kuat untuk aplikasi HPC modern karena kemampuannya menggabungkan *zero-cost abstraction* dengan keamanan konkuren.

2.1.4.3 Java

Java merupakan bahasa pemrograman berorientasi objek yang telah lama menjadi standar industri untuk aplikasi berskala besar. Dukungan terhadap concurrency muncul melalui API java.util.concurrent, yang mencakup *ExecutorService*, *Fork/Join Framework*, dan *Parallel Streams*. Framework ini memanfaatkan *thread pooling* dan *work-stealing algorithms* untuk meningkatkan efisiensi eksekusi paralel (Lea, 2000).

Java juga mengintegrasikan *Just-In-Time (JIT) compiler* yang mengoptimalkan performa saat runtime, serta *garbage collector (GC)* adaptif yang mengelola memori secara otomatis. Abhinav et al. (2020) mencatat bahwa Java memberikan hasil

optimal pada tugas *matrix multiplication* berukuran besar karena optimisasi JVM dan manajemen thread yang efisien. Namun, overhead GC dan kompleksitas sinkronisasi antar-thread dapat menjadi faktor pembatas dalam *low-latency systems*.

Dalam konteks *multi-core processing*, Java menonjol karena keseimbangan antara kemudahan pemrograman, performa stabil, dan ketersediaan pustaka paralel yang matang.

2.1.4.4 Python

Python dikenal sebagai bahasa dengan sintaks sederhana dan ekosistem luas untuk komputasi ilmiah, namun memiliki keterbatasan dalam eksekusi paralel akibat Global Interpreter Lock (GIL). Mekanisme ini hanya mengizinkan satu thread Python berjalan dalam satu waktu, sehingga menghambat *true parallelism* pada *CPU-bound tasks*. Untuk mengatasi hal ini, Python menyediakan modul multiprocessing yang membuat proses independen agar dapat berjalan secara paralel di beberapa core (Van der Walt & Aivazis, 2011).

Selain itu, pustaka eksternal seperti NumPy dan Numba memanfaatkan backend C dan BLAS yang teroptimasi, sehingga tetap mampu mencapai performa tinggi dalam komputasi numerik. Perkembangan terbaru, PEP 703, mengusulkan versi Python tanpa GIL, yang diharapkan membuka dukungan penuh terhadap eksekusi paralel sejati (Gross, 2023).

Python unggul dalam pengembangan cepat dan integrasi dengan pustaka HPC eksternal, namun secara native masih kurang efisien untuk beban komputasi berat berbasis CPU dibandingkan bahasa lain seperti Go atau Rust.

2.1.5 *Benchmark* Komputasi Paralel

Untuk mengevaluasi performa secara objektif, diperlukan serangkaian benchmark yang merepresentasikan pola-pola komputasi paralel fundamental. Benchmark yang dipilih dalam penelitian ini, seperti *parallel sorting* dan *pipeline* (untuk pola *producer-consumer*), merupakan contoh klasik yang dibahas secara luas dalam literatur mengenai pola pemrograman paralel (McCool et al., 2012).

2.1.5.1 Matrix Multiplication

Matrix-matrix multiplication (MMM) merupakan salah satu benchmark paling umum dalam pengujian *parallel computing* karena bersifat compute-bound dan memiliki pola data yang terstruktur. Michailidis dan Margaritis (2012) menunjukkan bahwa operasi ini sangat sensitif terhadap teknik optimasi seperti *loop interchange* dan *blocking*, yang dapat meningkatkan pemanfaatan cache dan efisiensi memori pada prosesor multi-core.

Akoushideh dan Shahbahrami (2022) menambahkan bahwa implementasi MMM menggunakan berbagai model pemrograman paralel seperti SIMD, OpenMP, Hybrid OpenMP-SIMD, dan OpenCL mampu mencapai *speedup* hingga 32 kali dibandingkan

versi sekuensial. Benchmark ini ideal untuk mengukur efektivitas *data-level parallelism* pada CPU karena seluruh elemen data dapat diproses secara independen dan simultan.

Dalam konteks penelitian ini, *matrix multiplication* dipilih untuk menguji kemampuan setiap bahasa dalam memanfaatkan *thread-level* maupun *data-level parallelism*, serta mengamati dampak strategi manajemen memori dan runtime terhadap efisiensi eksekusi.

2.1.5.2 Parallel Sorting

Parallel sorting digunakan untuk mengukur kinerja *task-level parallelism* dalam memproses beban kerja yang melibatkan sinkronisasi antar thread dan pembagian data yang tidak homogen. Algoritma seperti *parallel merge sort* dan *quick sort* memungkinkan pemrosesan bagian data secara bersamaan, lalu menggabungkannya kembali dalam tahap sinkronisasi.

