# A Comparative Analysis of E4S Programming Models: User Experience, Performance Portability, and Strategic Selection for Scientific Software

## I. Executive Summary: A Framework for Selecting Parallel Programming Models

The transition to heterogeneous, accelerator-based high-performance computing (HPC) systems has fragmented the programming landscape. For scientific software teams, selecting the right parallel programming model is a strategic decision with multi-year implications for performance, portability, and developer productivity. This report provides a comparative analysis of key E4S-related programming models—Kokkos, OpenMP, OpenACC, MPI (MPICH/OpenMPI), UPC++, and parallel C++ features—based on documented user experiences.

The analysis reveals a landscape defined by trade-offs:

* **Investment vs. Return:** C++ library-based portability, exemplified by Kokkos, offers the highest potential for true, single-source performance portability across all vendor hardware. This, however, requires a high-cost strategic investment, often measured in multiple man-years of developer effort for a full application port.1 This contrasts sharply with the low-cost, "easy on-ramp" of directive-based models.2
* **Ecosystem vs. Niche Performance:** For inter-node communication, the MPI standard (MPICH, OpenMPI) provides an unmatched ecosystem of mature, sophisticated tools for debugging and performance tuning.3 This mature, general-purpose model is challenged by the Partitioned Global Address Space (PGAS) model of UPC++, which demonstrates superior performance and programmer productivity for specific *irregular* problem classes (e.g., graph analytics, sparse solvers).4
* **Control vs. Simplicity:** At the on-node level, a clear philosophical divide exists. "Prescriptive" models like OpenMP target offload and library-based models like Kokkos give the expert developer fine-grained control, but require that control to achieve performance. "Descriptive" models like OpenACC prioritize simplicity, empowering the compiler to make optimization decisions.6 A third path, C++ Standard Parallelism (std::par), offers a "zero-effort" model that cedes control entirely to the compiler and runtime in exchange for 100% standard, non-intrusive code.7
* **Maturity vs. Standardization:** The directive-based space is defined by a conflict between the mature, high-performance OpenACC standard and the vendor-agnostic OpenMP target offload standard. Users report that OpenACC is easier to use and more robust, but that ecosystem support is shifting to OpenMP. This forces developers into a "painful" transition, working with "immature" OpenMP compiler implementations whose performance can degrade with application complexity.2

This report details the user experience, performance characteristics, and portability of each model, culminating in a set of strategic recommendations (Section IV.C) to guide scientific teams in selecting the optimal tool for their specific context.

## II. On-Node Parallelism: A Comparative Analysis of GPU Portability Models

This section analyzes the developer experience and performance portability of models designed to abstract parallelism on a single compute node, particularly for GPUs and many-core CPUs.

### A. Kokkos: The C++ Library Approach to Performance Portability

Kokkos is not a single library but a comprehensive C++ programming model and ecosystem designed to provide true performance portability for scientific applications.1 This ecosystem includes Kokkos Core, which provides the fundamental programming model; Kokkos Kernels, which offers performance-portable math kernels (sparse, dense, and graph); and Kokkos Tools, which provides hooks for profiling and debugging.10

User Experience: The Programming Paradigm

The user experience is that of a modern C++ template library. It is designed to be "descriptive, not prescriptive" and aligns with the evolution of the C++ standard.1 Developers must learn to express their algorithms using Kokkos's core abstractions 13:

* **Execution Spaces:** Logical groupings of computation units, such as a CUDA device or an OpenMP thread pool.13
* **Memory Spaces:** Abstractions for different memory locations, such as high-bandwidth GPU memory (HBM), standard DDR, or non-volatile memory.1
* **Execution Patterns:** Standard parallel algorithms like Kokkos::parallel\_for, Kokkos::parallel\_reduce, and Kokkos::parallel\_scan.14

User Experience: The Porting Effort

Adopting Kokkos is a strategic, multi-year investment, not a quick port. Analysis of existing application ports provides concrete estimates of this effort:

