MANTECH Publications
International Journal of Programming Languages and Internet of Things
Volume 2, Issue 1, January-June 2026

Concurrency, Parallelism, and Memory Management in Programming Languages: Models and Trade-offs

Dr. Arnab Chatterjee[1], Sudeshna Banerjee[2], Debasish Mukherjee[3], Ananya Sen[4]

ABSTRACT

Modern software must exploit multiple processor cores, remain responsive, and manage memory correctly, and the way a programming language supports concurrency and manages memory is central to its design; this paper reviews the models and trade-offs. It distinguishes concurrency, the structuring of a program as independently executing tasks, from parallelism, their simultaneous execution, and describes the principal models of concurrency: the sharing of memory among threads, which communicate through shared variables and require synchronisation, with the attendant hazards of data races and deadlock; the passing of messages between tasks with private state, as in the actor and communicating-sequential-process models, which avoid these hazards; and the asynchronous, event-driven model, which provides concurrency without many threads. It describes the management of memory: manual allocation and freeing, which give control at the risk of error; automatic reclamation through reference counting and tracing garbage collection, which give safety at some cost in overhead and predictability; and ownership systems, which provide memory safety without garbage collection at compile time. It describes the interaction of concurrency and memory, through the memory model and the requirement of memory safety for safe concurrency. The study finds that these choices involve trade-offs among performance, safety, and ease of programming, so that languages adopt different combinations suited to their aims, and that recent designs seek to provide safety without sacrificing performance. It concludes that concurrency and memory management are central to language design. Certain figures are illustrative.

Page 1-11 © MANTECH PUBLICATIONS 2026. All Rights Reserved

KEYWORDS: *Concurrency, Parallelism, Threads, Message Passing, Memory Management, Garbage Collection, Ownership, Data Races, Programming Languages*

INTRODUCTION

Two concerns are central to the design and use of modern programming languages: how a program expresses and manages the concurrent and parallel execution of its parts, and how it manages the memory it uses. The first has grown in importance as processors have come to provide many cores rather than ever-faster single cores, so that exploiting their power requires concurrency and parallelism, and as software has come to serve many simultaneous users and to handle much input and output. The second is fundamental to the correctness, safety, and performance of every program. This paper reviews the models of concurrency and the approaches to memory management, and the trade-offs among them [1], [2].

It is useful to distinguish concurrency from parallelism. Concurrency is the structuring of a program as multiple tasks that execute independently and may make progress in overlapping periods; it is a matter of the structure of the program, concerned with dealing with many things at once, and it is valuable for responsiveness and for handling input and output as well as for parallelism. Parallelism is the simultaneous execution of tasks, which requires multiple processors and is concerned with doing many things at once to increase throughput. A concurrent program may or may not run in parallel, and the models of concurrency differ in how tasks are expressed and how they interact [1], [3].

The management of memory is the other central concern, and languages differ in whether the programmer manages memory manually, with control but at the risk of error, or the language manages it automatically, with safety but at some cost. The central purpose of this paper is to review the models of concurrency, the approaches to memory management, and their interaction, and to explain the trade-offs among performance, safety, and ease of programming that they involve. The paper draws on the established literature of programming languages, presents the subject systematically, and notes that certain figures are illustrative.

LITERATURE REVIEW

Page 1-11 © MANTECH PUBLICATIONS 2026. All Rights Reserved

The models of concurrency are described in the standard literature [1], [2], [4]. In the shared-memory model, multiple threads of execution share a common memory and communicate by reading and writing shared variables; because concurrent access to shared data can produce incorrect results, the threads must be synchronised, using locks, semaphores, condition variables, and atomic operations, and the failure to synchronise correctly gives rise to hazards, including data races, in which the outcome depends on the unpredictable timing of access, and deadlock, in which threads wait for one another indefinitely. In the message-passing model, tasks do not share memory but communicate by sending messages, as in the actor model, in which independent actors exchange messages, and the communicating-sequential-processes model, in which processes communicate through channels; because there is no shared state, the hazards of shared-memory concurrency are avoided by design. The asynchronous, event-driven model provides concurrency within a single thread, through an event loop and constructs such as futures and the asynchronous awaiting of results, and is well suited to programs dominated by input and output. The literature describes these models and their properties [3], [5].