Benchmark ini menggambarkan keseimbangan antara kecepatan komputasi dan biaya komunikasi antar thread. Java dan Rust, misalnya, memiliki dukungan pustaka yang baik untuk *parallel sorting* melalui *Fork/Join Framework* dan *Rayon crate*, sementara Go dan Python mengandalkan *goroutines* serta *multiprocessing* untuk menjalankan sorting secara konkuren. Pengujian ini penting untuk menilai overhead manajemen thread dan efisiensi pembagian tugas pada masing-masing bahasa.

2.1.5.3 Producer-Consumer Pattern

Pola *producer–consumer* digunakan untuk mengevaluasi efisiensi mekanisme *concurrency* yang melibatkan komunikasi dan koordinasi antar proses. Benchmark ini menggambarkan situasi di mana satu atau beberapa *producer* menghasilkan data, dan satu atau beberapa *consumer* memproses data tersebut secara bersamaan.

Model ini sering digunakan untuk menguji *message passing*, *channel communication*, dan *synchronization primitives* seperti *mutex*, *semaphore*, atau *queue*. Go, misalnya, secara native mendukung pola ini melalui *goroutines* dan *channels* (Castro et al., 2019), sementara Java mengimplementasikannya dengan *BlockingQueue* atau *ExecutorService*.

Benchmark *producer–consumer* termasuk kategori I/O-bound concurrency, sehingga cocok untuk menilai kemampuan bahasa dalam menangani tugas-tugas dengan tingkat interaksi tinggi, beban komunikasi besar, dan potensi *bottleneck* pada sinkronisasi.

2.1.5.4 Paramater Evaluasi

Dalam seluruh benchmark tersebut, performa diukur menggunakan beberapa metrik utama:

1. Waktu Eksekusi (Execution Time): total waktu yang dibutuhkan untuk menyelesaikan tugas tertentu.

2. Pemanfaatan CPU (CPU Utilization): tingkat penggunaan prosesor selama proses berjalan.
3. Konsumsi Memori (Memory Usage): jumlah memori yang digunakan selama eksekusi.
4. Skalabilitas (Scalability): perubahan performa terhadap penambahan jumlah inti prosesor.

Keempat metrik ini digunakan untuk memberikan gambaran kuantitatif terhadap efisiensi concurrency dan parallelism pada setiap bahasa yang diuji.

2.2 Penelitian Terdahulu

Penelitian terkait *concurrency* dan *parallelism* telah banyak dilakukan, dengan fokus yang bervariasi mulai dari aspek performa, model pemrograman, hingga keamanan eksekusi. Bagian ini membahas beberapa studi relevan yang menjadi dasar dan pembanding utama dalam penelitian ini.

2.2.1 Abhinav et al. (2020) Concurrency Analysis of Go and Java

Penelitian oleh (Abhinav et al., 2020) membandingkan performa dan arsitektur *concurrency* pada bahasa Go dan Java menggunakan pendekatan *thread-based* dan *goroutine-based parallelism*. Hasilnya menunjukkan bahwa Go memiliki keunggulan pada efisiensi manajemen *lightweight thread* (goroutines), terutama untuk *I/O-bound tasks*, sedangkan Java lebih stabil untuk *CPU-bound workloads* karena optimisasi JVM. Studi ini menegaskan perbedaan mendasar antara *concurrency* (koordinasi tugas) dan

parallelism (eksekusi simultan), yang menjadi dasar definisi teoritis penelitian ini.

2.2.2 Costanza et al. (2019) A Comparison of Three Programming Languages for a Full-Fledged NGS Tool

Costanza et al. (2019) melakukan evaluasi terhadap C++, Go, dan Java dalam pengembangan alat bioinformatika skala besar. Studi ini memperlihatkan bahwa *concurrency* digunakan untuk koordinasi proses analisis data secara pipeline, sedangkan *parallelism* diterapkan untuk mempercepat tahap komputasi intensif. Go menunjukkan efisiensi tinggi pada koordinasi tugas antar modul, sementara C++ unggul dalam performa numerik mentah. Studi ini menegaskan bahwa performa sistem tidak hanya ditentukan oleh kecepatan eksekusi paralel, tetapi juga oleh efisiensi manajemen *task concurrency*.

2.2.3 Michailidis & Margaritis (2012) Performance Study of Matrix Computations Using Multi-Core Programming Tools

Michailidis dan Margaritis (2012) menganalisis performa *matrix computations* pada arsitektur multi-core menggunakan OpenMP dan pustaka paralel lainnya. Hasil penelitian menunjukkan bahwa efisiensi *thread-based parallelism* sangat bergantung pada keseimbangan pembagian beban dan optimasi memori seperti *loop tiling* dan *blocking*. Studi ini memperkuat pentingnya analisis *data-level parallelism* dalam menentukan performa sistem komputasi ilmiah, serta menjadi dasar pemilihan *matrix multiplication* sebagai *benchmark* dalam penelitian ini.

