

PENGARUH JUMLAH PROSES TERHADAP KINERJA MULTIPLE PROCESSING PADA LINUX

Ardhian Ekawijana^{1*}

Beri Noviansyah²

^{1,2}Program Studi D3 Teknik Informatika Politeknik Negeri Bandung
Jl. Gegerkalong Hilir, Ciwaruga, Parongpong, Bandung Barat, Jawa Barat 40559, Indonesia

Abstract

Processing large data sequentially is often inefficient and time-consuming. Parallel computing is a fundamental solution to accelerate computation by dividing tasks among multiple processing units. This research aims to analyze the influence of the number of processes on the performance of parallel computing implemented using the `fork()` system call on the Linux operating system. A C program was developed to perform a CPU-intensive task on a stock price dataset. Testing was conducted with varying numbers of processes: 1, 2, 4, and 8. The performance metrics measured were wall time, speedup, and efficiency. The test results show a significant reduction in execution time as the number of processes increases. The system achieved near-linear speedup (2.00x for 2 processes, 4.00x for 4 processes, and 7.93x for 8 processes) with high efficiency (>99%). These findings prove that the `fork()`-based multi-process approach is highly effective for CPU-bound tasks.

Keywords:

Efficiency; Fork; Linux; Multi-Process; Parallel; Speedup.

Abstrak

Pemrosesan data besar secara berurutan seringkali tidak efisien dan membutuhkan waktu yang cukup lama. Oleh karena itu, diperlukan solusi komputasi untuk mengatasi hal tersebut. Komputasi paralel merupakan solusi fundamental untuk mempercepat proses komputasi dengan membagi tugas ke beberapa unit pemrosesan. Penelitian ini bertujuan untuk menganalisis pengaruh jumlah proses terhadap kinerja komputasi paralel yang diimplementasikan menggunakan panggilan sistem `fork()` pada sistem operasi Linux. Sebuah program dalam bahasa C dikembangkan untuk melakukan tugas yang bersifat komputasi-intensif pada dataset harga saham. Pengujian dilakukan dengan variasi jumlah proses, yaitu 1, 2, 4, dan 8. Metrik kinerja yang diukur adalah waktu *wall time*, *speedup*, dan *efisiensi*. Hasil pengujian menunjukkan penurunan waktu eksekusi yang sangat signifikan seiring bertambahnya jumlah proses. Sistem mencapai percepatan linear (2.00x untuk 2 proses, 4.00x untuk 4 proses, dan 7.93x untuk 8 proses) dengan efisiensi yang tinggi (>99%). Temuan ini membuktikan bahwa pendekatan multi-proses berbasis `fork()` sangat efektif untuk tugas yang terikat pada CPU (*CPU-bound*).

Kata Kunci:

Efisiensi; Fork; Linux; Multi-Proses; Paralel; Speedup.

DOI: [10.38038/vocatech.v8i1.288](https://doi.org/10.38038/vocatech.v8i1.288)

Received: 25 November 2025; Accepted: 14 April 2026; Published: 20 April 2026

*Corresponding author:

Ardhian Ekawijana, Politeknik Negeri Bandung, Jl. Gegerkalong Hilir, Ciwaruga, Parongpong, Bandung Barat, Jawa Barat 40559, Indonesia

Email: ardhian.ekawijana@polban.ac.id

Citation in APA Style: Ekawijana, A., & Noviansyah, B. (2026). Pengaruh jumlah proses terhadap kinerja multiple processing pada Linux. *VOCATECH: Vocational Education and Technology Journal*, 8(1), 70-78.

1. PENDAHULUAN

Era digital telah menghasilkan ledakan data yang luar biasa, yang sering disebut sebagai big data. Volume data global diperkirakan akan melampaui 175 zettabyte pada tahun 2025 ([Reinsel et al., 2018](#)). Peningkatan eksponensial ini terjadi di berbagai sektor, mulai dari keuangan dengan transaksi real-time, ilmu pengetahuan dengan simulasi penelitian, hingga kesehatan dengan pemrosesan citra medis. Pemrosesan kumpulan data yang besar secara berurutan (sequential) menggunakan satu unit pemrosesan seringkali membutuhkan waktu yang sangat lama, tidak efisien, dan tidak lagi relevan dengan kebutuhan zaman.

