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.

2.1.6 Metodologi Evaluasi Performa Sistem Konkruen

Evaluasi performa sistem konkuren memerlukan metodologi yang sistematis dan terstandar untuk memastikan hasil yang valid dan dapat direproduksi. Menard et al. (2023) menekankan pentingnya beberapa aspek metodologi dalam benchmarking, yaitu: (1) warm-up iterations untuk menghindari bias dari cold start, (2) multiple runs dengan analisis statistik untuk mengurangi noise sistem, (3) penggunaan median atau geometric mean untuk agregasi hasil, dan (4) pengukuran confidence interval untuk menilai variabilitas hasil.

Diehl et al. (2023) menambahkan bahwa pengukuran performa harus memisahkan waktu inisialisasi dari waktu eksekusi utama, serta menggunakan mode CPU governor yang konsisten (seperti performance mode) untuk menghindari variasi frekuensi prosesor. Lebih lanjut, mereka menekankan pentingnya meminimalkan background processes dan

menggunakan alat ukur yang presisi seperti perf stat untuk mendapatkan metrik tingkat rendah seperti cache misses dan instruction per cycle (IPC).

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 Diehl et al. (2023) Benchmarking the Parallel 1D Heat Equation Solver

Penelitian ini membandingkan 10 bahasa pemrograman (Chapel, Charm++, C++, HPX, Go, Julia, Python, Rust, Swift, dan Java) menggunakan model problem 1D heat equation solver. Hasil menunjukkan Python sebagai yang paling lambat, sementara C++, Rust, Chapel, Charm++, dan HPX memberikan performa tertinggi. Studi ini menekankan pentingnya asynchronous programming dan queue-based communication untuk performa optimal (Diehl et al., 2023).

2.2.2 Menard et al. (2023) High-Performance Deterministic Concurrency using Lingua Franca

Menard et al. mengevaluasi Lingua Franca menggunakan Savina benchmark suite dan membandingkannya dengan Akka dan CAF. Hasil menunjukkan bahwa determinisme tidak mengurangi performa—Lingua Franca bahkan mengungguli Akka (1.86×) dan CAF (1.42×). Penelitian ini membuktikan bahwa deterministic concurrency dapat mencapai performa tinggi sambil mempertahankan reproducibility dan testability (Menard et al., 2023).

2.2.3 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.

2.2.4 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*.

2.2.5 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.

2.2.6 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.

2.2.7 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*.

2.2.8 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.

2.2.9 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).