Evaluasi Performa *Multithreading* dalam Algoritma Pencarian Data Berbasis CPU pada Sistem Operasi *Multi-Core*

Gusti Ahmad Muttahid Bilhaq[1], Ahmad Mukhlis Farhan[2], Eko Prasetio Widhi[3]
Teknik Informatika, Universitas Darussalam Gontor
e-mail: gustimuttahidbilhaq73@student.cs.unida.gontor.ac.id[1],
ahmadmukhlisfarhan93@student.cs.unida.gontor.ac.id[2], ekoprasetiowidhi@unida.gontor.ac.id[3]

Kata Kunci:

Multithreading
Global Interpreter Lock (GIL)
Sistem Operasi *Multi-Core*
Algoritma CPU-*Bound*
Speedup Ratio

ABSTRAK

Eksekusi algoritma pencarian data berbasis *single-threaded* rentan mengalami *bottleneck* saat memproses *dataset* berskala besar karena tidak seluruh *core* CPU pada sistem operasi *multi-core* termanfaatkan secara optimal. Penelitian ini mengevaluasi efisiensi, waktu eksekusi, dan rasio speedup pemrosesan CPU-*bound search algorithm* pada dataset 20 juta data acak, menggunakan skema *multithreading* dengan variasi 1, 2, 4, 8, 16, dan 32 *thread*. Pengujian dilakukan pada perangkat AMD Ryzen 9 5900HX (8 core/16 *logical processor*), RAM 16 GB, sistem operasi Windows 11 Home. Hasil pengujian menunjukkan waktu eksekusi *baseline* pada 1 *thread* sebesar 688,51 ms (*speedup* 1,00x), sementara peningkatan jumlah *thread* hingga 16 *thread* hanya menghasilkan *speedup marginal*, dengan nilai tertinggi 1,03x pada 16 *thread* (667,37 ms). Fenomena ini disebabkan oleh mekanisme *Global Interpreter Lock* (GIL) pada level *interpreter* yang membatasi eksekusi paralel murni untuk beban CPU-*bound*, sehingga penambahan *thread* tidak berbanding lurus dengan peningkatan performa. Pada pengujian 32 *thread*, waktu eksekusi justru membengkak menjadi 716,13 ms dengan *speedup* merosot ke 0,96x, membuktikan munculnya *overhead context switching* dan perebutan *resource* oleh *kernel* sistem operasi ketika jumlah *thread* melampaui kapasitas *logical processor*. Temuan ini menegaskan bahwa pemetaan alokasi *thread* terhadap arsitektur *hardware* dan karakteristik *interpreter* sangat menentukan *throughput* komputasi optimal.

Keyword:

First Keyword
Second Keyword
Third Keyword
Fourth Keyword
Fifth Keyword

ABSTRACT

Single-threaded execution of data search algorithms is prone to performance bottlenecks when processing large-scale datasets, since not all CPU cores on multi-core operating systems are optimally utilized. This study evaluates the efficiency, execution time, and speedup ratio of a CPU-bound search algorithm on a dataset of 20 million random records, using a multithreading scheme with 1, 2, 4, 8, 16, and 32 threads. Testing was conducted on an AMD Ryzen 9 5900HX (8 cores/16 logical processors), 16 GB RAM, running Windows 11 Home. The results show a baseline execution time of 688.51 ms at 1 thread (speedup 1.00x), while increasing the thread count up to 16 threads produced only marginal gains, peaking at 1.03x (667.37 ms). This pattern is attributed to the Global Interpreter Lock (GIL) at the interpreter level, which restricts parallel execution for CPU-bound workloads, so adding threads does not translate proportionally into performance. At 32 threads, execution time increased to 716.13 ms with speedup dropping to 0.96x, confirming context-switching overhead and resource contention by the operating system kernel once thread count exceeds the available logical processors. These findings indicate that thread allocation should match both hardware architecture and interpreter characteristics for optimal throughput.

