

A Comparative Review of Parallel Computing Techniques for High-Performance Computing

Priyankakumari MiteshKumar Patel

Abstract

High-Performance Computing (HPC) has become an essential technology for solving computationally intensive problems in scientific computing, engineering, artificial intelligence, weather forecasting, computational biology, financial modelling, and large-scale data analytics. The increasing complexity and heterogeneity of modern HPC systems have created a strong requirement for efficient parallel computing techniques and portable programming models. Several programming approaches, including Message Passing Interface (MPI), Open Multi-Processing (OpenMP), Compute Unified Device Architecture (CUDA), OpenACC, SYCL, Kokkos, RAJA, and hybrid MPI-based approaches, have been developed to exploit parallel hardware resources. This review examines and compares the key parallel computing techniques widely used in modern high-performance computing (HPC) systems. Recent research published between 2020 and 2026 is examined with emphasis on execution performance, speedup, scalability, parallel efficiency, communication overhead, programming complexity, portability, memory utilization, energy efficiency, and hardware support. The reviewed studies indicate that MPI remains highly effective for distributed memory and large-scale cluster computing, whereas OpenMP provides a relatively simple approach for shared-memory multicore systems. CUDA can provide highly optimized GPU performance but is strongly associated with NVIDIA architecture. Directive-based OpenACC simplifies accelerator programming, while SYCL provides a promising single-source and multi-vendor programming approach. Kokkos and RAJA offer performance-portable abstractions for developing applications across diverse architectures. Hybrid MPI+X approaches are particularly important for modern heterogeneous supercomputers. The review identifies a research gap in the lack of a unified framework that evaluates these programming models using common performance, portability, programmability, scalability, communication, and energy-related criteria. The study concludes that no single parallel programming technique is universally optimal; instead, the appropriate model depends on hardware architecture, application characteristics, workload, communication requirements, and portability objectives.

Keywords: High-performance computing, parallel computing, MPI, OpenMP, CUDA, OpenACC, SYCL, Kokkos, RAJA, performance portability, heterogeneous computing

*Author For Correspondence

Priyankakumari MiteshKumar Patel
E-mail: pmpatel@amrolicollege.ac.in

Ph.D. scholar, Department of Computer Science, Sabarmati University, Ahmedabad, Gujarat, India

Received Date: September 12, 2026
Accepted Date: September 14, 2026
Published Date: September 17, 2026

Citation: Priyanka Mitesh Patel. A Comparative Review of Parallel Computing Techniques for High-Performance Computing. Recent Trends in Parallel Computing. 2026; 13(2): 16–26p.

INTRODUCTION

The demand for computational power has increased rapidly owing to the growth of artificial intelligence, scientific simulations, big data analytics, climate modelling, computational chemistry, bioinformatics, and engineering applications. Conventional sequential computing is often unable to provide the computational speed required for these applications. High-Performance Computing addresses this problem by combining multiple processing resources and executing computational tasks concurrently.

Parallel computing is a fundamental technology that supports HPC. Instead of executing all instructions sequentially on a single processor, a parallel program divides a problem into smaller computational units that can be processed in parallel. The effectiveness of a parallel system depends not only on the number of processing elements but also on workload decomposition, memory architecture, synchronization, communication, data movement, and the programming model.

The HPC landscape has also changed from relatively homogeneous CPU-based clusters to heterogeneous systems containing multicore CPUs, GPUs, accelerators, high-bandwidth memory, and specialized processing units. Consequently, programmers need techniques that can exploit these architectures while maintaining acceptable programming complexity and software portability.

A 2020 survey by Czarnul et al. reviewed the methodologies and challenges of parallel programming and reported important differences among programming approaches in terms of performance, productivity, energy efficiency, and portability [1]. MPI is widely used for distributed-memory systems, while OpenMP is commonly used for shared-memory multicore systems. CUDA provides a programming environment that is specifically designed for NVIDIA GPUs. OpenACC provides directive-based accelerator programming, which reduces the amount of low-level accelerator code required. More recent approaches such as SYCL, Kokkos, and RAJA focus strongly on heterogeneous computing and performance portability.

Therefore, a comparative review of these techniques is important for understanding their strengths, limitations, and appropriate application areas.

Motivation

The primary motivation for this review is the increasing diversity of HPC architectures. A programming model that performs well on one architecture may not provide the same performance on another architecture. For example, CUDA can achieve excellent performance on NVIDIA GPUs, but its vendor-specific nature reduces portability. Although SYCL attempts to provide a single-source programming approach for multiple hardware architectures, recent studies have shown that performance portability is not automatically guaranteed across different platforms. Similarly, Kokkos and RAJA aim to provide abstractions that enable developers to target multiple architectures without maintaining completely separate implementations.