The approaches to memory management are treated in the literature [6], [7], [8]. In manual memory management, the programmer explicitly allocates and frees memory, obtaining the greatest control and performance but risking errors, including memory leaks, the use of memory after it has been freed, and freeing it more than once, which are a major source of faults and security vulnerabilities. In automatic memory management, the language reclaims memory no longer in use: reference counting tracks the number of references to each object and reclaims it when the count reaches zero, giving prompt reclamation but requiring the counting to be maintained and failing to reclaim cycles without additional measures; and tracing garbage collection periodically determines which objects are reachable from a set of roots and reclaims the rest, using algorithms such as mark-and-sweep, copying, and generational collection, giving safety at the cost of overhead and of pauses that may be undesirable in time-sensitive programs. The ownership model, exemplified by systems in which each value has a single owner and borrowing is governed by rules checked at compile time, provides memory safety, and freedom from data races, without a garbage collector. The literature compares these approaches and their trade-offs [7], [8].

Page 1-11 © MANTECH PUBLICATIONS 2026. All Rights Reserved

The interaction of concurrency and memory management is treated in the literature [9], [10]. Correct shared-memory concurrency depends on the memory model of the language, which defines when the effects of one thread become visible to another and governs the use of atomic operations and memory ordering; without a well-defined memory model, concurrent programs cannot be reasoned about reliably. Moreover, safe concurrency depends on memory safety, since data races are a form of memory-safety violation, and languages that ensure memory safety differ in whether they also prevent data races: garbage-collected languages prevent the memory-safety errors of manual management but not necessarily data races, whereas ownership systems can prevent both at compile time. The management of memory must itself accommodate concurrency, and garbage collectors may run concurrently with, or must pause, the program's threads. The literature emphasises that concurrency and memory management are intertwined and that their combination is a central concern of language design. These considerations frame the present review [9], [10].

RESEARCH GAP

The models of concurrency and the approaches to memory management are treated thoroughly in specialised texts, but a concise, integrated account that presents them together, explains their interaction, and sets out the trade-offs among performance, safety, and ease of programming that they involve, is useful for orientation and for the informed choice and design of languages. There is therefore value in a consolidated review that connects concurrency and memory management and situates them within language design.

OBJECTIVES

The specific objectives of this study are:

- To distinguish concurrency from parallelism and explain their motivations.
- To describe the shared-memory, message-passing, and asynchronous models of concurrency.
- To describe manual, reference-counted, tracing, and ownership-based memory management.
- To explain the interaction of concurrency and memory through the memory model and safety.
- To set out the trade-offs among performance, safety, and ease of programming.

Page 1-11 © MANTECH PUBLICATIONS 2026. All Rights Reserved

METHODOLOGY

A review-based approach was adopted. The models of concurrency and the approaches to memory management were consolidated from the established literature [1]-[10], and the material was organised into a framework comprising the models of concurrency, the spectrum of memory-management approaches, and the reclamation of memory by reachability. Where figures present the concurrency models, the memory-management spectrum, or garbage collection, these are illustrative and schematic, and the placement of particular languages is indicative rather than exact, since many languages provide several mechanisms.

Concurrency Models and Memory Spectrum

Two models of concurrency are represented in Figure 1, contrasting the sharing of memory among threads, with its synchronisation and hazards, with the passing of messages between tasks with private state. The spectrum of memory-management approaches is represented in Figure 2, ranging from manual control to automatic safety and positioning the principal approaches. These representations organise the subject as set out in the literature [1], [7].