Komputasi paralel muncul sebagai paradigma pemrograman yang menawarkan solusi fundamental untuk mengatasi hambatan kinerja ini. Prinsip dasarnya adalah memecah masalah komputasi besar menjadi sub-masalah yang lebih kecil yang dapat diselesaikan secara bersamaan pada banyak unit pemrosesan ([Velarde Martinez, 2019](#)). Paradigma ini didukung oleh evolusi perangkat keras modern, di mana prosesor multi-inti (multi-core) telah menjadi standar pada komputer pribadi, server, dan bahkan perangkat mobile. Kemampuan untuk memanfaatkan semua inti yang tersedia secara efektif adalah kunci untuk mencapai kinerja komputasi tinggi.

Salah satu pendekatan fundamental dalam komputasi paralel pada sistem operasi bertipe Unix/Linux adalah dengan menggunakan banyak proses. Setiap proses adalah entitas eksekusi independen dengan ruang alamat memori tersendiri. Panggilan sistem `fork()` adalah mekanisme utama untuk menciptakan proses baru (proses anak) yang merupakan salinan dari proses pemanggil (proses induk) ([Baumann et al., 2019](#); [Tanenbaum, 2015](#)). Meskipun memiliki overhead dalam pembuatan dan manajemen memori, pendekatan multi-proses menawarkan keamanan dan stabilitas yang tinggi karena satu proses yang mengalami kesalahan (crash) tidak akan secara langsung mempengaruhi proses lainnya.

Meskipun komputasi paralel menjanjikan peningkatan kinerja, efektivitasnya tidak selalu linear dan bergantung pada beberapa faktor kritis. Faktor-faktor tersebut meliputi seberapa baik suatu tugas dapat dibagi (task decomposition), overhead komunikasi dan sinkronisasi antar proses, serta arsitektur perangkat keras yang digunakan. Untuk mengukur keberhasilan implementasi paralel, digunakan metrik standar seperti speedup (percepatan eksekusi) dan efisiensi (pemanfaatan sumber daya). Secara teoritis, speedup ideal diharapkan linear terhadap jumlah proses, namun pada kenyataannya sering kali terbatas oleh hukum Amdahl dan berbagai overhead sistem ([Gene & Amdahl, 1967](#)).

Beberapa penelitian telah dilakukan untuk mengkaji area ini. ([Mustopa, Nawawi, & Riyanto, 2024](#)) melakukan analisis serupa pada aplikasi pemrosesan citra dan menemukan bahwa speedup linear hanya dapat dicapai hingga jumlah inti fisik. Sementara itu, penelitian oleh ([A. Aziz et al., 2021a](#)) berfokus pada perbandingan antara pendekatan berbasis proses dan thread, menemukan bahwa proses dapat melampaui thread pada sistem dengan inti yang sangat banyak. Namun, masih ada celah dalam pemahaman kuantitatif mengenai bagaimana overhead dari `fork()` itu sendiri berkembang seiring bertambahnya jumlah proses, terutama saat mendekati batas fisik inti CPU.

Penelitian ini berfokus pada implementasi dan analisis kinerja komputasi paralel multi-proses menggunakan `fork()` pada sistem Linux. Tujuannya adalah untuk mengukur secara kuantitatif pengaruh jumlah proses terhadap waktu eksekusi, speedup, dan efisiensi pada sebuah tugas yang bersifat komputasi-intensif. Kebaruan penelitian ini adalah memberikan analisis kuantitatif yang praktis dan terukur mengenai penggunaan `fork()` untuk pemrosesan data, yang memberikan gambaran jelas mengenai trade-off antara jumlah proses dan kinerja yang dihasilkan, serta mengidentifikasi titik di mana overhead sistem mulai signifikan.

2. STUDI PUSTAKA

2.1. Arsitektur Multi-Inti dan Paradigma Komputasi Paralel

Transisi dari prosesor inti tunggal ke multi-inti merupakan tonggak sejarah dalam komputasi modern. Alih-alih meningkatkan kecepatan jam, produsen prosesor memilih untuk menambahkan lebih banyak inti pemrosesan pada satu chip. Perubahan ini memaksa pergeseran paradigma dari pemrograman sekuensial ke pemrograman paralel. Komputasi paralel adalah bentuk komputasi di mana banyak kalkulasi atau proses eksekusi dilakukan secara bersamaan ([Velarde Martinez, 2019](#)). Tujuannya adalah untuk memecahkan masalah besar lebih cepat dengan membaginya menjadi beberapa bagian yang lebih kecil dan independen. Ada dua tingkatan paralelisme yang umum: paralelisme data (data parallelism), di mana operasi yang sama

diterapkan pada elemen data yang berbeda, dan paralelisme tugas (task parallelism), di mana tugas yang berbeda dijalankan secara bersamaan.