2.2.4 Akoushideh & Shahbahrami (2022) Performance Evaluation of Matrix–Matrix Multiplication

Penelitian ini menguji berbagai implementasi *parallel matrix multiplication* menggunakan OpenMP, SIMD, OpenCL, dan model hibrid OpenMP–SIMD. Hasilnya menunjukkan bahwa pendekatan hibrid menghasilkan *speedup* tertinggi, mencapai 32 kali lebih cepat dari versi sekuensial. Temuan ini memperkuat konsep *hybrid parallelism*, yaitu kombinasi antara *thread-level* dan *data-level parallelism*, yang relevan untuk menganalisis bahasa seperti Go dan Rust yang menggabungkan keduanya dalam eksekusi runtime.

2.2.5 Castro et al. (2019) Distributed Programming Using Role-Parametric Session Types

Castro et al. (2019) memperkenalkan model *role-parametric session types* untuk pemrograman terdistribusi yang aman. Studi ini memperluas konsep CSP (Communicating Sequential Processes) dengan menambahkan tipe peran yang memastikan komunikasi antar entitas terjadi secara terstruktur dan bebas dari kesalahan sinkronisasi. Model ini menjadi acuan dalam pemahaman *structured concurrency* pada Go dan bahasa serupa yang mengandalkan *message-passing model*.

2.2.6 Yuan & Yang (2020) Effective Concurrency Testing for Distributed Systems

Yuan dan Yang (2020) membahas metode pengujian efektivitas concurrency dalam sistem terdistribusi menggunakan *systematic concurrency testing (SCT)*. Pendekatan ini menyoroti kompleksitas bug yang muncul akibat *race condition* dan *non-deterministic scheduling*, serta

menawarkan strategi untuk mendeteksi kesalahan melalui eksplorasi ruang eksekusi. Penelitian ini memberikan kontribusi teoretis terhadap pemahaman risiko *race condition* dalam sistem konkuren seperti yang diimplementasikan di Rust dan Go.

2.2.7 Roscoe (2005) The Theory and Practice of Concurrency

Roscoe (2005) memberikan landasan matematis bagi model *Communicating Sequential Processes (CSP)* yang menjadi dasar bagi banyak bahasa modern, termasuk Go dan Erlang. Model ini menjelaskan bahwa proses dapat berjalan secara independen dan hanya berinteraksi melalui saluran komunikasi, yang menjadi fondasi bagi *message-passing concurrency*. Teori CSP ini menjadi rujukan utama dalam membedakan *shared-memory concurrency* (Java, C++) dan *message-based concurrency* (Go, Erlang).

BAB III
METODE PENELITIAN

3.1 Jenis dan Pendekatan Penelitian

Penelitian ini merupakan penelitian eksperimental dengan pendekatan kuantitatif. Menurut Sugiyono (2020), penelitian kuantitatif dilakukan untuk mengetahui pengaruh suatu perlakuan terhadap variabel lain dalam kondisi yang terkontrol. Pendekatan kuantitatif digunakan karena berorientasi pada data numerik yang dapat diolah secara statistik untuk menjawab rumusan masalah penelitian secara objektif. Dalam konteks ini, seluruh pengujian dilakukan melalui pengukuran terukur seperti waktu eksekusi, penggunaan CPU, dan konsumsi memori pada masing-masing bahasa pemrograman.

Selaras dengan itu, Sadish et al. (2002) menekankan bahwa penelitian eksperimental harus dirancang dengan prinsip *controlled variation* yakni setiap perubahan hasil harus dapat ditelusuri ke perlakuan yang diberikan. Dalam penelitian ini, perlakuan yang dimaksud berupa variasi bahasa pemrograman (Go, Rust, Java, dan Python) serta jumlah thread atau *goroutines* yang dijalankan.

Pendekatan eksperimental dipilih karena memungkinkan pengamatan langsung terhadap pengaruh perubahan konfigurasi sistem terhadap performa eksekusi program secara terukur. Sementara pendekatan kuantitatif dipilih karena hasil penelitian berupa data numerik (waktu, memori, dan CPU usage) yang dapat dianalisis menggunakan metrik *speedup ratio* dan *efficiency* untuk menilai efektivitas masing-masing bahasa dalam mengimplementasikan *concurrent* dan *parallel programming*.

3.2 Kerangka Kerja Penelitian

Kerangka kerja penelitian ini menggambarkan tahapan-tahapan sistematis yang dilakukan peneliti untuk memperoleh data, menguji hipotesis, dan menarik kesimpulan secara ilmiah. Penelitian ini dirancang dengan pendekatan eksperimental, sehingga setiap tahap berfokus pada kontrol variabel, replikasi pengujian, serta analisis kuantitatif terhadap hasil pengukuran performa sistem. Secara umum, penelitian ini terdiri atas lima tahapan utama, yaitu:

3.2.1 Studi Literatur

Tahapan awal yang bertujuan untuk mengidentifikasi teori dasar, model pemrograman, serta hasil penelitian terdahulu yang relevan dengan topik *concurrency* dan *parallelism*. Literatur yang digunakan mencakup teori dari Amdahl, (1967); Gustafson, (1988); Roscoe, (2005), serta penelitian empiris seperti Abhinav et al., (2020); Akoushideh & Shahbahrami, (2022); Costanza et al., (2019); Michailidis & Margaritis, (2012). Hasil dari tahap ini digunakan untuk merumuskan variabel eksperimen dan memilih algoritma *benchmark* yang representatif.