Copyright © SENASIF 2026
ISSN: 2598-0076

I. PENDAHULUAN

Pencarian data pada *dataset* berskala besar adalah operasi yang hampir selalu muncul dalam sistem perangkat lunak modern, mulai dari basis data transaksional sampai aplikasi pengolahan *big data*. Pada skema *single-threaded*, seluruh beban pencarian ditanggung satu alur eksekusi saja, sehingga waktu pemrosesan tumbuh linear seiring bertambahnya ukuran data. Masalah ini terasa janggal justru pada prosesor *multi-core*, karena sebagian besar kapasitas komputasinya menganggur selama proses berjalan. Sistem operasi modern sebenarnya sudah menyediakan mekanisme *scheduling* yang memungkinkan beban kerja didistribusikan ke banyak *core*, tetapi pemanfaatannya bergantung penuh pada bagaimana aplikasi di atasnya menerapkan konkurensi, dan penambahan jumlah *thread* tidak selalu berbanding lurus dengan peningkatan kinerja karena sistem operasi dapat dipaksa melakukan *context switching* berlebihan begitu jumlah *thread* aktif melampaui kapasitas *core* fisik yang tersedia [1].

Ekspektasi terhadap *speedup* pada arsitektur *multi-core* kerap berbenturan dengan realitas batasan *runtime environment*, khususnya pada *interpreter* bahasa pemrograman tertentu. Pada studi evaluasi performa *multithreading* di lingkungan aplikasi *desktop*, penambahan *thread* justru tidak selalu diikuti perbaikan waktu eksekusi yang signifikan, sebab efisiensi paralelisasi turut ditentukan oleh karakteristik beban kerja dan *overhead* manajemen *thread* itu sendiri [2]. Pola serupa juga ditemukan pada penerapan *multiprocessing* dan *multithreading* di perangkat berdaya rendah, di mana pemilihan skema konkurensi yang tepat justru lebih menentukan efisiensi ketimbang sekadar menambah jumlah unit eksekusi [3]. Pada *interpreter* CPython secara khusus, keberadaan Global Interpreter Lock (GIL) membatasi eksekusi *bytecode* Python hanya pada satu thread dalam satu waktu, sehingga skema *multiprocessing* sering menjadi alternatif untuk beban CPU-*bound* meski harus menanggung *overhead* komunikasi antar-proses yang jauh lebih besar dibanding *threading* [4]. Kondisi ini pada dasarnya berakar dari batasan fundamental komputasi paralel, di mana bagian sekuensial dari suatu proses akan selalu membatasi perolehan *speedup* maksimum yang bisa dicapai, betapapun banyak unit pemroses yang ditambahkan [5].

Dari titik itu, penelitian ini mencoba menjawab pertanyaan sejauh mana penambahan alokasi *worker*, dari 1 sampai 32, benar-benar berbanding lurus dengan peningkatan performa pencarian data CPU-*bound* pada sistem operasi *multi-core*. Secara spesifik, penelitian ini bertujuan mengukur efisiensi waktu eksekusi dan rasio *speedup* pada algoritma pencarian data terhadap *dataset* 20 juta data acak, dengan variasi alokasi *worker* 1, 2, 4, 8, 16, dan 32, pada perangkat AMD Ryzen 9 5900HX berkapasitas 8 *core* dan 16 *logical processor* dengan sistem operasi Windows 11.

Mengacu pada karakteristik GIL yang telah diuraikan, penelitian ini berhipotesis bahwa penambahan *worker* pada rentang 1 sampai 16 hanya akan menghasilkan *speedup* yang stagnan atau marginal, sementara pada 32 *worker* performa berpotensi menurun akibat *overhead context switching* yang dilakukan kernel sistem operasi ketika jumlah *worker* melampaui kapasitas *logical processor*.

Copyright © SENASIF 2026
ISSN: 2598-0076

Bagi pengembang perangkat lunak paralel, temuan ini tidak hanya berupa catatan teoretis, karena keputusan menambah *thread* tanpa mempertimbangkan karakteristik *interpreter* dan arsitektur *hardware* berisiko menurunkan *throughput* alih-alih menaikkannya. Kontribusi empiris penelitian ini diharapkan dapat memberi pertimbangan konkret bagi perancangan sistem yang mengandalkan pemrosesan data skala besar di lingkungan *multi-core*, sekaligus memperkaya kajian sejenis dalam konteks sistem operasi dan komputasi paralel di lingkungan riset dalam negeri.