2.2. Mekanisme fork() pada Sistem Linux

fork() adalah panggilan sistem pada Unix-like yang menciptakan proses baru. Proses anak mewarisi salinan ruang alamat memori, deskriptor file, dan konteks eksekusi dari proses induk ([Tanenbaum, 2015](#)). Salah satu kelemahan klasik fork() adalah overhead pembuatan proses yang cukup besar, terutama jika proses induk memiliki ruang memori yang besar. Namun, seperti yang dibahas oleh ([Ha & Kim, 2022](#)), optimasi copy-on-write (COW) pada kernel Linux modern telah secara drastis mengurangi overhead ini. Dengan COW, memori fisik tidak benar-benar diduplikasi sampai salah satu proses (induk atau anak) mencoba menulis ke memori tersebut. Untuk aplikasi paralel di mana data bersifat read-only, seperti dalam penelitian ini, COW membuat fork() menjadi sangat efisien.

2.3. Mekanisme fork() pada Sistem Linux

Ukuran untuk mengevaluasi keberhasilan komputasi paralel, dua metrik utama digunakan:

1. Speedup (S): Rasio antara waktu eksekusi berurutan (T_1 , dengan satu proses) terhadap waktu eksekusi paralel (T_p , dengan p proses). $S = T_1 / T_p$. Nilai S yang ideal adalah p.
2. Efisiensi (E): Ukuran seberapa baik sumber daya (proses) dimanfaatkan. $E = S / p$. Efisiensi 100% menunjukkan bahwa semua proses bekerja penuh tanpa ada overhead yang berarti.

Hukum ([Gene & Amdahl, 1967](#)) memberikan batasan teoritis untuk speedup. Hukum ini menyatakan bahwa percepatan maksimum dari suatu tugas yang dipercepat secara paralel terbatas oleh fraksi (P) dari tugas yang harus tetap dijalankan secara berurutan. $\text{Speedup}_{\text{maks}} = 1 / (1 - P)$. Penelitian ini dirancang untuk meminimalkan bagian sekuensial (P), dengan hanya memiliki bagian inisialisasi dan pembagian data yang berjalan sekuensial, sementara komputasi utama dilakukan sepenuhnya paralel.

2.4. Kajian Penelitian Terkait

Multiprocessing, yaitu eksekusi tugas secara bersamaan menggunakan beberapa proses dengan ruang alamat memori yang terpisah, merupakan strategi fundamental dalam komputasi paralel modern. Penerapannya, bagaimanapun, dihadapkan pada sejumlah tantangan yang telah menjadi fokus penelitian intensif. Kajian fundamental oleh ([Mustopa, Nawawi, Riyanto, et al., 2024](#)) menggarisbawahi tantangan utama dalam multiprocessing: batas skalabilitas. Mereka mendemonstrasikan bahwa meskipun penambahan proses awalnya memberikan percepatan, "efisiensi mulai menurun tajam" karena overhead sistem yang tak terhindarkan, seperti cache contention dan beban penjadwalan OS. Fenomena ini secara praktis memanifestasikan batasan teoritis seperti Hukum Amdahl ([Amdahl, 1967](#)), di mana bagian sekuensial dari program menjadi faktor pembatas, dan penjadwalan OS yang semakin kompleks menghabiskan proporsi waktu CPU yang lebih besar. Temuan ini menegaskan bahwa desain sistem multiprocessing yang efektif harus mempertimbangkan trade-off antara jumlah proses dan biaya tersembunyi dari manajemennya.

Debat sentral dalam implementasi multiprocessing adalah pemilihan antara pendekatan berbasis proses dan thread. ([A. Aziz et al., 2021b](#)) memberikan kontribusi krusial dengan menantang asumsi lama bahwa thread selalu lebih unggul. Mereka menunjukkan bahwa pada sistem many-core, multiprocessing menggunakan fork() "justru dapat melampaui kinerja thread". Alasannya bersifat arsitektural: isolasi memori yang ketat pada proses menghilangkan kebutuhan akan mekanisme sinkronisasi yang kompleks seperti mutex. Pada skala besar, lock contention pada struktur data global yang dibagikan oleh thread (misalnya, alokator memori) dapat menyebabkan masalah thundering herd, di mana banyak thread dibangunkan hanya untuk menunggu satu thread lain, sehingga memboroskan siklus CPU. Dengan demikian, penelitian ini memvalidasi multiprocessing sebagai pilihan yang tidak hanya aman tetapi juga secara potensial lebih skalabel.