3.2.2 Perancangan Eksperimen

Pada tahap ini ditentukan:

1. Variabel bebas: bahasa pemrograman (Go, Rust, Java, dan Python) serta jumlah thread/goroutines (1, 2, 4, 8, dan 16).
2. Variabel terikat: waktu eksekusi, pemanfaatan CPU, penggunaan memori, dan rasio *speedup*.

3. Variabel kontrol: algoritma dan dataset yang sama untuk tiap bahasa, lingkungan uji yang identik, serta versi compiler dan interpreter yang dikendalikan.

Desain penelitian menggunakan model eksperimen faktorial 4×5, di mana empat bahasa pemrograman diuji dengan lima variasi jumlah thread. Analisis hasil didasarkan pada Hukum Amdahl dan Gustafson untuk menilai efisiensi dan skalabilitas sistem.

3.2.3 Implementasi Benchmark

Setiap algoritma diimplementasikan dengan logika yang identik dalam empat bahasa pemrograman berbeda:

1. Go: menggunakan *goroutines* dan *channels* (model CSP).
2. Rust: menggunakan *async/await* dan pustaka *Tokio* untuk *task scheduling*.
3. Java: menggunakan *Fork/Join Framework* dan *ExecutorService*.
4. Python: menggunakan *multiprocessing* dan pustaka *NumPy*.

Tiga algoritma uji yang digunakan meliputi:

1. *Matrix Multiplication*: *compute-bound workload*.
2. *Parallel Sorting*: *mixed workload*.
3. *Producer–Consumer Pattern*: *I/O-bound workload*.

Ketiganya dipilih karena mewakili pola kerja paralel yang umum digunakan dalam sistem multi-core modern.

3.2.4 Pelaksanaan Pengujian

Eksperimen dilakukan pada lingkungan sistem yang dikontrol, menggunakan perangkat keras dengan spesifikasi konstan dan sistem operasi tunggal. Setiap kombinasi konfigurasi dijalankan sebanyak lima kali pengulangan, dan nilai median digunakan untuk mengurangi pengaruh *noise* sistem. Data dikumpulkan menggunakan alat ukur seperti:

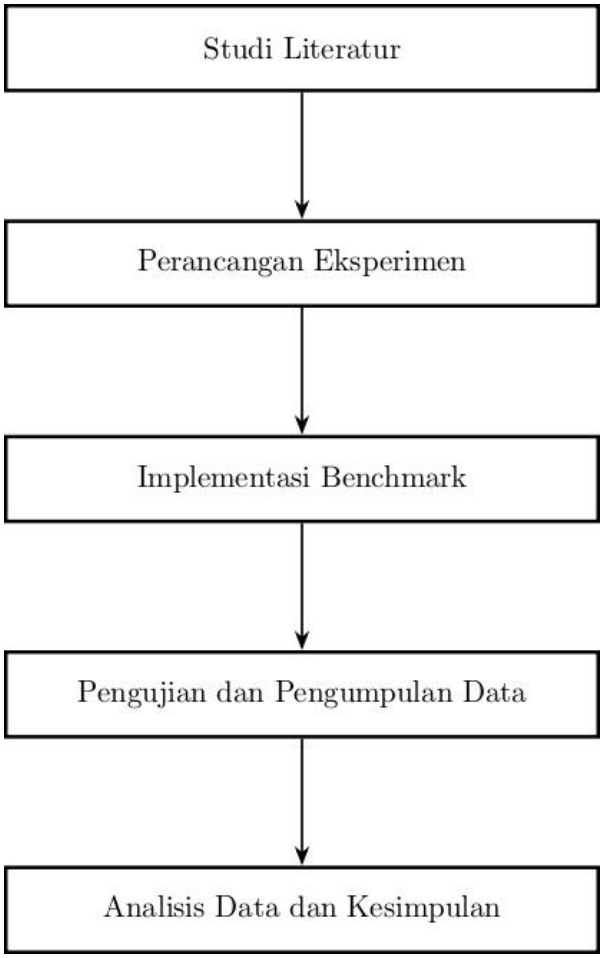
a. /usr/bin/time -v untuk waktu eksekusi dan penggunaan memori.
b. perf stat -d untuk siklus CPU, *cache misses*, dan efisiensi pemrosesan.

Mode *CPU governor* diset ke performance agar frekuensi prosesor tetap stabil selama pengujian.