II. METODE

Penelitian ini menggunakan pendekatan kuantitatif eksperimental melalui skema pengujian *benchmark* performa. Variabel bebasnya adalah jumlah alokasi *worker*, baik dalam bentuk *thread* maupun *process*, dengan enam taraf perlakuan: 1, 2, 4, 8, 16, dan 32 *worker*. Variabel terikatnya berupa waktu eksekusi dalam milidetik dan rasio *speedup* pada tiap taraf perlakuan, mengikuti metrik yang lazim dipakai dalam evaluasi kinerja komputasi paralel berbasis *worker* [6]. Variabel kontrol dijaga tetap sepanjang pengujian, mencakup ukuran dataset sebanyak 20 juta data, skenario *worst-case* pada algoritma *linear search*, serta spesifikasi *hardware* dan sistem operasi, sehingga perbedaan hasil yang muncul dapat dikaitkan langsung dengan perubahan jumlah *worker*, bukan faktor lain di luar itu.

Populasi data berupa larik bilangan bulat acak sebanyak 20 juta elemen, dibangkitkan dengan modul *random* pada rentang 1 sampai 1.000.000. Seluruh 20 juta elemen dipakai sebagai sampel uji tanpa *sampling* parsial, sebab pengujian CPU-*bound* justru membutuhkan beban kerja penuh agar *bottleneck*-nya benar-benar teramati [7]. Target pencarian ditetapkan pada nilai -999, angka yang sengaja tidak pernah muncul dalam dataset, sehingga setiap worker wajib menelusuri seluruh segmen datanya sendiri hingga akhir. Skema ini merepresentasikan kondisi *worst-case* pada *linear search*, sejalan dengan pendekatan pengukuran waktu eksekusi dan penggunaan memori pada algoritma pencarian sekuensial yang menekankan pentingnya skenario tanpa kemungkinan pencarian berhenti lebih awal untuk memperoleh gambaran kinerja yang representatif [6].

Eksperimen dijalankan pada satu unit perangkat dengan spesifikasi tetap: prosesor AMD Ryzen 9 5900HX berkapasitas 8 *core* dan 16 *logical processor*, RAM 16 GB, sistem operasi Windows 11 Home 64-bit. Pengujian dijalankan di atas *runtime* Python menggunakan modul *concurrent.futures.ThreadPoolExecutor* untuk skema multithreading, dan *ProcessPoolExecutor* untuk skema *multiprocessing* sebagai pembanding. *Dataset* dibagi rata ke tiap *worker* menggunakan skema *chunking* berbasis indeks awal dan akhir, dengan sisa pembagian dialokasikan ke *worker* terakhir agar tidak ada data yang terlewat, mengikuti pola distribusi *task* independen yang umum diterapkan pada *task parallelism* berbasis *worker pool* [8].

Waktu eksekusi diukur dengan *time.perf_counter()*, fungsi penghitung waktu presisi tinggi pada Python yang tidak terpengaruh oleh penyesuaian jam sistem operasi, sehingga sesuai untuk

Copyright © SENASIF 2026

ISSN: 2598-0076

mengukur interval singkat pada level milidetik sekaligus mengisolasi *overhead* langsung dari *overhead* tidak langsung akibat gangguan *cache* maupun penjadwalan interupsi prosesor [9]. Untuk mengurangi bias akibat *cold start* dan fluktuasi beban sistem, setiap taraf jumlah *worker* diuji sebanyak tiga kali pengulangan *(triplo)*, kemudian hasilnya dirata-rata untuk memperoleh nilai waktu eksekusi yang lebih stabil dan representatif. Analisis data dilakukan dengan menghitung rasio *speedup* menggunakan formula $S = T1 / Tn$, dengan *T1* sebagai waktu eksekusi pada satu *worker (baseline)* dan *Tn* sebagai waktu eksekusi pada konfigurasi *n worker* [8]. Degradasi performa pada jumlah *worker* yang tinggi dianalisis lebih lanjut dengan mengaitkannya pada dua sumber *overhead* yang berbeda karakter: *context switching* pada skema *multithreading*, yang melibatkan biaya langsung penyimpanan dan pemulihan konteks serta biaya tidak langsung berupa gangguan *cache* prosesor dan *overhead* komunikasi antar-proses (IPC) beserta biaya serialisasi data pada skema *multiprocessing.*