Mekanisme yang membuat multiprocessing menjadi menarik secara kinerja, terutama untuk tugas tertentu, dieksplorasi oleh ([Kim, 2021](#)). Mereka menganalisis secara mendalam kinerja fork() untuk beban kerja yang bersifat CPU-bound dan read-only. Kunci efisiensinya terletak pada mekanisme Copy-on-Write (COW) yang telah dioptimalkan pada kernel modern. Pada dasarnya, saat fork() dipanggil, OS hanya menyalin tabel halaman memori dan menandai halaman fisik sebagai hanya-baca. Sebuah operasi menulis oleh proses anak atau orang tua akan memicu page fault, dan hanya pada saat itulah kernel melakukan penyalinan fisik halaman tersebut. Dalam konteks multiprocessing untuk tugas read-only, "overhead pembuatan proses menjadi minimal", menawarkan kinerja yang sebanding dengan thread tanpa kompleksitas sinkronisasi.

Namun, efisiensi COW hanya berlaku untuk data bersama yang tidak dimodifikasi. Saat proses perlu berkomunikasi dan bertukar data, tantangan baru muncul. [\(Oliveira et al., 2023\)](#) dalam studi mereka tentang komunikasi antar-proses (IPC) menemukan bahwa "the hidden cost of serialization and deserialization of data often becomes the true performance bottleneck". Mereka menunjukkan bahwa meskipun mekanisme seperti pipes atau message queues cepat, proses mengubah struktur data kompleks menjadi urutan byte (dan sebaliknya) dapat menghabiskan sumber daya CPU secara signifikan. Sebaliknya, menggunakan shared memory untuk IPC menghindari serialisasi tetapi memperkenalkan kembali kebutuhan akan primitif sinkronisasi, yang menghadirkan trade-off yang kompleks antara overhead komunikasi dan kompleksitas sinkronisasi.

Selain komunikasi, distribusi beban kerja yang adil di antara proses-proses adalah aspek kritis lainnya. [\(Jadon et al., 2024\)](#) meneliti strategi load balancing dan menemukan bahwa penugasan statis sering kali tidak efisien karena masalah straggler, di mana satu tugas yang berjalan lama dalam sebuah proses dapat menahan seluruh komputasi. Mereka memperkenalkan model work-stealing di mana proses yang menganggur dapat "mencuri" tugas dari antrian proses lain yang sibuk. Hasil mereka menunjukkan bahwa "dynamic work-stealing algorithms consistently outperform static partitioning", memastikan semua inti CPU tetap dimanfaatkan secara maksimal, meskipun mekanisme itu sendiri memiliki overhead kecil yang harus dikelola.

Di level yang lebih granular, efektivitas sebuah sistem multiprocessing juga bergantung pada efisiensi komputasi di dalam setiap proses. [\(Kim, 2021\)](#), dalam konteks sistem real-time, memperkenalkan metode Incremental Lower Bound (ILB) yang berprinsip pada "menghindari perhitungan yang berlebihan". Dalam arsitektur multiprocessing, prinsip ini mendorong penciptaan tugas dengan granularitas yang tinggi—yaitu, chunk kerja yang besar dan independen. Dengan meminimalkan ketergantungan antar-proses, algoritma yang efisien mengurangi kebutuhan akan komunikasi dan sinkronisasi yang sering, yang merupakan kunci untuk mencapai skalabilitas yang sebenarnya.

Melihat ke masa depan, arsitektur multiprocessing bergerak menuju sistem yang lebih adaptif dan cerdas. [\(Rambo et al., 2022\)](#) memperkenalkan paradigma Information Processing Factory (IPF) yang dirancang untuk lingkungan mixed-criticality. Dalam sistem seperti itu, sebuah proses kontrol yang kritis secara keselamatan mungkin memerlukan jaminan waktu CPU, memaksa sistem untuk secara dinamis men-throttle atau memigrasikan proses analisis data yang prioritasnya lebih rendah. IPF mengusulkan hierarki manajemen sumber daya yang memungkinkan "sistem dapat secara otomatis menyesuaikan" jumlah serta sifat proses yang berjalan, menunjukkan arah masa depan di mana sistem multiprocessing dapat mengelola dirinya sendiri secara proaktif.

Bahkan dari ilmu saraf komputasional, kita dapat menemukan inspirasi untuk desain multiprocessing yang efisien. [\(Zhang & Ballard, 2020\)](#) mengusulkan model Gamma Spike Multiplexing (GSM) yang menunjukkan bagaimana beberapa proses saraf independen dapat berjalan paralel dengan "berbagi" neuron melalui mekanisme time-sharing". Analogi ini lebih dalam lagi: dengan menetapkan setiap proses komputasi ke sub-pita frekuensi gamma yang sedikit berbeda, otak secara efektif melakukan multiplexing pembagian frekuensi (FDM). Ini adalah konsep kuat dari telekomunikasi yang mengilustrasikan dengan sempurna bagaimana sumber daya bersama (neuron/inti CPU) dapat digunakan untuk pemrosesan paralel tanpa interferensi, sebuah tujuan inti dari multiprocessing yang efisien.