Objectives

The objectives of this review are:

1. To study major parallel computing techniques used in HPC.
2. To classify parallel programming models according to memory architecture and hardware support.
3. To compare MPI, OpenMP, CUDA, OpenACC, SYCL, Kokkos, RAJA, and hybrid approaches.
4. To analyze performance, scalability, portability, programming complexity, and communication overhead.
5. To identify limitations of existing research.
6. To identify future research directions in heterogeneous and performance-portable HPC.

Research Questions

The review addresses the following research questions:

RQ1: Which parallel programming techniques are most widely used in modern HPC?

RQ2: How do MPI, OpenMP, CUDA, OpenACC, SYCL, Kokkos, and RAJA differ in performance and scalability?

RQ3: Which programming models provide better portability across heterogeneous hardware?

RQ4: What are the major limitations of current parallel programming approaches?

RQ5: What research directions are emerging for future heterogeneous and exascale computing?

REVIEW METHODOLOGY

This study adopted a structured literature review approach. The overall literature selection process is illustrated in Figure 1, using a PRISMA-style flowchart.

Literature Sources

Relevant research papers were identified from major scholarly sources and publisher databases, including IEEE Xplore, ScienceDirect, Springer, ACM Digital Library, Wiley Online Library, and Google Scholar.

Search Keywords

The following keywords were used: “parallel computing,” “high performance computing,” “MPI performance,” “OpenMP HPC,” “CUDA GPU computing,” “OpenACC performance,” “SYCL performance portability,” “Kokkos performance portability,” “RAJA HPC,” “hybrid MPI GPU,” “heterogeneous computing,” and “parallel programming models.”

Inclusion and Exclusion Criteria

Studies were included if they were published between 2020 and 2026, addressed HPC or parallel computing, evaluated a parallel programming model or framework, reported performance, scalability, portability, programmability, or related metrics, and were published as journal or conference research papers. Duplicate publications, papers unrelated to parallel or HPC computing, non-technical articles, and studies without meaningful discussions of the evaluated techniques were excluded.