* An **"optimistic estimate" suggests 10% of an application's code must be rewritten** to adopt the Kokkos shared-memory parallel model.1
* For typical scientific applications (300k-600k Lines of Code, e.g., LAMMPS, Uintah), this translates to **2-3 Man-Years** of developer effort.1
* For large-scale libraries or frameworks, the cost is significantly higher. The E3SM climate model (1 million lines) is estimated at **5 Man-Years**.1

Performance Portability (The Payoff)

The justification for this high porting cost is achieving true, single-source performance portability. The Kokkos C++ abstractions are mapped by the library to vendor-native, high-performance backends. This allows a single application codebase to run efficiently on all major HPC architectures 1:

* **NVIDIA GPUs** via the CUDA backend.1
* **AMD GPUs** via the HIP/ROCm backend.1
* **Intel GPUs** via the SYCL backend.18
* **Multicore CPUs** via OpenMP or C++ threads backends.1

User Experience (Challenges and "Gotchas")

The learning curve for Kokkos is non-trivial, and developers can fall into performance traps. A crucial user-reported challenge is that the Kokkos API can be "CPU-biased".20 This can lead non-expert developers to create design patterns that "spend too much time copying data in and out of GPU memory," effectively negating the performance benefits of the accelerator.20

Furthermore, users report difficulty in writing a single piece of code that achieves optimal performance *both* through CPU vectorization and CUDA portability.20 This suggests that while Kokkos provides functional portability, achieving high performance on different architectures still requires a deep understanding of the underlying hardware and the Kokkos abstractions.

The decision to adopt Kokkos is therefore a major strategic commitment to a C++-centric development model. The high up-front cost is amortized over the application's lifecycle by eliminating the need to write and maintain separate, vendor-specific code branches.

### B. Directive-Based Models: OpenMP and OpenACC

Directive-based models offer a different value proposition: accelerating existing codebases (written in C, C++, or Fortran) with minimal code intrusion.21

Contrasting Philosophies (The Core User-Experience-Difference)

A key difference in user experience between the two primary directive standards is their fundamental philosophy 6:

* **OpenACC** is a **"descriptive"** approach. The programmer uses directives (pragmas) to *identify* data-independent loops and regions of code. The *compiler* is then responsible for deciding how to best parallelize and offload those loops for the target architecture.6
* **OpenMP** is a **"prescriptive"** approach. The programmer uses directives to *explicitly tell the compiler how and where* to parallelize, mapping parallelism onto a hierarchy of teams, threads, and vectors.6 This offers more control but demands more expertise.

OpenACC: The "Easy On-Ramp"

OpenACC is designed for "significantly less programming effort" than low-level models like CUDA.22 It provides an "easy on-ramp to accelerated computing," a primary reason for its popularity in the scientific community.2

* **Target Audience:** Its multi-language support (C, C++, Fortran) makes it a preferred solution for porting large, existing scientific codebases.21
* **Productivity:** A user experience report from Princeton Plasma Physics Laboratory (PPPL) highlights that OpenACC was their "preferred approach due to its **ease of implementation and non-interference with the CPU code**".2
* **Performance:** This ease-of-use does not equate to low performance. A SC22 presentation on accelerating a computational fluid dynamics (CFD) solver demonstrated that "careful choice of OpenACC decorations" led to:
  + Achieving **46% of peak GPU FLOPS** on the most computationally expensive kernel.
  + A **500-times speedup** on an NVIDIA A100 GPU compared to a single modern Intel CPU core.
  + **Ideal weak scaling up to 13,824 GPUs** on the OLCF Summit supercomputer.25

OpenMP Target Offload: The Vendor-Agnostic Standard

OpenMP target offload is the industry-standard, vendor-agnostic solution for directive-based offloading. It is supported by all major HPC vendor compilers, including the NVIDIA HPC SDK, AMD ROCm (via LLVM), and Intel oneAPI.9 Like OpenACC, it supports C/C++ and Fortran.23