III. HASIL DAN PEMBAHASAN

Analisis Kinerja *Multithreading*

Pengujian pada skema *multithreading* menghasilkan pola yang jauh dari ekspektasi teoretis linear speedup. Tabel 1 merangkum hasil lengkap pengujian pada enam taraf jumlah thread.

Tabel 1. Ringkasan Hasil Uji *Multithreading (ThreadPoolExecutor)*

Jumlah *Thread*	Waktu Eksekusi (ms)	*Speedup*
1	688,51	1,00x (Baseline)
2	663,00	1,04x
4	685,75	1,00x
8	684,05	1,01x
16	667,37	1,03x
32	716,13	0,96x

Dari rentang 1 sampai 16 *thread, speedup* yang diperoleh hanya bergerak tipis antara 1,00x sampai 1,04x, jauh dari peningkatan proporsional yang diharapkan pada prosesor dengan 16 *logical processor*. Pola stagnasi ini konsisten dengan temuan pada optimasi manajemen *thread* di sistem operasi, di mana penambahan *thread* murni tanpa disertai teknik seperti *thread pooling* atau penjadwalan dinamis cenderung tidak memberi perbaikan performa berarti pada beban kerja yang terikat pada satu sumber daya bersama [10]. Penyebab utamanya kembali pada karakteristik GIL yang dibahas pada bagian sebelumnya: dalam pengujian empiris terhadap proyek-proyek Python populer, konkurensi berbasis *thread* jarang menjadi sumber masalah performa yang nyata justru karena GIL secara struktural mencegah lebih dari satu *thread* mengeksekusi *bytecode* secara bersamaan, sehingga beban CPU-*bound* seperti pencarian linear pada dasarnya tetap berjalan semi-

Copyright © SENASIF 2026
ISSN: 2598-0076

sekuensial berapapun jumlah *thread* yang dialokasikan [11]. *Speedup* 1,04x pada 2 *thread* yang sedikit melampaui *baseline* kemungkinan besar hanya mencerminkan *noise* pengukuran dan variasi *scheduling* OS, bukan paralelisme nyata.

Dampak *Oversubscription* dan *Context Switching*

Anomali paling mencolok muncul pada konfigurasi 32 *thread*, di mana waktu eksekusi justru membengkak menjadi 716,13 ms dengan *speedup* merosot ke 0,96x, angka yang lebih buruk dibandingkan eksekusi satu *thread*. Kondisi ini adalah kasus klasik *oversubscription*: jumlah *thread* aktif (32) melampaui dua kali kapasitas *logical processor* yang tersedia (16), sehingga kernel sistem operasi terpaksa melakukan penjadwalan *time-slice* yang lebih agresif. Setiap kali CPU berpindah dari satu *thread* ke *thread* lain, terjadi *context switching* yang membawa dua jenis biaya sekaligus: biaya langsung berupa penyimpanan dan pemulihan *register* serta program *counter*, dan biaya tidak langsung berupa gangguan pada *cache prosesor* yang harus dimuat ulang setelah setiap perpindahan [12]. Struktur *Process Control Block* dan mekanisme setara pada level *thread* menyimpan seluruh konteks eksekusi ini, dan semakin sering perpindahan terjadi, semakin besar proporsi waktu CPU yang habis untuk pekerjaan administratif alih-alih pekerjaan komputasi sesungguhnya [12]. Ditambah lagi, pengujian pada sistem *multicore* lain menunjukkan bahwa efek *cache thrashing* dari perpindahan *thread* yang berlebihan cenderung baru terlihat signifikan begitu jumlah *thread* jauh melampaui titik optimal arsitektur *core*, persis seperti pola degradasi yang teramati pada 32 *thread* dalam penelitian ini [11].