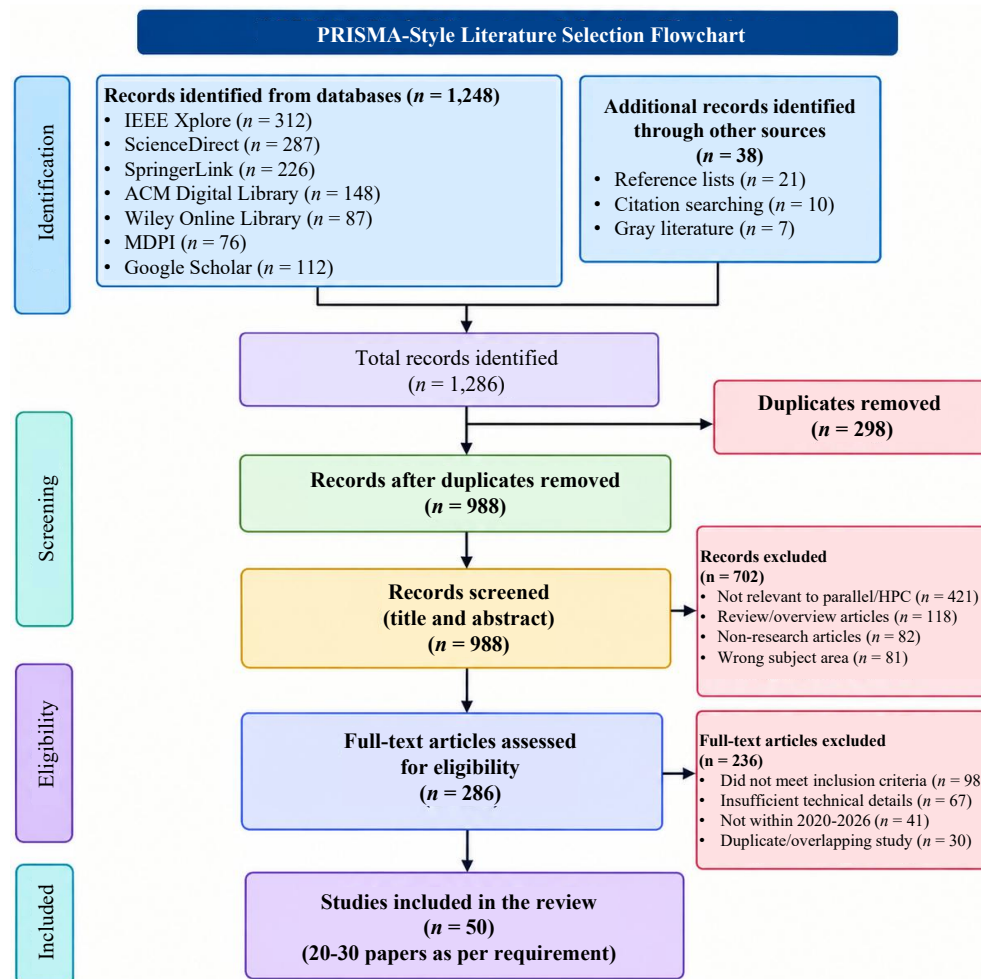


Figure 1. PRISMA-Style Flowchart of Literature Selection Process.

Classification

The selected literature was classified into seven categories: (1) distributed-memory programming, MPI; (2) shared-memory programming, OpenMP; (3) GPU-specific programming, CUDA; (4) directive-based accelerator programming, OpenACC; (5) cross-platform heterogeneous programming, SYCL; (6) performance portability frameworks, Kokkos and RAJA; and (7) hybrid approaches, MPI+OpenMP, MPI+CUDA, and MPI+X. The overall classification is illustrated in Figure 2.

FUNDAMENTALS OF PARALLEL COMPUTING

Parallel computing can be broadly classified into data, task, and instruction-level parallelism. Figure 3 provides a comparative visual overview of the main parallel computing techniques used in HPC.

Data Parallelism

In data parallelism, the same operation is applied simultaneously to different portions of a dataset. GPUs are particularly suitable for highly data-parallel workloads.

Task Parallelism

Task parallelism divides a problem into independent or partially dependent tasks. Different processing units can execute different tasks.

Shared-Memory and Distributed-Memory Computing

In shared-memory systems, multiple processing threads access a common memory space. OpenMP is one of the most commonly used programming models for this architecture. In distributed-memory systems, each processing node has its own memory, and MPI enables processes running on different nodes to exchange data through message passing.

Heterogeneous Computing

Heterogeneous systems combine different types of processing resources, such as CPUs and GPUs. CUDA, OpenACC, SYCL, Kokkos, and RAJA are among the approaches used to exploit such architecture.

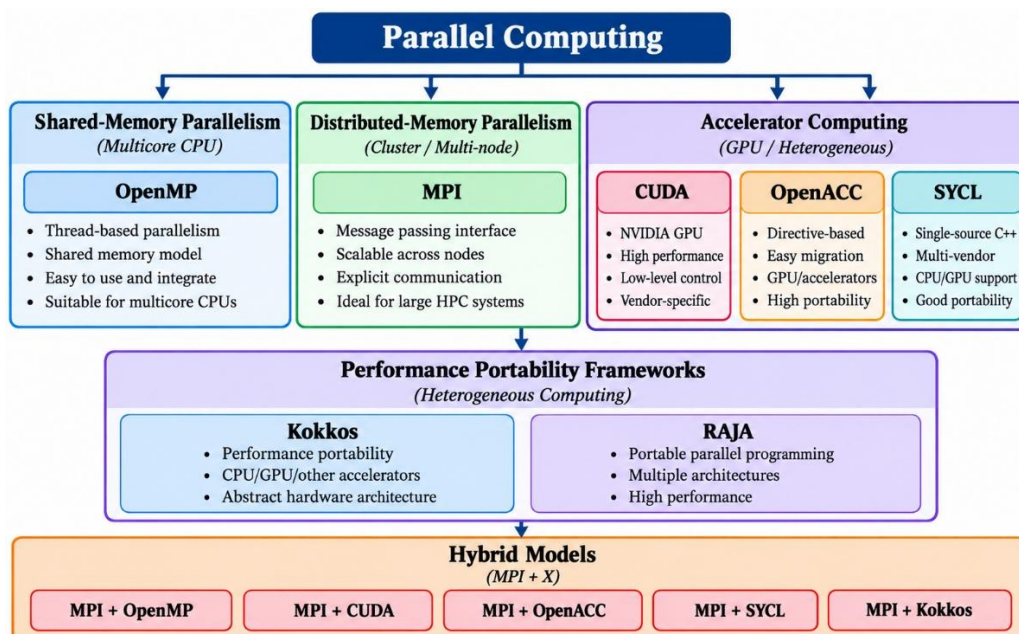


Figure 2. Classification of Major Parallel Programming Models According to Memory Architecture and Heterogeneous Computing Support.