* **User Experience:** The "prescriptive" model requires the developer to explicitly manage offload regions (e.g., #pragma omp target) and data mapping (map(...)).28
* **Performance and Maturity Concerns:** While OpenMP is the *standard*, its implementations for GPU offload are perceived as less mature than OpenACC's. This is critically important for complex C++ applications. Researchers implementing a Kokkos backend using OpenMP target constructs reported two major findings:
  1. Performance **"varies widely based on details of the implementation strategy and the chosen compiler"**.8
  2. The performance **"decreases with increasing complexity of the investigated algorithms"**.8

This strongly implies that while OpenMP target offload is functionally portable, its performance portability is inconsistent. The user experience for complex C++ codes involves significant compiler-specific tuning and may yield poor results for highly complex algorithms.

The Developer's Dilemma: OpenACC vs. OpenMP

This situation creates a "painful" dilemma for scientific programmers. The PPPL user report captures this perfectly: developers prefer OpenACC for its ease and maturity, but "it appears that OpenMP is now being promoted more forcefully" by vendors and facilities.2 This forces development teams to migrate from the mature OpenACC model and "work with immature implementations" of OpenMP target offload, leading to a period of reduced productivity and performance.2

A developer must therefore choose between:

1. **OpenACC:** For immediate productivity, high performance, and minimal code intrusion, especially on NVIDIA hardware. This comes at the risk of the standard becoming obsolete.
2. **OpenMP:** For long-term, cross-vendor portability, which is the industry's stated direction. This comes at the cost of a steeper learning curve, more complex ("prescriptive") code, and battling "immature" compiler implementations.2

### C. Standard C++ Approaches to Heterogeneous Parallelism

For C++-centric applications, two language-based (non-directive) approaches are emerging as viable alternatives.

SYCL: The C++ Abstraction Layer

SYCL is an open, royalty-free, Khronos-group standard that provides a cross-platform abstraction layer for programming heterogeneous processors (CPUs, GPUs, FPGAs) using modern ISO C++.30

* **User Experience:** It is *not* a directive model. Developers write standard C++ using templates and lambda functions, with host and kernel code often co-located in the same source file.31 It is a "language-based alternative" to directives.30
* **Role in the Ecosystem:** SYCL serves two roles. As a user-facing model, it allows developers to write heterogeneous C++ applications. Perhaps more importantly, it serves as a *foundational backend target* for other portability layers. Features in the SYCL 2020 standard are "an important step for implementing backend support for SYCL in the Kokkos... performance portability ecosystem".19 Kokkos, for example, uses SYCL as its backend to achieve performance portability on Intel GPUs.18

C++17 Standard Parallelism (std::par): The "Zero-Effort" Model

This approach leverages parallel algorithm features introduced in the C++17 standard. By invoking standard library algorithms (e.g., std::for\_each, std::transform\_reduce) with a parallel execution policy (std::execution::par or std::execution::par\_unseq), developers can request parallelization.7

* **User Experience:** This model is the ideal for C++ purists. It requires **"no language extensions, pragmas, directives, or non-standard libraries"**.7 The user writes 100% standard, portable C++ code.
* **Implementation:** This standard C++ code is GPU-accelerated by specific compilers:
  + **NVIDIA:** The **NVC++** compiler, when used with the -stdpar command-line flag, automatically offloads std::par algorithms to the GPU.7
  + **AMD:** ROCm 6.1 and later support this via **HIPSTDPAR**, which also requires only a compiler flag to offload to AMD accelerators.35
* **The Data Management Trade-off:** The "zero-effort" experience comes at the cost of ceding all explicit data management. Data movement between host (CPU) and device (GPU) memory is handled **"implicitly and automatically"** by the underlying system, such as CUDA Unified Memory (for NVIDIA) or Heterogeneous Memory Management (HMM) (for AMD).7
* **Performance:** This is not a "toy" model. User reports and benchmarks show it can be significantly faster than multicore CPU parallelism.36 For example, a Lulesh proxy application kernel implemented with std::par on an NVIDIA GPU achieved a **13.57x speedup** over the same code parallelized with OpenMP on a 64-core AMD EPYC CPU.37

This std::par model is ideal for **prototyping and new C++ applications**. A developer can write a correct, standard C++ parallel algorithm and receive "good enough" GPU performance for free. If that kernel later proves to be a performance bottleneck, it can be refactored into a more explicit model like Kokkos or OpenMP for fine-grained tuning, while the rest of the application remains in standard C++.