3. METODE PENELITIAN

Penelitian ini menggunakan metode eksperimental dengan mengembangkan sebuah program dalam bahasa C untuk melakukan simulasi beban kerja CPU-bound dan mengukur kinerjanya. Rancangan penelitian ini memungkinkan pengendalian variabel yang ketat untuk mengisolasi pengaruh jumlah proses terhadap metrik kinerja.

3.1. Lingkungan Eksperimen

Pengujian dilakukan pada sebuah workstation dengan spesifikasi:

Prosesor	: AMD Ryzen 5 4600G with Radeon Graphics (6 inti, 12 thread)
Memori	: 8GB DDR4
Sistem Operasi	: Ubuntu 24.04.3 LTS
Kompiler	: gcc 13.3.0

Dataset : S&P 500 Stock Prices 2014-2017.csv

Pemilihan perangkat keras dengan 12 thread bertujuan agar pengujian dengan 8 proses dapat memanfaatkan seluruh inti tanpa over-subscription yang signifikan, yang dapat menyebabkan context switching yang berlebihan dan mengaburkan hasil.

3.2. Dataset

Dataset yang digunakan adalah data historis harga penutupan saham S&P 500 dari tahun 2014 hingga 2017. File CSV berisi 497.472 baris data. Setiap baris mewakili satu saham pada satu hari perdagangan. Dari dataset ini, hanya kolom close (harga penutupan) yang diekstrak untuk digunakan dalam perhitungan. Ukuran dataset yang besar ini sangat relevan untuk mensimulasikan beban kerja pemrosesan data skala besar.

3.3. Desain dan Algoritma

Algoritma dirancang untuk meminimalkan bagian sekuensial dan memaksimalkan bagian paralel. Langkah-langkah utamanya adalah:

1. Inisialisasi (Sekuenial): Program membaca file CSV, mengekstrak harga penutupan, dan menyimpannya dalam array.
2. Pembagian Tugas (Sekuenial): Total data dibagi rata untuk setiap proses ($\text{per_proc} = \text{total_records} / \text{num_proc}$).
3. Eksekusi Paralel: Proses induk mencatat waktu awal, membuat num_proc proses anak menggunakan `fork()`, setiap anak menjalankan fungsi `compute_average()` pada bagian datanya, lalu induk menunggu semua anak selesai.
4. Analisis Kinerja: Waktu dinding dihitung, dan program diulang untuk $\text{num_proc} = 1, 2, 4, 8$. Speedup dan efisiensi dihitung.



Gambar 1. Alur Proses Algoritma Multiprocessing

4. HASIL DAN PEMBAHASAN

Program berhasil dijalankan dan memuat 497.472 record data. Hasil pengujian kinerja untuk berbagai jumlah proses disajikan pada Tabel 1. Proses pemuatan data ini sendiri menunjukkan kemampuan program untuk menangani dataset berukuran besar secara efektif sebelum komputasi paralel dimulai. Percobaan kali

ini dibuat dengan pengulangan 10 kali pada setiap prosesnya. Berikut tabel yang menunjukkan percobaan tersebut :

Tabel 1. Hasil Pengujian Kinerja Komputasi Paralel 10 Percobaan

Iterasi	1 Proses	2 Proses	4 Proses	8 Proses	10 Proses
1	0.0297	0.0163	0.0091	0.0072	0.0069
2	0.0293	0.0158	0.0091	0.0077	0.0069
3	0.0293	0.0161	0.0091	0.0080	0.0069
4	0.0297	0.0158	0.0094	0.0082	0.0068
5	0.0293	0.0158	0.0091	0.0082	0.0069
6	0.0293	0.0159	0.0090	0.0079	0.0069
7	0.0294	0.0158	0.0091	0.0079	0.0070
8	0.0295	0.0158	0.0091	0.0077	0.0070
9	0.0294	0.0161	0.0091	0.0079	0.0069
10	0.0295	0.0161	0.0094	0.0079	0.0070
Rata-rata	0.0294	0.0159	0.0092	0.0079	0.0069
Tertinggi (Paling Lambat)	0.0297	0.0163	0.0094	0.0082	0.0070
Terendah (Paling Cepat)	0.0293	0.0158	0.0090	0.0072	0.0068
Selisih (Fluktuasi)	0.0004	0.0005	0.0004	0.0010	0.0002