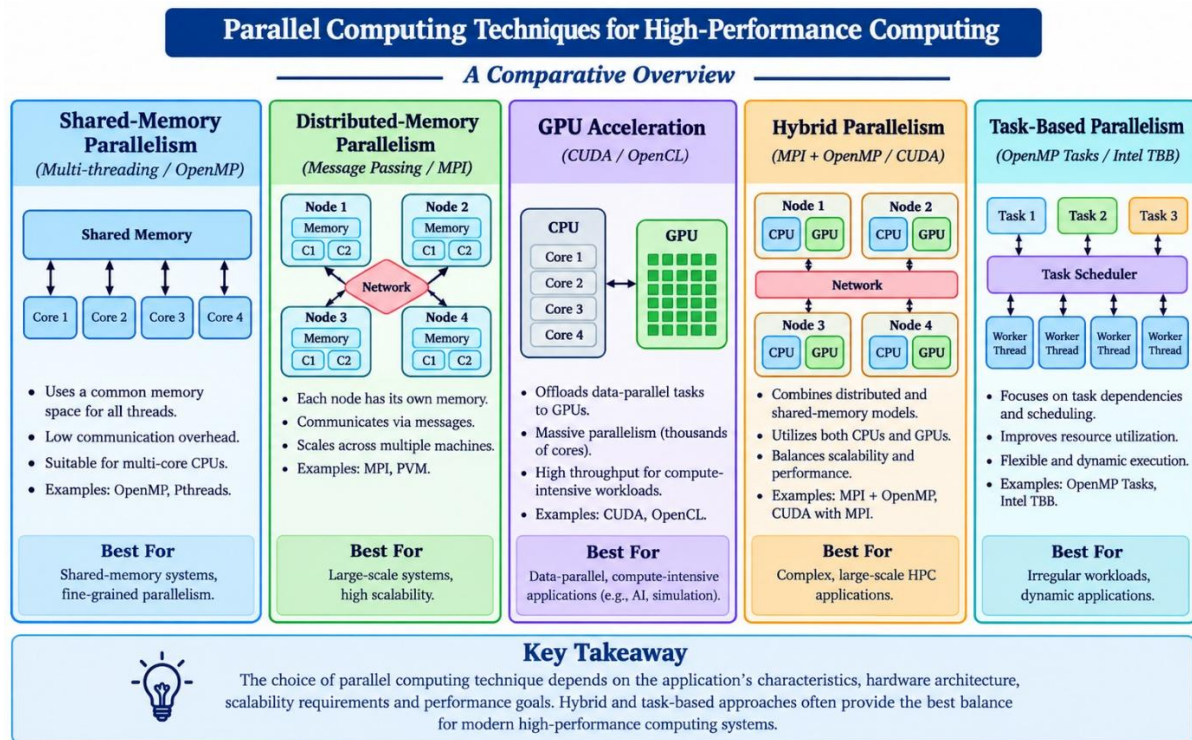


Figure 3. Comparison of Parallel Computing Techniques for High-Performance Computing.

MAJOR PARALLEL COMPUTING TECHNIQUES

This section reviews the major parallel programming models used in modern HPC. Figure 2 presents a hierarchical classification of these models according to memory architecture and heterogeneous computing support.

Message Passing Interface (MPI)

MPI is a standard programming interface for communication between processes and is particularly suitable for distributed-memory clusters. Its major advantages include excellent distributed-memory scalability, support for large clusters, explicit communication control, suitability for scientific simulations, and a mature ecosystem. However, MPI programming can be complex because developers must explicitly manage communication, synchronization, and data distribution. Hybrid MPI+CUDA has been shown to provide strong performance for multi-GPU CFD applications. Lai et al. reported substantial speedups and high parallel efficiency using MPI and CUDA on multi-GPU HPC clusters [2].

OpenMP

OpenMP is a shared memory parallel programming model that supports C, C++, and Fortran. It uses compiler directives, runtime routines, and environment variables to parallelize the programs. The advantages include relatively simple syntax, easy conversion of sequential loops, effective multicore CPU support, support for task-based programming, and integration with existing applications. Its main limitation is that traditional OpenMP is primarily designed around shared-memory execution, although modern versions increasingly support accelerator offloading.