Komparasi dengan Skema *Multiprocessing*

Skema *multiprocessing* menunjukkan pola degradasi yang jauh lebih ekstrem dibandingkan *multithreading*. Tabel 2 merangkum hasil pengujian pada *ProcessPoolExecutor*.

Tabel 2. Ringkasan Hasil Uji *Multiprocessing (ProcessPoolExecutor)*

Jumlah *Process*	Waktu Eksekusi (ms)	*Speedup*
1	2.313,92	1,00x (Baseline)
2	2.487,16	0,93x
4	3.491,35	0,66x
8	6.156,48	0,38x
16	10.941,97	0,21x
32	21.069,38	0,11x

Berbeda dengan *multithreading* yang stagnan di sekitar angka 1,00x, *multiprocessing* justru mengalami penurunan performa yang konsisten dan tajam seiring bertambahnya jumlah *process*, hingga waktu eksekusi membengkak sembilan kali lipat dari *baseline* pada 32 *process*. Pola ini sejalan dengan pengujian kuantitatif terhadap pengaruh jumlah proses pada Linux, yang mendapati

Copyright © SENASIF 2026

ISSN: 2598-0076

bahwa *overhead* pembuatan proses melalui pemanggilan sistem seperti *fork()* tumbuh signifikan seiring bertambahnya jumlah proses aktif, jauh melampaui *overhead* pembuatan *thread* yang jauh lebih ringan [13]. Setiap process baru membawa struktur *Process Control Block* tersendiri lengkap dengan alokasi ruang alamat virtual *(virtual address space)* yang terisolasi penuh dari *process* induk, sehingga biaya inisialisasinya jauh lebih mahal dibanding sekadar menambah entri pada tabel *thread* di dalam satu process [12]. Karena *dataset* 20 juta elemen harus dibagi dan dikirim ke tiap *process* yang terisolasi memorinya, setiap pengiriman data melibatkan proses serialisasi *(pickling)* di sisi pengirim dan deserialisasi di sisi penerima, sebuah langkah yang sepenuhnya dihindari pada skema *threading* karena seluruh *thread* berbagi ruang memori yang sama [14]. Biaya serialisasi ini bertambah proporsional dengan jumlah *process,* dan ditambah dengan latensi *Inter-Process Communication* yang secara arsitektural jauh lebih mahal dibandingkan komunikasi antar-*thread* dalam satu ruang memori, kombinasi keduanya menjelaskan mengapa waktu eksekusi tumbuh hampir eksponensial alih-alih linear seiring bertambahnya jumlah *process* [15].

Implikasi Desain Perangkat Lunak dan Sistem Operasi

Temuan ini menegaskan bahwa pemilihan skema konkurensi tidak bisa dilepaskan dari karakteristik beban kerja. Untuk tugas CPU-*bound* murni seperti pencarian linear pada *dataset* besar, baik *multithreading* maupun *multiprocessing* pada Python gagal memberikan *speedup* yang berarti, meskipun karena alasan yang berbeda: *multithreading* terhambat oleh GIL pada level *interpreter* [11], sementara *multiprocessing* justru terbebani oleh *overhead* struktural pembuatan *process* dan komunikasi antar-*process* yang jauh melampaui manfaat paralelismenya sendiri untuk beban kerja bergranularitas rendah [12]. Kondisi ini berbeda dari kasus beban kerja yang lebih kasar granularitasnya, seperti ekstraksi fitur *audio* per *file,* di mana *overhead* pembagian *task* masih dapat diamortisasi oleh durasi komputasi tiap unit kerja yang cukup besar. Implikasi praktisnya, pengembang perangkat lunak paralel di lingkungan Python perlu mempertimbangkan alternatif di luar *threading* dan *multiprocessing* standar untuk beban CPU-*bound* bergranularitas halus, seperti memanfaatkan *library* C-*extension* yang melepas GIL selama komputasi berat, atau mendesain ulang unit kerja agar granularitasnya cukup besar untuk mengamortisasi biaya pembuatan *process* dan serialisasi data. Dari sisi sistem operasi, hasil ini juga menegaskan pentingnya membatasi jumlah *worker* sesuai kapasitas *logical processor* yang tersedia, karena melampaui batas tersebut hanya menambah beban penjadwalan tanpa kontribusi nyata terhadap *throughput* komputasi.