## III. Inter-Node Parallelism: Distributed Memory Communication Strategies

This section analyzes the models for scaling an application *across* compute nodes, focusing on the user experience and performance of MPI versus the PGAS model.

### A. The MPI Standard: MPICH vs. OpenMPI

The Message Passing Interface (MPI) is the de facto standard for distributed-memory programming. Users write their code to the MPI standard, making it portable between implementations.38 The choice between MPICH and OpenMPI is one of implementation philosophy, not user-facing API.

* **Implementation Philosophies:**
  + **MPICH:** A high-performance, "widely portable implementation".39 Its goals are twofold: to provide an efficient implementation for diverse hardware (from commodity clusters to proprietary supercomputers) and to serve as a **"modular framework" for cutting-edge MPI research**.39 It is often considered the "reference implementation".40
  + **OpenMPI:** An "open source implementation... developed and maintained by a consortium" of academic and industry partners.41 It is the merger of four previous MPI projects and "targets the common case".40 Its goal is to be a **"production-quality"** community-driven MPI.42
* Comparative User Experience and Performance:  
  For the application developer, the experience is nearly identical. The choice is often made by the HPC facility, which provides a module optimized for its specific interconnect. Performance differences are typically minor and system-dependent. OpenMPI has been reported to have faster process launching at large scale (1000+ processes) 40, and recent optimizations to MPI\_AllReduce in OpenMPI (e.g., at AWS) show it now matches or exceeds the performance of MPICH and commercial MPIs.44 A key feature where MPICH has shown leadership is in ABI compatibility, which is critical for E4S to enable libraries to be "built once, run many".45
* Crucial Feature: "GPU-Aware" MPI  
  For modern hybrid parallelism, the single most important feature is "GPU-Aware" (e.g., "CUDA-aware") support.46
  + **What it is:** A GPU-aware MPI library "can send and receive GPU buffers directly".46
  + **User Experience Benefit:** This is a massive performance and productivity win. The developer passes a GPU device pointer *directly* to an MPI call (e.g., MPI\_Send). The library automatically handles the transfer, **eliminating the need to manually stage data** by copying it to host (CPU) memory first (e.g., via cudaMemcpy).47
  + **How it Works (OpenMPI):** OpenMPI's preferred mechanism for this is the **UCX (Unified Communication X)** transport layer.46 It relies on features like NVIDIA's Unified Virtual Addressing (UVA) to automatically detect whether a pointer is located on the host or the device.46

For most scientific users, the choice *between* MPICH and OpenMPI is not a code-level decision. The critical question to ask a facility is: **"Does the provided MPI module have GPU-Aware support (e.g., via UCX) enabled?"** An application's performance will be crippled by manual data staging if this feature is not available.

### B. UPC++: The PGAS Model for C++

UPC++ is a C++ library (not a language) that provides a Partitioned Global Address Space (PGAS) programming model.48 This model gives each process its own private memory *plus* access to a global address space, enabling one-sided communication.49