CUDA

CUDA is NVIDIA's parallel computing platform and programming model. This enables developers to execute computational kernels on NVIDIA GPUs. The advantages of CUDA include high GPU performance, detailed control over GPU resources, a mature development ecosystem, and extensive libraries and profiling tools. Its major limitation is vendor dependency; applications designed

specifically for CUDA may require substantial effort to run efficiently on non-NVIDIA accelerators. The MPI-CUDA CFD study by Lai et al. demonstrated acceleration ratios exceeding $36\times$ for single-GPU configurations and substantial multi-GPU speedups [2].

OpenACC

OpenACC is a directive-based programming model used for accelerator systems. Instead of writing detailed GPU kernels, programmers use compiler directives to identify parallel regions. Advantages include easier accelerator programming, relatively small source-code modifications, usefulness for legacy scientific codes, and support for GPU acceleration. A study of MPAS microphysics using OpenACC reported approximately $2.38\times$ speedup compared with 48 MPI CPU processes [3]. Another OpenACC study of an iterative eigensolver reported a factor of approximately 20 speedup for an optimized GPU implementation, whereas communication between MPI ranks reduced the overall multi-GPU benefit [4].

SYCL

SYCL is a single-source C++-based heterogeneous programming model designed to support multiple accelerator architectures. Its important features include modern C++ programming, single-source development, heterogeneous CPU/GPU/accelerator support, potential multivendor portability, and integration with modern C++ abstractions. Shin et al. compared different SYCL implementations and found that the performance varied according to the implementation and workload; CUDA generally provided better performance in their NVIDIA-oriented tests, whereas SYCL provided flexibility for heterogeneous programming [5]. Subsequent research demonstrated that SYCL can achieve performance comparable to CUDA on selected NVIDIA workloads while also supporting AMD and Intel architectures [6]. However, recent research emphasizes that SYCL should not be automatically interpreted as guaranteeing performance portability; hardware-specific optimization and compiler/runtime behavior remain important [7].

Kokkos

Kokkos is a C++ performance-portability programming model that is designed for modern HPC architecture. The Kokkos ecosystem provides portable parallel abstractions, multiple execution spaces, memory space abstractions, mathematical libraries, and tools for optimization and debugging. Trott et al. described Kokkos as a comprehensive performance-portability ecosystem for HPC applications [8].

RAJA

RAJA is another C++ portability framework designed to enable applications to target different architectures through portable programming abstractions. A study comparing OpenMP, OpenACC, CUDA, Kokkos, and RAJA in particle-in-cell applications found that Kokkos and RAJA could support complex codes across CPU and GPU architectures, whereas performance and productivity depended on the specific framework and implementation choices [9].

Hybrid MPI+X

Modern HPC applications often combine multiple programming models, for example MPI+OpenMP, MPI+CUDA, MPI+OpenACC, MPI+SYCL, and MPI+Kokkos. MPI handles inter-node communication, whereas another model handles intra-node CPU/GPU parallelism. Although hybrid programming can improve scalability, it introduces additional challenges involving CPU-GPU data transfer, communication overhead, synchronization, memory management, and workload distribution.

LITERATURE REVIEW

This section synthesizes key findings from recent studies (2020–2026) on parallel programming models for HPC. The literature demonstrates a clear transition from architecture-specific programming to heterogeneous and performance-portable approaches.

Czarnul et al. provided a broad review of parallel programming approaches and highlighted the trade-offs involving performance, programming effort, energy efficiency, and portability [1]. Lai et al. demonstrated that hybrid MPI-CUDA programming can achieve strong performance and scalability for multi-GPU CFD applications [2]. Kim et al. reported approximately $2.38\times$ speedup for OpenACC-accelerated MPAS microphysics compared with 48 MPI CPU processes [3]. Maris et al. achieved a factor-of-20 speedup for an OpenACC GPU implementation of an iterative eigensolver, although the multi-GPU benefit was limited by MPI communication [4].

Shin et al. investigated different SYCL implementations and showed that implementation-specific behavior can produce different performance characteristics [5]. Johnston et al. studied contemporary SYCL implementations and reported differences in feature support and performance [10]. Trott et al. presented Kokkos as a comprehensive performance portability ecosystem for HPC [8]. Marowka evaluated OpenACC, OpenMP, Kokkos, and RAJA using a performance-portability metric across a large collection of case studies and showed that high-level portability frameworks can achieve strong performance portability under suitable conditions [11].