3.2.5 Analisis dan Interpretasi Hasil

Data hasil pengujian dianalisis menggunakan pendekatan kuantitatif deskriptif dan komparatif. Langkah-langkah analisis mencakup:

1. Perhitungan *mean*, *median*, dan *standard deviation*.
2. Perhitungan *speedup (S(p))* dan *efficiency (E(p))* terhadap baseline (1 thread).
3. Pembuatan grafik hubungan *jumlah thread vs waktu eksekusi* dan *speedup vs efficiency*.
4. Interpretasi hasil dengan membandingkan teori concurrency dan parallelism di Bab II.



Gambar 1. Diagram Alur Kerja Penelitian

3.3 Uraian Kerangka Kerja Penelitian

3.3.1 Studi Literatur

Tahap ini bertujuan untuk memperoleh landasan teoritis dan pemahaman menyeluruh mengenai konsep *concurrency* dan *parallelism*, model pemrograman yang digunakan, serta hasil penelitian terdahulu yang relevan. Kegiatan yang dilakukan pada tahap ini meliputi:

1. Mengumpulkan dan mempelajari sumber pustaka seperti buku, artikel ilmiah, serta prosiding konferensi yang berkaitan dengan pemrograman konkuren dan paralel.

2. Mengidentifikasi teori dasar seperti Amdahl's Law (1967) dan Gustafson's Law (1988) sebagai acuan untuk mengukur batas kecepatan sistem paralel.
3. Menganalisis penelitian terdahulu seperti Abhinav et al. (2020), Costanza et al. (2019), Michailidis & Margaritis (2012), dan Akoushideh & Shahbahrami (2022) yang membahas implementasi *concurrency* pada berbagai bahasa pemrograman.

Hasil dari tahap ini digunakan untuk menentukan arah penelitian, memilih algoritma uji (*benchmark*), dan merumuskan variabel eksperimen.

3.3.2 Perancangan Eksperimen

Tahap ini berfokus pada perumusan desain penelitian dan variabel yang akan diuji. Langkah-langkah yang dilakukan meliputi:

3.3.2.1 Penetapan Algoritma Benchmark

Algoritma *benchmark* yang digunakan meliputi:

1. Matrix Multiplication, representasi *compute-bound task*.
2. Parallel Sorting, representasi *mixed workload*.
3. Producer–Consumer Pattern, representasi *I/O-bound task*.

Ketiga algoritma ini dipilih karena mewakili pola kerja umum dalam sistem komputasi paralel dan konkuren.

3.3.2.2 Penentuan Variabel Penelitian

Penelitian ini menggunakan tiga jenis variabel, yaitu:

1. Variabel bebas: bahasa pemrograman (Go, Rust, Java, Python) dan jumlah thread/goroutines (1, 2, 4, 8, 16).

2. Variabel terikat: waktu eksekusi, pemanfaatan CPU, penggunaan memori, dan rasio speedup.
3. Variabel kontrol: algoritma, ukuran input data, lingkungan sistem, serta versi compiler/interpreter yang digunakan.

3.3.2.3 Rancangan Desain Eksperimen Faktorial 4×5

Bahasa	Threads	Repetisi	Ukuran Input
Go	1-16	5 ×	512-4096
Rust	1-16	5 ×	512-4096
Java	1-16	5 ×	512-4096
Python	1-16	5 ×	512-4096

Tabel 1. Rancangan Desain Eksperimen

Desain eksperimental ini mengombinasikan empat bahasa pemrograman dan lima variasi jumlah thread/goroutines. Setiap kombinasi dijalankan sebanyak lima kali pengulangan untuk memperoleh hasil median yang stabil dan mengurangi *noise* sistem.

3.3.2.4 Penentuan Metode Analisis Data

Analisis hasil dilakukan menggunakan metrik *speedup* dan *efficiency* berdasarkan Hukum Amdahl dan Gustafson untuk mengukur *scalability* serta *parallel efficiency*.

$$S(p) = \frac{T_1}{T_P} \; ; \; E(p) = \frac{S(p)}{p}$$

dengan:

T_1 = waktu eksekusi pada 1 thread (*baseline*),

T_p = waktu eksekusi pada p thread,

$S(p)$ = *speedup* pada jumlah thread p,

$E(p)$ = efisiensi paralel sistem.

3.3.3 Implementasi Benchmark

Tahap ini merupakan proses penerjemahan rancangan eksperimen ke dalam implementasi kode nyata. Setiap algoritma diimplementasikan menggunakan gaya dan pustaka *concurrency/parallelism* yang idiomatik untuk masing-masing bahasa, yaitu:

- Go: menggunakan *goroutines* dan *channels* (CSP model).
- Rust: menggunakan *async/await* dan pustaka *Tokio* untuk *task scheduling*.
- Java: menggunakan *Fork/Join Framework* serta *ExecutorService*.
- Python: menggunakan *multiprocessing* dan pustaka *NumPy*.