* **The PGAS Philosophy:** UPC++ contrasts with MPI's two-sided message-passing. The model is built on:
  1. **One-sided Communication (RMA):** Direct put and get operations to remote memory.48
  2. **Remote Procedure Calls (RPC):** The ability to invoke a function on a remote process.4
  3. **Asynchrony by Default:** Operations are asynchronous and managed via standard C++ futures and promises, allowing for flexible overlap of communication and computation.4
* User Experience (Productivity):  
  The primary benefit of the PGAS model is avoiding the "punitive synchronisation costs of two-sided communication".51 For a developer, this translates to not having to manually ensure that a receive (MPI\_Irecv) is posted before a corresponding send (MPI\_ISend).52 This "message matching" is a major source of bugs and "performance anomalies" in complex MPI applications, and its avoidance in UPC++ is a significant productivity gain.52
* Performance and Target Problem Class:  
  UPC++ is not a general-purpose MPI replacement. Its design—low-overhead RMA/RPC and aggressive asynchrony—makes it excel at irregular applications.4
  + **Case Study 1 (Distributed Hash Table):** A UPC++ implementation achieved **"near-linear weak scaling up to 34,816 cores"**.4 This is a classic irregular problem where one-sided put/get operations are more natural than two-sided messages.
  + **Case Study 2 (Sparse Solver Component):** A UPC++ implementation **outperformed MPI variants by up to 3.1x** and showed robust strong scaling.4
  + Microbenchmarks confirm that UPC++ RMA is competitive with or *better than* MPI-3 RMA, showing up to 25% lower latency and 33% higher bandwidth.4
* User Experience (Downsides and Trade-offs):  
  The critical trade-off for adopting UPC++ is tool support. The MPI ecosystem has "many sophisticated tools... for performance tuning" (debuggers, profilers).3 In contrast, the tools for PGAS models like UPC++ are described as "more research-grade, in the bad way".3 Parallel I/O library support is also less mature.3

This presents a clear decision fork for a C++ developer:

* **Choose MPI** for: (a) "regular" problems (e.g., dense linear algebra, structured grids, CFD), (b) applications where mature ecosystem tools are critical, or (c) applications that must support Fortran or C.
* **Choose UPC++** for: (a) C++ applications dominated by *irregular* communication (e.g., graph analytics), and (b) teams willing to sacrifice mature tooling for a (potentially) more productive programming model and higher performance *on that specific problem class*.

## IV. Synthesis: Hybrid Programming Models and Strategic Recommendations

Scientific applications must combine on-node and inter-node parallelism. This synthesis section evaluates the user experience of hybrid models and provides a final decision-making framework.

### A. The "MPI + X" Paradigm: The De Facto Standard

The dominant HPC paradigm is "MPI + X," where MPI handles inter-node communication and "X" (e.g., OpenMP, OpenACC, Kokkos) handles on-node parallelism.53 The SC22 CFD solver that scaled to 13,824 GPUs was an **MPI+OpenACC** code.25 For Kokkos applications, the assumed model is MPI+Kokkos, where developers must pack data from Kokkos::Views into buffers for MPI sends.52

The performance of *all* "MPI+X" models is critically dependent on **GPU-Aware MPI**. The SC22 CFD solver's scaling was explicitly "improved by reduced communication times via CUDA-aware MPI".25 Without this feature, the application is crippled by D-H-H-D (Device-to-Host, Host-to-Device) copies at every node boundary.

### B. An Emerging Alternative: UPC++ and Kokkos Interoperability

A powerful, C++-only alternative is emerging: combining Kokkos for on-node parallelism with UPC++ for inter-node communication. UPC++ is explicitly designed to interoperate with models like OpenMP and CUDA.48

A case study where MPI was *replaced* with UPC++ in an existing Kokkos heat conduction code reveals the trade-offs.52

* **User Experience (Challenges):** This is *not* a simple drop-in replacement. It requires expert-level C++ data management. Because Kokkos cannot allocate memory in UPC++'s global device memory, the developer had to *manually* allocate GPU memory using upcxx::device\_allocator and then *pass that raw pointer* to a Kokkos::View constructor.50 This is a complex operation that breaks Kokkos's standard memory-management abstractions.
* **User Experience (The Payoff):** This high setup cost yielded significant benefits.
  1. **Improved Performance:** The Kokkos+UPC++ version demonstrated **"vastly improved performance and scalability"** and "exhibited good performance".52
  2. **Higher Productivity:** This is the most critical finding. The "straightforward" or "natural" MPI+Kokkos version suffered from **"performance anomalies."** The authors speculate this is due to *missed rendezvous*—a failure to always pre-post an MPI\_Irecv before its matching MPI\_ISend arrives.52 This is a notoriously difficult and common MPI tuning problem. The Kokkos+UPC++ version, by using one-sided communication, *does not have this problem*.