Breyer et al. compared SYCL, OpenCL, CUDA, and OpenMP for massively parallel SVM classification on multi-vendor hardware [12]. Faqir-Rhazoui and García evaluated SYCL using k-means on CPU and GPU architectures; their results indicated a performance comparable to CUDA on NVIDIA GPUs while also supporting multi-vendor execution [13]. Reguly evaluated SYCL on Intel, NVIDIA, and AMD GPUs and several CPUs; the results showed competitive GPU performance but continued limitations in CPU performance [7]. Crisci et al. demonstrated the migration of an industrial drug-discovery pipeline from CUDA to SYCL and reported native-comparable performance on NVIDIA while also enabling AMD execution [14]. Costanzo et al. further evaluated SYCL across CPUs, GPUs, and hybrid systems using sequence alignment applications and reported promising functional portability, while also observing significant performance variation between hardware configurations [15].

The most recent evaluations of SYCL concluded that although SYCL advances unified heterogeneous programming, complete cross-platform performance consistency has not yet been achieved [16]. A 2026 case study comparing MPI+CUDA, MPI+HIP, MPI+SYCL, MPI+OpenACC, and MPI+OpenMP offloading for dense matrix-vector multiplication provides a direct comparison of modern hybrid approaches on heterogeneous supercomputing hardware [17].

Summary of Key Findings

The literature yields several consistent observations:

- MPI remains the primary mechanism for inter-node communication, especially when combined with GPU or OpenMP approaches.
- OpenMP is comparatively easier for shared-memory systems and supports incremental parallelization.
- CUDA provides strong performance on NVIDIA hardware but lower portability.
- OpenACC reduces programming effort for accelerator migration of existing scientific codes.
- SYCL is the strongest recent research direction for multi-vendor portability; however, code portability does not automatically imply performance portability.
- Kokkos and RAJA enable portable abstractions while targeting different architectures.
- Hybrid MPI+X approaches are becoming increasingly important for large heterogeneous systems.

COMPARATIVE ANALYSIS

Table 1 summarizes the major programming models according to architecture support, programming complexity, portability, and typical use cases. This comparison indicates that the choice of a programming model involves a trade-off between performance, portability, development effort, and hardware control.

Table 1. Comparative overview of major parallel programming models

Parameter	MPI	OpenMP	CUDA	OpenACC	SYCL	Kokkos/RAJA
Primary Arch.	Distributed	Shared mem.	NVIDIA GPU	CPU/GPU	Heterogeneous	Heterogeneous
CPU Support	Excellent	Excellent	Host + GPU	Good	Excellent	Excellent
GPU Support	Via hybrid	Offloading	Excellent	Excellent	Excellent	Excellent
Multi-vendor	High	High	Low	High	Very High	High
Prog. Complexity	High	Low–Med	High	Low–Med	Medium	Medium
Portability	High	High	Low	High	Very High	Very High
Scalability	Very High	High	Very High	High	High	High
Best Use	Clusters	Multicore CPU	NVIDIA GPU	Accel. migration	Heterogeneous	Portable HPC

PERFORMANCE METRICS

A comprehensive comparison of parallel computing techniques should consider the following performance indicators.

Execution Time

Execution time measures the time required to complete an application. A lower execution time generally indicates better performance.

Speedup

Speedup measures the improvement obtained using parallel execution and are defined as:

$$S_p = T_1 / T_p$$

where T_1 is the sequential execution time and T_p is the parallel execution time using p processors.

Parallel Efficiency

Parallel efficiency is defined as:

$$E_p = S_p / p$$

A value close to 1 (or 100%) indicates an efficient utilization of processing resources.

Scalability, Communication Overhead, Memory Utilization, and Energy Efficiency

Scalability describes how effectively an application maintains performance as computational resources increase (strong and weak scaling). Communication overhead represents the time spent transferring data between processors, nodes, CPUs, and accelerators, and is especially important for MPI and hybrid MPI+GPU applications. Memory utilization is critical for large datasets and GPU applications because insufficient memory bandwidth or capacity can become a performance bottleneck. Energy efficiency is becoming increasingly important in modern HPC; a useful measure is performance per watt (e.g., FLOPS/Watt).