IV. HASIL DAN PEMBAHASAN

Hasil pengujian empiris pada *dataset* 20 juta data membuktikan bahwa penambahan alokasi *worker* tidak berbanding lurus dengan peningkatan performa pencarian data CPU-*bound,* sehingga hipotesis penelitian ini terkonfirmasi. Pada skema *multithreading, speedup* yang diperoleh sepanjang rentang 1 hingga 16 *thread* hanya bergerak stagnan di kisaran 1,00x sampai 1,04x, jauh dari

Copyright © SENASIF 2026
ISSN: 2598-0076

ekspektasi teoritis pada prosesor berkapasitas 16 *logical processor*, akibat restriksi struktural Global Interpreter Lock yang membatasi eksekusi *bytecode* Python hanya pada satu *thread* dalam satu waktu. Anomali paling nyata muncul pada 32 *thread*, di mana waktu eksekusi justru membengkak dan *speedup* merosot ke 0,96x, sebagai konsekuensi langsung dari *oversubscription* yang memaksa kernel sistem operasi melakukan *context switching* lebih agresif dan memicu perebutan sumber daya prosesor antar-*thread*. Kondisi ini berbanding terbalik dengan skema *multiprocessing*, yang alih-alih menghindari batasan GIL melalui isolasi *interpreter* per *process*, justru mengalami pembengkakan waktu eksekusi yang jauh lebih ekstrem hingga sembilan kali lipat *baseline* pada 32 *process*, disebabkan oleh biaya alokasi *Process Control Block* dan ruang alamat virtual yang terisolasi penuh untuk tiap process baru, ditambah beban serialisasi data serta latensi komunikasi antar-*process* yang tumbuh signifikan seiring bertambahnya jumlah *worker*.

Temuan ini menegaskan bahwa keputusan menambah jumlah *worker* pada aplikasi paralel berbasis Python tidak boleh dilakukan tanpa mempertimbangkan kapasitas *logical core* fisik yang tersedia, karena alokasi yang melampaui batas tersebut hanya menambah *overhead* penjadwalan tanpa kontribusi nyata terhadap *throughput* komputasi. Baik *multithreading* maupun *multiprocessing* standar pada Python terbukti kurang cocok untuk beban kerja CPU-*bound* bergranularitas halus seperti pencarian linear pada *dataset* besar, sehingga pengembang perlu mempertimbangkan pendekatan alternatif, seperti pustaka berbasis ekstensi C yang melepas GIL selama komputasi berat, atau mendesain ulang unit kerja agar granularitasnya cukup besar untuk mengamortisasi biaya pembuatan *process* dan *overhead* serialisasi. Penelitian lanjutan disarankan untuk mengevaluasi karakteristik performa yang serupa pada build Python *free-threaded (nogil)* pasca-PEP 703 yang meniadakan GIL secara opsional, serta menguji skema alokasi *worker* pada arsitektur prosesor heterogen atau *hybrid core*, mengingat perbedaan karakteristik *performance core* dan *efficiency core* berpotensi memunculkan pola *bottleneck* yang berbeda dari temuan pada arsitektur homogen dalam penelitian ini.

Copyright © SENASIF 2026

ISSN: 2598-0076

DAFTAR RUJUKAN

[1] M. Kusbianto, M. A. K. Al Akmal, dan M. A. Yaqin, "Pengaruh Jumlah Thread terhadap Efisiensi CPU pada Sistem Multicore," *Jurnal Pustaka Data*, vol. 5, no. 2, hlm. 412–417, Des 2025, doi: 10.55382/jurnalpustakadata.v5i2.1433.

[2] R. Hadi, "EVALUASI PERFORMA MULTITHREADING PADA PENGOLAHAN LAPORAN PENJUALAN DENGAN APLIKASI DESKTOP," *JUTIK*, vol. 6, no. 3, Okt 2020, doi: 10.36002/jutik.v6i3.1158.