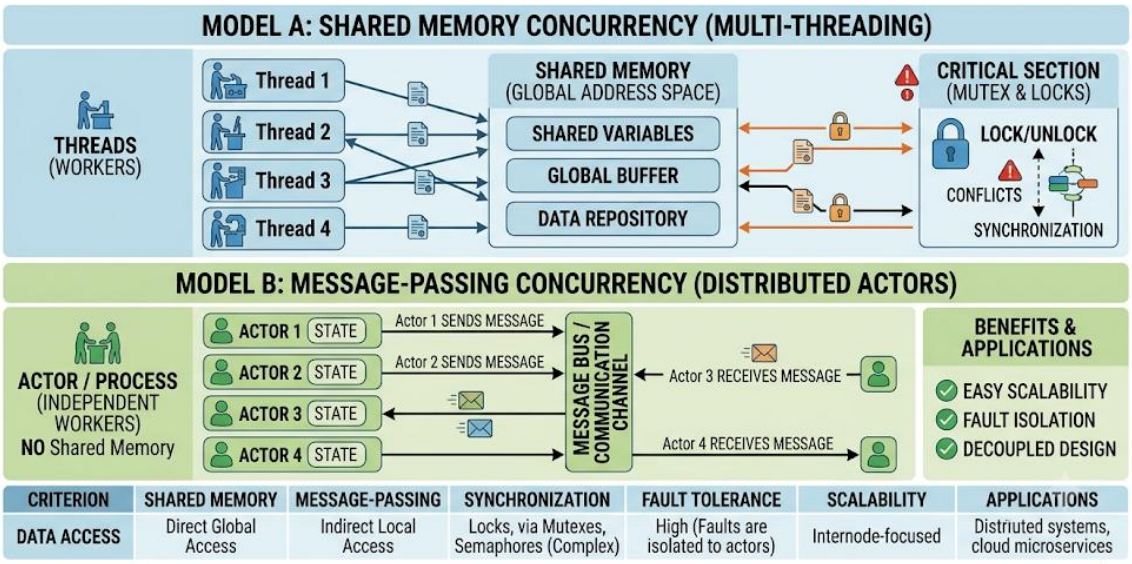


Figure 1: Two models of concurrency

Page 1-11 © MANTECH PUBLICATIONS 2026. All Rights Reserved

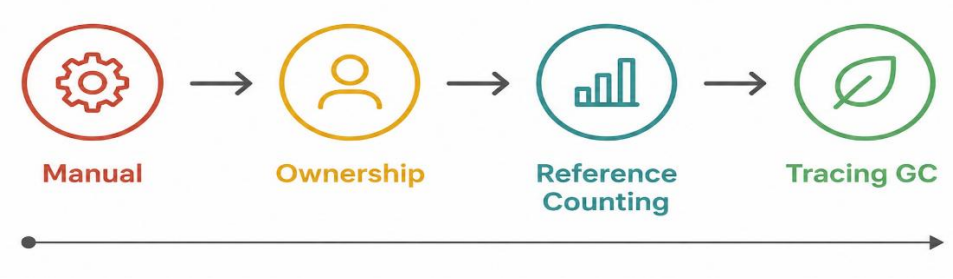


Figure 2: A spectrum of memory-management approaches

Garbage Collection

The reclamation of memory by reachability is represented in Figure 3, showing objects reachable from the roots, which are retained, and unreachable objects, which are reclaimed as garbage. This representation conveys the principle of tracing garbage collection, on which automatic memory management commonly rests [7], [8].