Setiap implementasi dioptimalkan untuk menjaga kesetaraan logika dan algoritmik, sehingga hasil perbandingan antar bahasa dapat dianggap adil (*fair comparison*).

3.3.4 Pengujian dan Pengumpulan Data

Tahap pengujian dilakukan dengan menjalankan seluruh implementasi pada perangkat keras dan sistem operasi yang sama untuk menjamin konsistensi hasil. Langkah-langkah pengujian meliputi:

1. Menjalankan setiap konfigurasi eksperimen sebanyak lima kali pengulangan.

2. Menggunakan alat ukur performa sistem, antara lain:
    a. `time -v` untuk waktu eksekusi dan penggunaan memori.
    b. `perf stat -d` untuk pengukuran siklus CPU, *cache misses*, dan efisiensi eksekusi.
3. Menyimpan hasil dalam format log dan mengekstrak nilai median dari tiap pengukuran.
4. Menetapkan *CPU governor* pada mode **performance** untuk menjaga kestabilan frekuensi prosesor selama eksperimen berlangsung.

Hasil dari tahap ini berupa dataset performa numerik yang siap untuk dianalisis.

3.3.5 Analisis Data dan Kesimpulan

Tahap ini bertujuan untuk menganalisis dan menginterpretasikan hasil pengujian menggunakan pendekatan kuantitatif. Langkah-langkah analisis meliputi:

1. Menghitung nilai rata-rata, median, dan *standard deviation* untuk setiap konfigurasi.
2. Menghitung *speedup ratio (S(p))* dan *efficiency (E(p))* berdasarkan hasil baseline (1 thread).
3. Membuat visualisasi berupa grafik hubungan antara jumlah thread dan waktu eksekusi, serta grafik *speedup–efficiency*.
4. Membandingkan hasil dengan teori Amdahl dan Gustafson guna menilai skalabilitas performa.

5. Menarik kesimpulan mengenai efektivitas implementasi *concurrency* dan *parallelism* pada masing-masing bahasa pemrograman.

Tahap ini menghasilkan interpretasi akhir yang menjadi dasar penyusunan kesimpulan penelitian pada Bab V.

3.4 Tempat Penelitian

Penelitian ini dilaksanakan di lingkungan pengembangan lokal menggunakan perangkat keras dan perangkat lunak yang dikonfigurasi secara terkontrol untuk menjamin konsistensi hasil eksperimen. Seluruh pengujian dilakukan secara *offline* untuk menghindari interferensi proses eksternal dan memastikan hasil yang stabil serta dapat direplikasi. Sistem yang digunakan berbasis Ubuntu 24.04 LTS (Noble Numbat) sebagai platform modern dengan dukungan penuh terhadap komputasi multi-core dan *multi-threaded execution*.

3.4.1 Spesifikasi Perangkat Keras

Komponen	Spesifikasi
Perangkat	ASUS Vivobook S 14 (Model M5406UA)
Prosesor	AMD Ryzen 7 8845HS (8 cores / 16 threads, hingga 5.1 GHz)
GPU	AMD Radeon 780M (Integrated RDNA3)
Memori (RAM)	16 GB LPDDR5X @ 7467 MHz
Penyimpanan	512 GB NVMe SSD
Resolusi Layar	1920 × 1200 @ 60 Hz
Mode Daya	*Performance mode* (CPU governor: performance)

Tabel 2. Perangak Keras Praktikan

3.4.2 Spesifikasi Perangkat Lunak

Komponen	Versi / Keterangan
Sistem Operasi	Ubuntu 24.04.3 LTS (Linux kernel 6.14.0-33-generic)
Desktop Environment	GNOME 46.2 (Wayland)
Bahasa Pemrograman	Go 1.22, Rust 1.75, Java 21 (OpenJDK), Python 3.12
Compiler / Toolchain	GCC 14, binutils 2.42, glibc 2.39 *(toolchain default Ubuntu 24.04)*
IDE / Editor	Neovim 0.9.5 Build type: Release LuaJIT 2.1.1703358377 dan terminal Z shell 5.9
Alat Analisis	/usr/bin/time, perf stat, htop, sysbench

Tabel 3. Perangkat Lunak Praktikan

Berdasarkan catatan rilis Ubuntu 24.04 LTS, versi bawaan toolchain adalah rustc 1.75, Go 1.22, GCC 14, dan Python 3.12 (Caconical, 2024). Versi-versi ini dipilih untuk menjamin kestabilan dan kompatibilitas dengan ekosistem *multi-core development* modern.

3.4.3 Lokasi dan Kondisi Eksperimen

Seluruh eksperimen dilakukan di sistem pribadi peneliti dengan kondisi lingkungan terkontrol sebagai berikut:

1. Semua proses dijalankan dalam mode performance untuk menjaga frekuensi prosesor konstan selama uji.
2. Aplikasi latar belakang non-esensial (browser, editor, media player) dinonaktifkan selama pengujian.