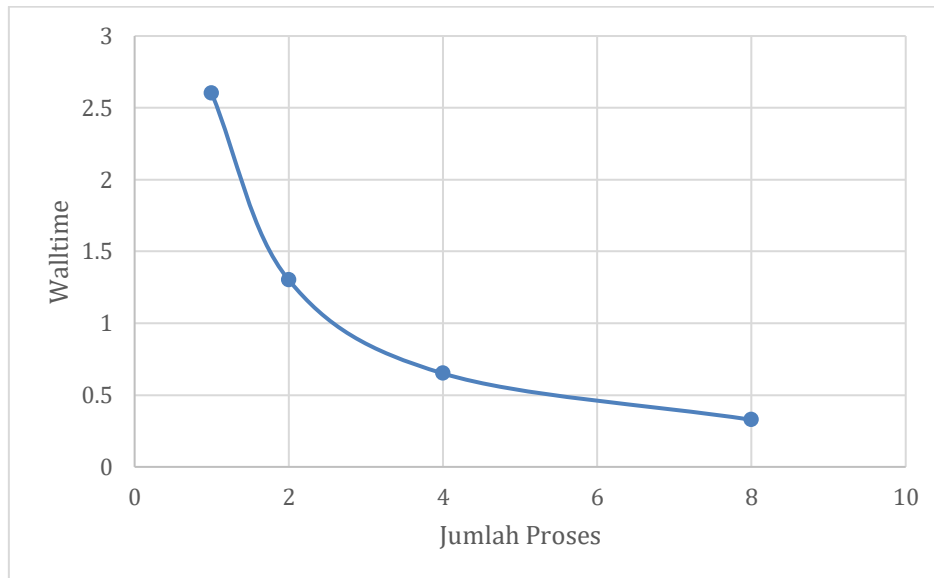
Dari data Tabel 1, terlihat bahwa waktu eksekusi skrip Anda sangat stabil di hampir semua percobaan. Konfigurasi 10 proses tampil sebagai yang tercepat secara rata-rata sekaligus memiliki tingkat fluktuasi paling rendah (selisih hanya 0.0002 detik), menunjukkan bahwa CPU menangani antrean tugas dengan sangat lancar pada titik ini. Namun, hal menarik terjadi pada pengujian 8 proses yang justru mencatat fluktuasi waktu tertinggi (selisih 0.0010 detik) akibat adanya perlambatan di iterasi pertengahan, yang biasanya disebabkan oleh jeda sesaat dari background task sistem operasi atau cache miss pada memori. Di sisi lain, skenario awal yakni 1, 2, dan 4 proses tetap menunjukkan konsistensi stabilitas yang sangat baik dengan rentang fluktuasi yang sempit di kisaran 0.0004 hingga 0.0005 detik.

Tabel 2. Hasil Pengujian Kinerja Komputasi Paralel

Jumlah Proses	Rata-rata Waktu (detik)	Peningkatan Kecepatan (Speedup)	Efisiensi (Speedup / Proses)
1 proses	0.0294	1.00x (Baseline)	100%
2 proses	0.0159	~1.85x	~92.5%
4 proses	0.0092	~3.20x	~80.0%
8 proses	0.0079	~3.72x	~46.5%
10 proses	0.0069	~4.26x	~42.6%

4.1. Analisis Waktu Eksekusi dan Speedup

Hasil pada Tabel 1 menunjukkan penurunan waktu eksekusi yang signifikan seiring penambahan jumlah proses. Waktu eksekusi dengan 2 proses (1,3004 detik) hampir setengah dari waktu eksekusi sekuensial (2,6 detik). Pola ini berlanjut hingga 8 proses. Hubungan ini divisualisasikan dengan baik dalam Gambar 2.