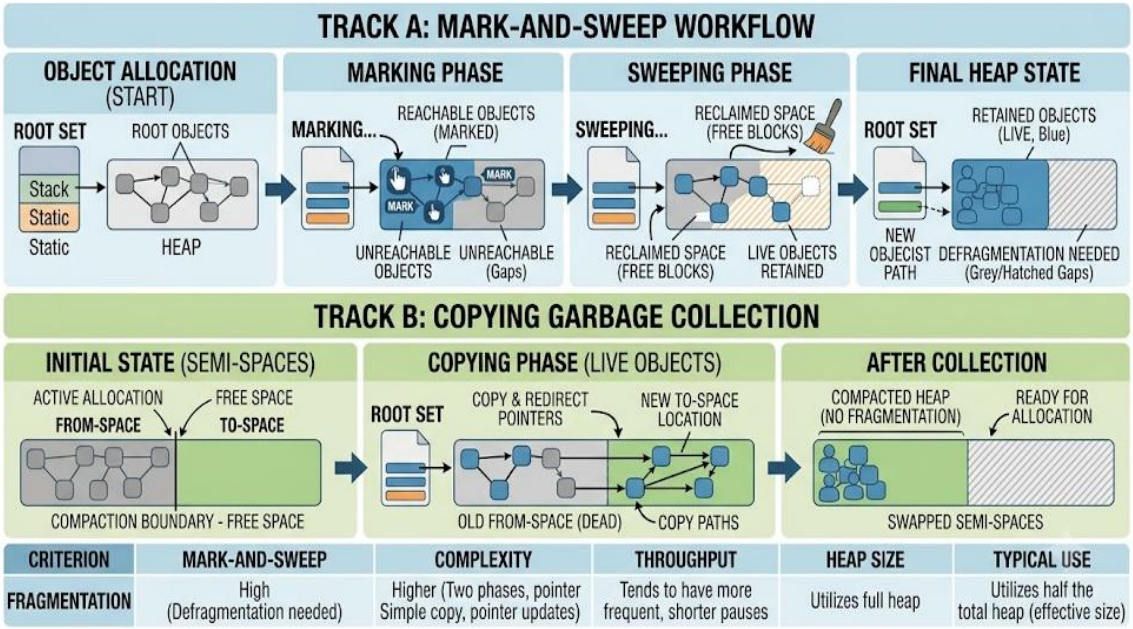


Figure 3: Reachability-based garbage collection

RESULTS AND FINDINGS

Concurrency Models

Table 1 sets out the models of concurrency, and Figure 1 depicts two of them. The findings show that the shared-memory model, in which threads communicate through shared variables, is powerful but requires careful synchronisation and is prone to data races and deadlock; that the message-passing model, in which tasks communicate by messages, avoids these hazards

Page 1-11 © MANTECH PUBLICATIONS 2026. All Rights Reserved

by eliminating shared state; and that the asynchronous model provides concurrency within a single thread for programs dominated by input and output. The finding is that the models differ in their safety and their suitability, the message-passing and asynchronous models avoiding some of the hazards of shared memory, so that the choice of model is a significant one, and that languages increasingly favour models that reduce the risk of concurrency errors [3], [5].

Table 1: Models of concurrency

Model	Central idea	Examples
Shared-memory threads	Threads share variables; synchronise	C/C++, Java threads
Message passing / actors	Private state; communicate by messages	Erlang, Go channels
Asynchronous / event-driven	Concurrency within one thread	JavaScript, asyncio
Data parallelism	Same operation over many data	SIMD, GPU, parallel loops

Memory Management

Table 2 sets out the approaches to memory management, and Figures 2 and 3 depict the spectrum and garbage collection. The findings show that manual management gives control and performance but risks errors; that reference counting gives prompt reclamation but must maintain counts and handle cycles; that tracing garbage collection gives safety but incurs overhead and pauses; and that ownership gives memory safety without a garbage collector at compile time. The finding is that these approaches occupy a spectrum from manual control to automatic safety, and that the choice among them is a trade-off, manual management and ownership favouring performance and control, garbage collection favouring safety and ease, so that the approach is chosen to suit the aims of the language and the program [6], [7], [8].

Table 2: Approaches to memory management.