This case study reveals a nuanced developer choice. The "MPI+Kokkos" model is standard, but achieving good performance requires *expert-level MPI tuning* to avoid common scaling bottlenecks. The "UPC+++Kokkos" model is *more complex to set up* at the code level but results in a *more productive development loop* and *better "out-of-the-box" performance* for C++ teams, as the PGAS model naturally avoids the most common MPI scaling problems.

### C. Final Recommendations and Decision Matrix

Based on this analysis, the following strategic recommendations are provided for scientific software teams.

* **For New, Flagship C++ Applications:**
  + **On-Node:** **Kokkos**. The 2-3 man-year porting cost 1 is a strategic investment to obtain a single-source application that performs well on all NVIDIA, AMD, and Intel GPUs.17
  + **Inter-Node:** **MPI** (for mature tooling and regular problems) or **UPC++** (for C++-only, irregular problems where 3.1x speedups are possible 4).
* **For Existing C/C++/Fortran Legacy Codes:**
  + **On-Node:** **OpenACC**. It is the "easy on-ramp" 2, "descriptive" 6, and proven to deliver massive speedups (500x) and scaling (13k+ GPUs) with minimal code intrusion.25 This is the choice for a PI who needs their existing, mature Fortran CFD code running on a GPU *now*.
* **For Long-Term, Vendor-Agnostic C/C++/Fortran:**
  + **On-Node:** **OpenMP Target Offload**. It is the vendor-agnostic *standard*.26 However, the user must be prepared for a "prescriptive" 6 model, a "painful" 2 transition from OpenACC, and performance that "varies widely" and "decreases with complexity," especially for C++.8
* **For Modern C++ Prototyping:**
  + **On-Node:** **C++17 std::par**. For new C++ kernels that fit STL algorithm patterns, this is the "zero-effort, zero-lock-in" path to GPU acceleration on both NVIDIA and AMD.7 It is perfect for prototyping before committing to a larger Kokkos or directive port.

#### Programming Model Selection Matrix

| **Model** | **Primary Paradigm** | **Primary Language(s)** | **Best-Fit Problem Class** | **GPU Portability (NVIDIA/AMD/Intel)** | **Developer Effort (Porting)** | **Scalability Model** | **Key User Experience** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Kokkos** | C++ Template Library | C++ only | General / All | Excellent (via backends CUDA, HIP, SYCL) 1 | Very High (2-3 Man-Years) 1 | On-Node | Maximum control & portability; steep learning curve.20 |
| **OpenMP (Target)** | Compiler Directive | C/C++/Fortran 23 | General / All | Good (Vendor-supported standard) 26 | Medium (Intrusive directives) 28 | On-Node | "Prescriptive" 6; "immature" compilers; "painful".2 |
| **OpenACC** | Compiler Directive | C/C++/Fortran 21 | Regular (CFD, Grids) | Excellent (NVIDIA), Limited (Others) | Low (Non-intrusive) 22 | On-Node | "Descriptive" 6; "easy on-ramp"; mature; 500x speedup.2 |
| **UPC++** | C++ Library (PGAS) | C++ only 49 | Irregular (Graphs, Sparse) 4 | N/A (Interoperates with GPU models) 48 | High (C++ only) | Inter-Node (PGAS) | Productive async; avoids message-matching; "research-grade" tools.3 |
| **MPI (MPICH/OpenMPI)** | Library Standard | C/C++/Fortran | General / All | Excellent (via GPU-Aware MPI) 46 | Low (Standard) | Inter-Node (Message Passing) | Mature tools; robust ecosystem; GPU-Aware support is essential.46 |
| **C++ std::par** | C++ Standard Feature | C++17 only 7 | General / All | Good (NVIDIA/AMD) 7 | Very Low (C++17 only) | On-Node | "Zero-lock-in"; implicit data movement; great for prototyping.7 |