3. Setiap konfigurasi dijalankan sebanyak lima kali, dan hasil median digunakan untuk mengurangi pengaruh *noise*.
4. Semua hasil eksekusi disimpan dalam bentuk *log file* dan diolah menggunakan skrip Python untuk analisis kuantitatif.

Penelitian ini dilaksanakan pada bulan Oktober 2025, menggunakan seluruh perangkat keras dan perangkat lunak dalam kondisi terkini (stable release), agar hasil pengujian mencerminkan performa lingkungan pengembangan modern berbasis arsitektur *multi-core*.

3.5 Teknik Analisis Data

Analisis data pada penelitian ini dilakukan menggunakan pendekatan kuantitatif deskriptif dan komparatif, dengan tujuan menilai performa dan efisiensi implementasi *concurrency* serta *parallelism* pada empat bahasa pemrograman modern: Go, Rust, Java, dan Python.

Data yang dianalisis meliputi waktu eksekusi, pemanfaatan CPU, dan penggunaan memori dari setiap kombinasi konfigurasi eksperimen. Setiap konfigurasi dijalankan sebanyak lima kali pengulangan, dan nilai median digunakan sebagai representasi hasil akhir untuk mengurangi pengaruh *noise* sistem atau fluktuasi beban CPU.

3.5.1 Analisis Kuantitatif Deskriptif

Tahap ini bertujuan untuk menggambarkan karakteristik dasar dari hasil eksperimen. Data performa dikumpulkan dalam format numerik dan diolah menggunakan *Python script* dengan pustaka NumPy dan Pandas. Langkah-langkah analisis deskriptif mencakup:

1. Menghitung nilai rata-rata (*mean*), median, serta standar deviasi (*standard deviation*) untuk tiap kombinasi variabel.
2. Menampilkan hasil dalam bentuk tabel performa per bahasa pemrograman dan per jumlah thread.
3. Membuat visualisasi menggunakan Matplotlib untuk menunjukkan hubungan antara jumlah thread dengan waktu eksekusi, *speedup*, dan efisiensi.

3.5.2 Analisis Perbandingan (Comparative Analysis)

Analisis perbandingan dilakukan untuk menilai perbedaan performa antar bahasa pemrograman terhadap beban dan kondisi eksekusi yang identik. Langkah ini dilakukan dengan menghitung rasio performa relatif antara setiap bahasa menggunakan persamaan berikut:

$$R_{i,j} = \frac{T_i}{T_j}$$

dengan:

T_i= waktu eksekusi bahasa pemrograman ke-i,

T_j= waktu eksekusi bahasa pemrograman ke-j,

$R_{i,j}$= rasio perbandingan performa antara dua bahasa pemrograman.

Nilai $R_{i,j} > 1$ menunjukkan bahwa bahasa j lebih cepat daripada i, sedangkan $R_{i,j} < 1$ menandakan sebaliknya. Analisis ini membantu mengidentifikasi bahasa dengan efisiensi eksekusi terbaik pada tiap jenis workload (*compute-bound*, *I/O-bound*, atau *mixed*).

3.5.3 Analisis Skalabilitas

Analisis skalabilitas dilakukan untuk menilai sejauh mana kinerja meningkat seiring bertambahnya jumlah thread atau goroutines yang dijalankan. Dua metrik utama digunakan dalam tahap ini: speedup dan efficiency, berdasarkan Amdahl's Law dan Gustafson's Law.

$$S(p) = \frac{T_1}{T_p}; E(p) = \frac{S(p)}{p}$$

dengan:

T_1= waktu eksekusi pada satu thread (*baseline*),

T_p= waktu eksekusi pada $p$thread,

$S(p)$= *speedup* terhadap baseline,

$E(p)$= efisiensi sistem paralel terhadap jumlah thread.

Interpretasi:

$S(p) \approx p \rightarrow$ *linear scalability*,

$E(p) > 0.8 \rightarrow$ efisiensi sangat baik,

$E(p) < 0.5 \rightarrow$ terdapat overhead tinggi atau *bottleneck concurrency*.

3.5.4 Visualisasi dan Interpretasi Hasil

Hasil analisis disajikan dalam bentuk grafik dan tabel agar pola performa lebih mudah diamati. Visualisasi dilakukan menggunakan pustaka Matplotlib dan Seaborn dengan tiga bentuk utama:

1. Grafik hubungan antara jumlah thread (X) dan waktu eksekusi (Y),
2. Grafik hubungan antara jumlah thread (X) dan speedup (Y),
3. Grafik hubungan antara jumlah thread (X) dan efficiency (Y).

Interpretasi dilakukan dengan cara:

1. Menganalisis tren peningkatan atau penurunan kinerja tiap bahasa terhadap pertambahan thread.
2. Mengamati titik jenuh (*saturation point*) di mana penambahan thread tidak lagi meningkatkan performa.
3. Mengaitkan hasil empiris dengan teori concurrency dan parallelism pada Bab II, serta hukum Amdahl dan Gustafson sebagai kerangka konseptual.