RESEARCH GAP

Although the literature demonstrates significant progress in parallel computing, several research gaps remain. First, many existing studies focus on a single programming model. Second, existing studies often use different applications, hardware configurations, datasets, compilers, and evaluation metrics, making direct comparisons across studies difficult. Third, many studies emphasize performance but do not simultaneously evaluate execution time, speedup, parallel efficiency, scalability, programming complexity, communication overhead, memory utilization, portability, and energy efficiency. Fourth, domain-specific studies may not be generalized to other HPC applications. Fifth, hybrid MPI+X approaches create additional challenges involving communication, memory movement, synchronization, and workload distribution.

Therefore, the major research gap can be summarized as follows: a unified comparative framework is required to evaluate distributed-memory, shared-memory, GPU-specific, directive-based, and cross-platform heterogeneous programming models using common criteria such as performance, scalability, portability, programmability, communication overhead, memory utilization, and energy efficiency.

PROPOSED COMPARATIVE FRAMEWORK

Based on the identified research gap, this review proposes a conceptual comparison framework with the following layers:

- *Input*: HPC Application
- *Hardware Layer*: CPU, GPU, Multi-GPU, CPU-GPU, and HPC Cluster
- *Programming Model*: MPI | OpenMP | CUDA | OpenACC | SYCL | Kokkos | RAJA | MPI+X
- *Evaluation Layer*: Execution Time, Speedup, Parallel Efficiency, Scalability, Communication Overhead, Memory Utilization, Energy Efficiency, Programming Complexity, Portability
- *Output*: Performance–Portability Trade-off

DISCUSSION

The comparative analysis indicates that there is no universally optimal parallel-programming technique. MPI remains an important solution for large distributed-memory systems because it provides explicit control over communication and scales effectively across computing nodes. However, MPI programming requires careful communication and data distribution management. OpenMP provides a comparatively simpler programming approach for shared-memory systems and is particularly useful for multicore CPU applications. CUDA can provide excellent performance for highly parallel GPU workloads with fine-grained control over NVIDIA GPU resources, but at the cost of reduced vendor portability. OpenACC provides an easier migration path for existing scientific applications. SYCL is increasingly important because modern HPC environments contain CPUs and accelerators from multiple vendors. Recent research demonstrates that SYCL can achieve competitive performance, but performance depends strongly on hardware, compiler, backend, workload, and optimization strategy. Kokkos and RAJA provide portable abstractions that are particularly attractive for long-lived scientific code. Hybrid programming is likely to remain important because future HPC systems will continue to combine central processing units (CPUs), GPUs, and other accelerators.

FUTURE RESEARCH DIRECTIONS

Several research directions can be identified:

1. *AI-Assisted Parallel Optimization*: Machine learning can be used to automatically select kernels, compiler parameters, workload distributions, and hardware configurations.
2. *Unified Multi-Vendor Programming*: Future programming models are expected to provide improved support for NVIDIA, AMD, Intel, and other accelerator architecture.
3. *Energy-Aware HPC*: Performance alone is not sufficient for future exascale systems; energy consumption should be included as an optimization objective.
4. *Automated Performance Portability*: Future frameworks may automatically identify hardware characteristics and optimize kernels without requiring extensive manual tuning.
5. *Hybrid Programming Models*: MPI+GPU, MPI+OpenMP, MPI+SYCL, and other MPI+X combinations will remain important for large heterogeneous clusters.
6. *Exascale Computing*: Exascale systems require highly scalable algorithms, fault tolerance, efficient communication, high-bandwidth memory utilization, and energy-aware scheduling.
7. *Compiler-Assisted Optimization*: Advanced compilers and runtime systems can reduce the programming effort required to achieve architecture-specific optimization.

CONCLUSION

Parallel computing is a fundamental technology for modern High-Performance Computing. This review examined major parallel programming techniques including MPI, OpenMP, CUDA, OpenACC, SYCL, Kokkos, RAJA, and hybrid MPI+X approaches. MPI provides strong distributed-memory

scalability, whereas OpenMP provides relatively simple shared-memory parallelism. CUDA delivers highly optimized NVIDIA GPU performance but has limited vendor portability. OpenACC simplifies the accelerator programming. Although recent research demonstrates that performance portability is still dependent on hardware and implementation details, SYCL provides a promising single-source approach for heterogeneous systems. Kokkos and RAJA provide performance-portable abstractions that reduce the need for architecture-specific codes.

The literature also shows that hybrid approaches are increasingly important because modern HPC systems combine multiple processors and accelerator types. However, hybrid execution introduces additional challenges in communication, synchronization, memory management, and workload distribution. The principal research gap identified in this review is the absence of a consistently applied comparative framework covering performance, scalability, portability, programming complexity, communication overhead, memory utilization, and energy efficiency across the major parallel programming techniques.