#### Works cited

1. Kokkos: C++ Performance Portability for Production, accessed November 6, 2025, <https://extremecomputingtraining.anl.gov/wp-content/uploads/sites/96/2019/08/ATPESC_2019_Track-2_3_8-1_830am_Trott-Kokkos.pdf>
2. OpenACC Birds of a Feather (BOF) at SC21 | OpenACC, accessed November 6, 2025, <https://www.openacc.org/events/openacc-birds-feather-bof-sc21>
3. UPC in HPC - experience and suggestions [closed] - Stack Overflow, accessed November 6, 2025, <https://stackoverflow.com/questions/6515171/upc-in-hpc-experience-and-suggestions>
4. UPC++: A High-Performance Communication ... - Amir Kamil, accessed November 6, 2025, <https://amirkamil.com/papers/ipdps19.pdf>
5. UPC++: A High-Performance Communication Framework for Asynchronous Computation - GitHub, accessed November 6, 2025, <https://raw.githubusercontent.com/wiki/symPACK/symPACK/pubs/ipdps19-upcxx.pdf>
6. Exploring Differences Between Directive-Based GPU Programming ..., accessed November 6, 2025, <https://pawseysc.github.io/sc20-gpu-offloading/>
7. Accelerating Standard C++ with GPUs Using stdpar | NVIDIA Technical Blog, accessed November 6, 2025, <https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/>
8. The Kokkos OpenMPTarget Backend: Implementation and Lessons ..., accessed November 6, 2025, <https://www.ornl.gov/publication/kokkos-openmptarget-backend-implementation-and-lessons-learned>
9. The Kokkos OpenMPTarget Backend: Implementation and Lessons Learned - OSTI, accessed November 6, 2025, <https://www.osti.gov/servlets/purl/2224192>
10. Kokkos, accessed November 6, 2025, <https://kokkos.org/>
11. Kokkos documentation, accessed November 6, 2025, <https://kokkos.org/kokkos-core-wiki/>
12. The Kokkos EcoSystem: Comprehensive Performance Portability For High Performance Computing - OSTI.GOV, accessed November 6, 2025, <https://www.osti.gov/pages/servlets/purl/1817719>
13. 2. Machine Model - Kokkos documentation, accessed November 6, 2025, <https://kokkos.org/kokkos-core-wiki/ProgrammingGuide/Machine-Model.html>
14. Get Started - Kokkos documentation, accessed November 6, 2025, <https://kokkos.org/kokkos-core-wiki/get-started.html>
15. Enhancing Kokkos with OpenACC - UPCommons, accessed November 6, 2025, <https://upcommons.upc.edu/bitstreams/b9aba2d3-ef04-47bf-82d4-29c290ea7f3e/download>
16. The Kokkos EcoSystem – News - Sandia National Laboratories, accessed November 6, 2025, <https://www.sandia.gov/news/publications/hpc-annual-reports/article/kokkos/>
17. Cross Platform Computing Framework? : r/HPC - Reddit, accessed November 6, 2025, <https://www.reddit.com/r/HPC/comments/y61n37/cross_platform_computing_framework/>
18. Multi-GPU Support - Kokkos documentation, accessed November 6, 2025, <https://kokkos.org/kokkos-core-wiki/API/core/MultiGPUSupport.html>
19. Experiences with implementing Kokkos' SYCL backend - OSTI.GOV, accessed November 6, 2025, <https://www.osti.gov/servlets/purl/2336667>
20. SYCL, CUDA, and others --- experiences and future trends in heterogeneous C++ programming? : r/cpp - Reddit, accessed November 6, 2025, <https://www.reddit.com/r/cpp/comments/1im99l2/sycl_cuda_and_others_experiences_and_future/>