Approach	How it works	Trade-off
Manual	Programmer allocates and frees	Control vs risk of errors
Reference counting	Free when references reach zero	Prompt; cycles need care

Page 1-11 © MANTECH PUBLICATIONS 2026. All Rights Reserved

Approach	How it works	Trade-off
Tracing garbage collection	Reclaim unreachable objects	Safety vs overhead, pauses
Ownership / borrowing	Compile-time ownership rules	Safety and performance; stricter

Interaction and Hazards

Table 3 sets out the hazards of concurrency, and the findings consider the interaction of concurrency and memory. The findings show that shared-memory concurrency is subject to hazards, including data races, deadlock, livelock, and starvation, and that correct concurrency depends on the memory model and on memory safety, since data races are a memory-safety violation. The finding is that concurrency and memory management are intertwined, that safe concurrency requires memory safety, and that languages differ in the guarantees they provide, garbage-collected languages preventing the errors of manual memory management but not necessarily data races, and ownership systems preventing both, so that the combined treatment of concurrency and memory is central to the safety of a language [9], [10].

Table 3: Hazards of shared-memory concurrency.

Hazard	Description
Data race	Unsynchronised concurrent access with incorrect result
Deadlock	Threads wait for one another indefinitely
Livelock	Threads act but make no progress
Starvation	A thread is perpetually denied a needed resource

DISCUSSION

The principal point of this review is that the way a language supports concurrency and manages memory is central to its design, and that both involve trade-offs among performance, safety, and ease of programming. The models of concurrency differ in how tasks interact and in the hazards they admit: the shared-memory model is powerful and efficient but exposes the programmer to data races and deadlock, while the message-passing and asynchronous models trade some of that power for greater safety and simplicity. The approaches to memory management differ similarly: manual management and ownership offer control and performance, while garbage collection offers safety and ease at some cost in overhead and predictability. These are not questions with single answers but trade-offs that different languages resolve differently [1], [2].

Page 1-11 © MANTECH PUBLICATIONS 2026. All Rights Reserved

The interaction of concurrency and memory management deepens these considerations. Correct concurrent programming depends on a well-defined memory model, which specifies how the actions of one thread become visible to another, and safe concurrency depends on memory safety, since data races are themselves a violation of memory safety. Languages therefore differ not only in their concurrency model and their memory management considered separately, but in how the two combine: a garbage-collected language with shared-memory threads prevents the memory errors of manual management but still admits data races, whereas a language whose ownership system governs both memory and sharing can prevent both classes of error at compile time. The combined treatment of concurrency and memory is thus a defining characteristic of a language's approach to safety [9], [10].

The field continues to evolve in a direction that seeks to reconcile the trade-offs. The recognition of the difficulty and danger of shared-memory concurrency has encouraged the adoption of message-passing and asynchronous models and of language features that make concurrency safer; and the recognition of the costs of both manual management and garbage collection has motivated ownership systems that aim to provide memory safety, and freedom from data races, without the overhead of a collector. These developments do not abolish the trade-offs among performance, safety, and ease, but they shift them, offering combinations, such as safety with performance, that were previously difficult to achieve. Presented together, the models of concurrency, the memory-management spectrum, and garbage collection convey these central concerns as a connected whole. Certain figures are illustrative and the placement of particular languages is indicative rather than exact.

LIMITATIONS

This study is a review of established concepts and presents the models of concurrency and the approaches to memory management in summary form; it provides a conceptual account rather than detailed treatment of particular languages, runtimes, or algorithms. The concurrency models, memory-management spectrum, and garbage collection shown in the figures are illustrative and schematic, and the placement and classification of particular languages are indicative rather than exact, since many languages provide several mechanisms. Readers requiring authoritative and detailed treatment are referred to the standard texts on concurrency, memory management, and programming languages.

Page 1-11 © MANTECH PUBLICATIONS 2026. All Rights Reserved

FUTURE SCOPE

Further study building on this account can proceed in several directions:

- Study of safe concurrency models and language features that prevent data races.
- Investigation of ownership-based memory management and its adoption.
- Analysis of concurrent and low-pause garbage collection for latency-sensitive systems.
- Study of memory models and the reasoning about concurrent programs.
- Examination of data and task parallelism and their expression in languages.
- Consideration of tools for detecting concurrency and memory errors.