[3] I. K. Wibowo, A. R. A. Besari, dan Muh. R. Rizqullah, "Implementation of Multiprocessing and Multithreading for End Node Middleware Control on Internet of Things Devices," *Inf. J. Ilm. Bid. Teknol. Inf. dan Komun.*, vol. 6, no. 1, hlm. 54–60, Jan 2021, doi: 10.25139/inform.v6i1.3346.

[4] A. Fauzan, A. P. Wicaksono, dan E. A. Pambudi, "Comparative Scalability Analysis of Python Multiprocessing and OpenMP on Windows and WSL2," *SinkrOn*, vol. 10, no. 3, hlm. 1515–1526, Jul 2026, doi: 10.33395/sinkron.v10i3.16262.

[5] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," dalam *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*, Atlantic City, New Jersey: ACM Press, 1967, hlm. 483. doi: 10.1145/1465482.1465560.

[6] Husnul Amisyah dan Agung Hidayah, "Analisis Performa Waktu Eksekusi dan Penggunaan Memori Algoritma Linear Search dan Binary Search pada Sistem Pencarian Buku Perpustakaan Digital," *josh*, vol. 7, no. 4, Jul 2026, doi: 10.47065/josh.v7i4.10067.

[7] M. A. Dermawan, V. A. Nabilla, dan M. A. Yaqin, "Pengaruh Context Switching terhadap Efisiensi CPU pada Sistem Multicore," *Jurnal Pustaka Data*, vol. 5, no. 2, hlm. 431–437, Des 2025, doi: 10.55382/jurnalpustakadata.v5i2.1504.

[8] P. Chyan dan Sean Coonery Sumarta, "Implementasi dan Analisis Kinerja Komputasi Paralel Menggunakan Ray pada Lingkungan Multi-Core untuk Pemrosesan Audio," *Jur. Ris. Kom.*, vol. 13, no. 3, hlm. 899–908, Jun 2026, doi: 10.30865/jurikom.v13i3.9649.

[9] F. M. David, J. C. Carlyle, dan R. H. Campbell, "Context switch overheads for Linux on ARM platforms," dalam *Proceedings of the 2007 workshop on Experimental computer science*, San Diego California: ACM, Jun 2007, hlm. 3. doi: 10.1145/1281700.1281703.

[10] Nurkholik Safrudin, Dzaky Alaudin Malik, dan Elkin Rilvani, "Optimalisasi Manajemen Thread dalam Sistem Operasi untuk Meningkatkan Kinerja Multithreading," *Merkurius*, vol. 3, no. 2, hlm. 01–05, Jan 2025, doi: 10.61132/merkurius.v3i2.670.

[11] S. F. Soliha, "Analisis Tes Flakiness Untuk Machine Learning Menggunakan Python," *TEKNIK*, vol. 3, no. 1, hlm. 55–72, Mei 2023, doi: 10.51903/teknik.v1i1.323.

Copyright © SENASIF 2026
ISSN: 2598-0076

[12] Fadhila Ramadhani dan Muhammad Firdaus, "Analisa Peran Proses Block Control (PBC) terhadap Sistem Operasi pada Windows 10," *Mars*, vol. 3, no. 5, hlm. 179–190, Okt 2025, doi: 10.61132/mars.v3i5.1134.

[13] A. Ekawijana dan B. Noviansyah, "The Effect Of The Number Of Processes On Multiple Processing Performance On Linux," *VCT*, vol. 8, no. 1, hlm. 70–78, Apr 2026, doi: 10.38038/vocatech.v8i1.288.

[14] R. G. Panagiotis, M. Evdoxia, dan D. Minas, "Parallelization of the hierarchical search in Python for high performance embedded systems," dalam *2016 5th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, Thessaloniki, Greece: IEEE, Mei 2016, hlm. 1–4. doi: 10.1109/MOCAST.2016.7495138.

[15] Y. Xia, D. Du, Z. Hua, B. Zang, H. Chen, dan H. Guan, "Boosting Inter-process Communication with Architectural Support," *ACM Trans. Comput. Syst.*, vol. 39, no. 1–4, hlm. 1–35, Nov 2021, doi: 10.1145/3532861.

Copyright © SENASIF 2026
ISSN: 2598-0076