3.5.5 Kesimpulan Akhir Analisis

Tahap akhir ini bertujuan untuk menarik simpulan kuantitatif dari seluruh hasil analisis, meliputi:

1. Bahasa pemrograman dengan performa terbaik pada tiap jenis workload.
2. Bahasa dengan efisiensi tertinggi dalam pemanfaatan *multi-core processor*.
3. Pola skalabilitas masing-masing bahasa dan keterbatasannya.
4. Kesesuaian hasil empiris dengan teori skalabilitas paralel.
5. Kesimpulan dari tahap ini menjadi dasar utama penyusunan hasil dan pembahasan pada Bab IV, serta rekomendasi pengembangan sistem berbasis *concurrent* dan *parallel programming* di masa mendatang.

DAFTAR PUSTAKA

Abhinav, P. Y., Bhat, A., Joseph, C. T., & Chandrasekaran, K. (2020). Concurrency analysis of go and java. *Proceedings of the 2020 International Conference on Computing, Communication and Security, ICCCS 2020*. https://doi.org/10.1109/ICCCS49678.2020.9277498

Akoushideh, A., & Shahbahrami, A. (2022). Performance Evaluation of Matrix-Matrix Multiplication using Parallel Programming Models on CPU Platforms. *Research Square Preprint*, 1–23.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings - 1967 Spring Joint Computer Conference, AFIPS 1967*, 483–485. https://doi.org/10.1145/1465482.1465560

Caconical. (2024). *Ubuntu 24.04 LTS (Noble Numbat) Release Notes*. Canonical Ltd. https://discourse.ubuntu.com/t/ubuntu-24-04-lts-noble-numbat-release-notes

Castro, D., Hu, R., Jongmans, S. S., Ng, N., & Yoshida, N. (2019). Distributed Programming using Role-Parametric Session Types in Go: Statically-typed endpoint APIs for dynamically-instantiated communication structures. *Proceedings of the ACM on Programming Languages*, *3*(POPL). https://doi.org/10.1145/3290342

Costanza, P., Herzeel, C., & Verachtert, W. (2019). A comparison of three programming languages for a full-fledged next-generation sequencing tool. *BMC Bioinformatics*, *20*(1), 1–10. https://doi.org/10.1186/s12859-019-2903-5

Gross, S. (2023). *PEP 703 – Making the Global Interpreter Lock Optional in CPython*. Python Software Foundation. https://peps.python.org/pep-0703/

Gustafson, J. L. (1988). Reevaluating amdahl's law. *Communications of the ACM*, *31*(5), 532–533. https://doi.org/10.1145/42411.42415

Jung, R., Jourdan, J. H., Krebbers, R., & Dreyer, D. (2021). Safe systems programming in Rust. *Communications of the ACM*, *64*(4), 144–152. https://doi.org/10.1145/3418295

Klabnik, S., & Nichols, C. (2018). *The Rust Programming Language* (2nd ed.). No Starch Press.

Lea, D. (2000). A Java fork/join framework. *ACM 2000 Java Grande Conference*, 36–43. https://doi.org/10.1145/337449.337465

Lee, S. Y., & Aggarwal, J. K. (1987). A Mapping Strategy for Parallel Processing. *IEEE Transactions on Computers*, *C–36*(4), 433–442. https://doi.org/10.1109/TC.1987.1676925

McCool, M., Robison, A. D., & Reinders, J. (2012). Structured Parallel Programming. In *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier. https://doi.org/10.1016/C2011-0-04251-5

Michailidis, P. D., & Margaritis, K. G. (2012). Performance study of matrix computations using multi-core programming tools. *ACM International Conference Proceeding Series*, 186–192. https://doi.org/10.1145/2371316.2371353

Nurwina Quirante, M., Lincopinis, D., Nurwina Quirante, M. A., Sumagang, E. M., & Lincopinis, D. R. (2023). *Go Programming Language: Overview*. *May*. https://orcid.org/0000-0001-9503-8965,

Roscoe, A. W. (2005). *The Theory and Practice of Concurrency*. Prentice Hall.

Sadish, W. R., Cook, T. D., & Campbell, D. T. (2002). *Experimental and Quasi-Experimental Designs For Generalized Casual Inference*. Houghton Mifflin Company.

Sugiyono. (2020). *Metode penelitian kuantitatif, kualitatif, dan R&D*. Alfabeta.

Van der Walt, S., & Aivazis, M. (2011). The NumPy Array: A Structure for Efficient Numerical Computation, Computing in Science & Engineering. *Computing in Science and Engineering*, *13*(2), 22–30. http://aip.scitation.org/doi/abs/10.1109/MCSE.2011.37

Yuan, X., & Yang, J. (2020). Effective concurrency testing for distributed systems. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 1141–1156. https://doi.org/10.1145/3373376.3378484