Pursuing these directions would deepen the understanding of concurrency and memory management in language design.

CONCLUSION

This paper has reviewed concurrency, parallelism, and memory management in programming languages. It distinguishes concurrency, the structuring of a program as independent tasks, from parallelism, their simultaneous execution; describes the shared-memory model, with its synchronisation and its hazards of data races and deadlock, the message-passing model, which avoids these by eliminating shared state, and the asynchronous model; describes manual memory management, reference counting, tracing garbage collection, and ownership; and describes the interaction of concurrency and memory through the memory model and the requirement of memory safety. The central conclusion is that these choices involve trade-offs among performance, safety, and ease of programming, so that languages adopt different combinations suited to their aims, and that recent designs, favouring safer concurrency models and ownership-based memory management, seek to provide safety without sacrificing performance. Concurrency and memory management are central to language design and to the correctness and performance of software.

REFERENCES

1. M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, 2nd ed. Burlington, MA, USA: Morgan Kaufmann, 2020.
2. M. L. Scott, *Programming Language Pragmatics*, 4th ed. Waltham, MA, USA: Morgan Kaufmann, 2016.
3. R. H. Pike, “Concurrency is not parallelism,” Google I/O / Heroku talk, 2012.
4. C. A. R. Hoare, “Communicating sequential processes,” *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

Page 1-11 © MANTECH PUBLICATIONS 2026. All Rights Reserved

5. J. Armstrong, "A history of Erlang," in *Proc. ACM SIGPLAN HOPL III*, 2007, pp. 1–26.
6. B. Stroustrup, *The C++ Programming Language*, 4th ed. Boston, MA, USA: Addison-Wesley, 2013.
7. R. Jones, A. Hosking, and E. Moss, *The Garbage Collection Handbook*. Boca Raton, FL, USA: CRC Press, 2011.
8. P. R. Wilson, "Uniprocessor garbage collection techniques," in *Proc. Int. Workshop Memory Management*, 1992, pp. 1–42.
9. L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. C-28, no. 9, pp. 690–691, 1979.
10. S. V. Adve and H.-J. Boehm, "Memory models: A case for rethinking parallel languages and hardware," *Commun. ACM*, vol. 53, no. 8, pp. 90–101, 2010.
11. S. Klabnik and C. Nichols, *The Rust Programming Language*. San Francisco, CA, USA: No Starch Press, 2019.
12. D. Lea, *Concurrent Programming in Java*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2000.
13. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ, USA: Wiley, 2018.
14. H.-J. Boehm, "Threads cannot be implemented as a library," in *Proc. ACM PLDI*, 2005, pp. 261–268.
15. G. L. Steele, "Parallel programming and parallel abstractions in Fortress," in *Proc. PACT*, 2005, p. 157.
16. D. Ungar, "Generation scavenging: A non-disruptive high performance storage reclamation algorithm," in *Proc. ACM Softw. Eng. Symp.*, 1984, pp. 157–167.
17. J. Gosling *et al.*, *The Java Language Specification*. Redwood City, CA, USA: Oracle, 2021.
18. R. Pike, "Go at Google: Language design in the service of software engineering," 2012.

***Author for Correspondence**
Dr. Arnab Chatterjee
E-mail: arnab.chatterjee128@gmail.com

[1]Professor, Department of Computer Science and Engineering, Jadavpur University, Kolkata
[2]Student, Department of Information Technology, Jadavpur University, Kolkata
[3]Student, Department of Electronics and Telecommunication Engineering, Jadavpur University, Kolkata
[4]Student, School of Mobile Computing and Communication, Jadavpur University, Kolkata

Received Date: January 04, 2026
Accepted Date: January 26, 2026