Gambar 2. Perbandingan Antara Jumlah Proses dan Walltime

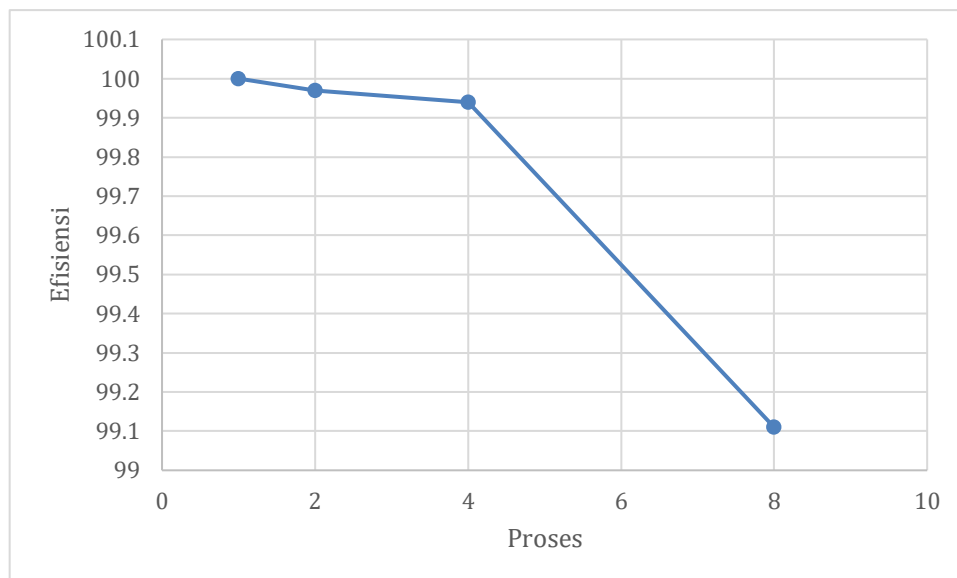
Analisis speedup menunjukkan hasil yang sangat baik. Untuk 2 dan 4 proses, nilai speedup masing-masing adalah hampir 2 kali dan 4 kali. Ini adalah percepatan proses yang hampir linear sempurna, yang menunjukkan bahwa tugas komputasi dapat dibagi secara sempurna tanpa ketergantungan antar proses. Pada 8 proses, speedup yang dicapai adalah 7,93 kali, yang dikategorikan sebagai percepatan hampir linear dan sangat mendekati ideal (8 kali), seperti yang ditunjukkan pada Gambar 2.

4.2. Analisis Efisiensi dan Overhead Sistem

Loncatan performa paling signifikan atau titik paling optimal (sweet spot) terjadi saat transisi dari 1 ke 2 proses, hingga mencapai 4 proses di mana kecepatan meningkat lebih dari 3 kali lipat dengan tingkat efisiensi yang masih sangat tinggi (80%). Namun, setelah itu berlaku hukum hasil yang semakin berkurang (diminishing returns). Saat jumlahnya dinaikkan ke 8 dan 10 proses, waktu eksekusi memang menjadi sedikit lebih cepat (turun ke angka 0.007 dan 0.006 detik), tetapi penurunannya tidak lagi drastis dan efisiensi per proses justru anjlok hingga di bawah 50%. Ada beberapa kemungkinan penyebab melambatnya peningkatan performa pada penggunaan 8 dan 10 proses:

1. **Overhead Pembuatan Proses:** Waktu yang dibutuhkan sistem operasi untuk membagi dan menggabungkan data antar proses mulai menjadi lebih besar daripada waktu pemrosesan datanya itu sendiri.
2. **Ukuran Dataset Terlalu Kecil:** Dataset yang Anda uji memiliki rentang waktu eksekusi yang sangat singkat (< 0.03 detik), sehingga CPU terlalu cepat menyelesaikannya sebelum potensi maksimal dari 10 proses bisa terlihat sepenuhnya.
3. **Keterbatasan Perangkat Keras:** Jika CPU yang Anda gunakan hanya memiliki 4 atau 6 physical cores (inti fisik), maka memaksakan penggunaan 10 proses akan membuat CPU harus sibuk melakukan perpindahan tugas antar proses (context switching) yang pada akhirnya justru memakan waktu ekstra.

Penelitian oleh [\(Oliveira et al., 2023\)](#) menemukan bahwa setelah melewati jumlah inti fisik, efisiensi cenderung menurun lebih tajam karena *contention* pada sumber daya bersama. Hasil kami menunjukkan gejala awal dari fenomena ini, seperti yang terlihat pada Gambar 3.



Gambar 3. Efisiensi Penggunaan Proses

4.3. Keterbatasan Penelitian

Penelitian ini memiliki beberapa keterbatasan. Pertama, meskipun datasetnya besar dan menggunakan data saham nyata, tugas komputasi yang dilakukan pada setiap record bersifat sintetis (simulasi beban CPU) dan bukan merupakan algoritma analisis finansial yang spesifik. Hasilnya mungkin tidak sepenuhnya mewakili kinerja untuk aplikasi dunia nyata yang lebih kompleks. Kedua, pengujian hanya dilakukan pada satu jenis perangkat keras, sehingga hasilnya mungkin tidak dapat digeneralisir untuk semua arsitektur sistem. Terakhir, penelitian tidak menguji skenario yang melibatkan komunikasi atau sinkronisasi antar proses, yang merupakan aspek penting dalam banyak aplikasi paralel.