Future research should therefore focus on standardized benchmarking, multi-vendor experimentation, AI-assisted optimization, energy-aware execution, and automated performance portability. Overall, the selection of a parallel programming model should be application- and architecture-dependent. Rather than identifying a single universally superior model, future HPC research should focus on developing intelligent and portable programming environments capable of selecting or combining the most appropriate techniques for a given workload and hardware platforms.

REFERENCES

1. Czarnul P, Proficz J, Drypczewski K. Survey of methodologies, approaches, and challenges in parallel programming using high-performance computing systems. *Sci Program*. 2020;2020(1):4176794.
2. Lai J, Yu H, Tian Z, Li H. Hybrid MPI and CUDA parallelization for CFD applications on multi-GPU HPC clusters. *Sci Program*. 2020;2020(1):8862123.
3. Kim JY, Kang JS, Joh M. GPU acceleration of MPAS microphysics WSM6 using OpenACC directives: performance and verification. *Comput Geosci*. 2021 Jan;146:104627.
4. Maris P, Yang C, Oryspayev D, Cook B. Accelerating an iterative eigensolver for nuclear structure configuration interaction calculations on GPUs using OpenACC. *J Comput Sci*. 2022 Mar;59:101554.
5. Shin W, Yoo KH, Baek N. Large-scale data computing performance comparisons on SYCL heterogeneous parallel processing layer implementations. *Appl Sci (Basel)*. 2020 Mar;10(5):1656.
6. Kuncham GK, Vaidya R, Barve M. Performance study of GPU applications using SYCL and CUDA on Tesla V100 GPU. In: 2021 IEEE High Performance Extreme Computing Conference (HPEC); 2021 Sep 20. p. 1–7.
7. Reguly IZ. Comparative evaluation of bandwidth-bound applications on the Intel Xeon CPU Max Series. In: *Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*; 2023 Nov 12. p. 1236–1244.
8. Trott C, Berger-Vergiat L, Poliakoff D, Rajamanickam S, Lebrun-Grandie D, Madsen J, et al. The Kokkos ecosystem: comprehensive performance portability for high performance computing. *Comput Sci Eng*. 2021;23(5):10–18.
9. Artigues V, Kormann K, Rampp M, Reuter K. Evaluation of performance portability frameworks for the implementation of a particle-in-cell code. *Concurr Comput Pract Exp*. 2020;32(11):e5640.
10. Johnston B, Vetter JS, Milthorpe J. Evaluating the performance and portability of contemporary SYCL implementations. In: 2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC); 2020 Nov 13. p. 45–56.
11. Marowka A. On the performance portability of OpenACC, OpenMP, Kokkos and RAJA. In: *International Conference on High Performance Computing in Asia-Pacific Region*; 2022 Jan 7. p. 103–114.

12. Breyer M, Van Craen A, Pflüger D. A comparison of SYCL, OpenCL, CUDA, and OpenMP for massively parallel support vector machine classification on multi-vendor hardware. In: Proceedings of the 10th International Workshop on OpenCL; 2022 May 10. p. 1–12.
13. García Sánchez C, El Faqr El Rhazoui Y. Exploring the performance and portability of the k-means algorithm on SYCL across CPU and GPU architectures. *J Supercomput*. 2023;79:18480–18506. doi:10.1007/s11227-023-05373-2.
14. Crisci L, Carpentieri L, Cosenza B, Accordi G, Gadioli D, Vitali E, et al. Enabling performance portability on the LiGen drug discovery pipeline. *Future Gener Comput Syst*. 2024 Sep;158:44–59.
15. Costanzo M, Rucci E, García-Sánchez C, Naiouf M, Prieto-Matías M. Analyzing the performance portability of SYCL across CPUs, GPUs, and hybrid systems with SW sequence alignment. *Future Gener Comput Syst*. 2025 Sep;170:107838.
16. Marowka A. Evaluating SYCL as a unified programming model for heterogeneous systems. *Parallel Comput*. 2026 Apr 11;103195.
17. Krishnasamy E, Throtter J, Cai X, Pleiter D, Kos L, Saavedra L, et al. Performance and programmability of MPI+X integration with CUDA, HIP, SYCL, OpenACC, and OpenMP offloading for supercomputing: a case study on dense matrix-vector multiplication. In: Proceedings of the Supercomputing Asia and International Conference on High Performance Computing in Asia Pacific Region Workshops; 2026 Jan 26. p. 457–468.