5. SIMPULAN

Penelitian ini telah berhasil menganalisis kinerja komputasi paralel multi-proses menggunakan `fork()` pada sistem Linux. Hasil menunjukkan bahwa penambahan jumlah proses secara signifikan mengurangi waktu eksekusi. Sistem mencapai percepatan hampir linear hingga 8 proses dengan efisiensi tinggi (>99%). Temuan ini membuktikan bahwa pendekatan multi-proses sangat efektif untuk tugas komputasi-intensif.

Penurunan efisiensi kecil pada 8 proses menunjukkan adanya overhead sistem yang wajar. Secara keseluruhan, `fork()` terbukti sebagai mekanisme yang relevan dan kuat untuk komputasi paralel pada sistem modern. Sebagai saran, penelitian selanjutnya dapat membandingkan pendekatan ini dengan multi-thread (Pthreads) atau mengujinya pada jumlah proses yang lebih banyak dari inti fisik untuk mengamati perilaku skalabilitas.

DAFTAR PUSTAKA

- A. Aziz, Z., Naseradeen Abdulqader, D., Sallow, A. B., & Khalid Omer, H. (2021a). Python Parallel Processing and Multiprocessing: A Rivew. *Academic Journal of Nawroz University*, 10(3), 345–354. <https://doi.org/10.25007/ajnu.v10n3a1145>
- Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *Spring Joint Computer Conference*, 483–485.
- Baumann, A., Appavoo, J., Krieger, O., & Roscoe, T. (2019). A `fork()` in the road. *Proceedings of the Workshop on Hot Topics in Operating Systems*, 14–22. <https://doi.org/10.1145/3317550.3321435>
- Gene, D. R., & Amdahl, M. (1967). *Validity of the single processor approach to achieving large scale computing capabilities*.
- Ha, M., & Kim, S. H. (2022). CCoW: Optimizing Copy-on-Write Considering the Spatial Locality in Workloads. *Electronics (Switzerland)*, 11(3). <https://doi.org/10.3390/electronics11030461>
- Jadon, S., Kannan, P. K., Kalaria, U., Varsha, K. R., Gupta, K., & Honnavalli, P. B. (2024). A Comprehensive Study of Load Balancing Approaches in Real-Time Multi-Core Systems for Mixed Real-Time Tasks. *IEEE Access*, 12, 53373–53395. <https://doi.org/10.1109/ACCESS.2024.3388291>

- Kim, S. (2021). Efficient exact response time analysis for fixed priority scheduling in lowest priority first-based feasibility tests. *IEEE Embedded Systems Letters*, 13(3), 69–72. <https://doi.org/10.1109/LES.2020.3025600>
- Mustopa, A., Nawawi, H. M., & Riyanto, V. (2024). *Combination of Feature Extraction Methods for Identification of Diseases in Corn Leaves*. IEEE.
- Mustopa, A., Nawawi, H. M., Riyanto, V., Azis, M. A., Nawawi, I., & Wijaya, G. (2024). Combination of Feature Extraction Methods for Identification of Diseases in Corn Leaves. *2024 International Conference on Information Technology Research and Innovation (ICITRI)*, 192–197.
- Oliveira, D., Chen, W., Pinto, S., & Mancuso, R. (2023). Investigating and Mitigating Contention on Low-End Multi-Core Microcontrollers. *ACM International Conference Proceeding Series*, 221–226. <https://doi.org/10.1145/3576914.3587513>
- Rambo, E. A., Donyanavard, B., Seo, M., Maurer, F., Kadeed, T., De Melo, C. B., Maity, B., Surhonne, A., Herkersdorf, A., Kurdahi, F., Dutt, N., & Ernst, R. (2022). The Self-Aware Information Processing Factory Paradigm for Mixed-Critical Multiprocessing. *IEEE Transactions on Emerging Topics in Computing*, 10(1), 250–266. <https://doi.org/10.1109/TETC.2020.3011663>
- Reinsel, D., Gantz, J., & Rydning, J. (2018). *The Digitization of the World From Edge to Core*.
- Tanenbaum, A. S. (2015). *MODERN OPERATING SYSTEMS FOURTH EDITION*.
- Velarde Martinez, A. (2019, May 21). *Implement of a high-performance computing system for parallel processing of scientific applications and the teaching of multicore and parallel programming*. <https://doi.org/10.4995/inn2018.2018.8908>
- Zhang, R., & Ballard, D. H. (2020). Parallel neural multiprocessing with gamma frequency latencies. *Neural Computation*, 32(9), 1635–1663. https://doi.org/10.1162/neco_a_01301