21. Specification - OpenACC, accessed November 6, 2025, <https://www.openacc.org/specification>
22. Get Started | OpenACC, accessed November 6, 2025, <https://www.openacc.org/get-started>
23. C/C++ or Fortran with OpenMP\* Offload Programming Model - Intel, accessed November 6, 2025, <https://www.intel.com/content/www/us/en/docs/oneapi/programming-guide/2025-1/c-c-or-fortran-with-openmp-offload-programming.html>
24. Final Program - SC21, accessed November 6, 2025, <https://sc21.supercomputing.org/app/uploads/2021/11/SC21-Final-Program-211114.pdf>
25. SC 2022 | OpenACC, accessed November 6, 2025, <https://www.openacc.org/events/sc-2022>
26. Unified schemes for directive-based GPU offloading - arXiv, accessed November 6, 2025, <https://arxiv.org/html/2411.18889v1>
27. HPE COMPILER GPU OFFLOADING - OpenMP, accessed November 6, 2025, <https://www.openmp.org/wp-content/uploads/2022-04-29-ECP-OMP-Telecon-HPE-Compiler.pdf>
28. Intro to GPU Programming with the OpenMP API, accessed November 6, 2025, <https://www.openmp.org/wp-content/uploads/2021-10-20-Webinar-OpenMP-Offload-Programming-Introduction.pdf>
29. Accelerated Compute with OpenMP\* GPU Offload - Intel Community, accessed November 6, 2025, <https://community.intel.com/t5/Blogs/Tech-Innovation/Tools/Accelerated-Compute-with-OpenMP-GPU-Offload/post/1550828>
30. SYCL - C++ Single-source Heterogeneous Programming for ..., accessed November 6, 2025, <https://www.khronos.org/sycl/>
31. Data Parallelism in C++ Using SYCL\* - Intel, accessed November 6, 2025, <https://www.intel.com/content/www/us/en/docs/oneapi/programming-guide/2025-1/data-parallelism-in-c-using-sycl.html>
32. Offload C++ Standard Parallel Code to SYCL\* Device Using oneDPL - Intel, accessed November 6, 2025, <https://www.intel.com/content/www/us/en/developer/articles/technical/offload-cpp-standard-parallel-code-to-sycl-device.html>
33. NVIDIA HPC STANDARD LANGUAGE PARALLELISM, C++, accessed November 6, 2025, <https://www.olcf.ornl.gov/wp-content/uploads/4-7-22-ORNL-Stdpar.pdf>
34. Multi-GPU Programming with Standard Parallel C++, Part 1 | NVIDIA Technical Blog, accessed November 6, 2025, <https://developer.nvidia.com/blog/multi-gpu-programming-with-standard-parallel-c-part-1/>
35. C++17 parallel algorithms and HIPSTDPAR - AMD GPUOpen, accessed November 6, 2025, <https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-hipstdpar-readme/>
36. Standard C++ Parallelism using nvc++ is slow - Stack Overflow, accessed November 6, 2025, <https://stackoverflow.com/questions/71714200/standard-c-parallelism-using-nvc-is-slow>
37. 5 Ways to Accelerate with GPUs, accessed November 6, 2025, <https://labs.icahn.mssm.edu/minervalab/wp-content/uploads/sites/342/2024/05/FiveWays-HealthCare-April2024.pdf>
38. which one is easier to master, OpenMPI or MPICH? : r/HPC - Reddit, accessed November 6, 2025, <https://www.reddit.com/r/HPC/comments/1b9mtgq/which_one_is_easier_to_master_openmpi_or_mpich/>
39. MPICH Overview | MPICH, accessed November 6, 2025, <https://www.mpich.org/about/overview/>
40. mpi - MPICH vs OpenMPI - Stack Overflow, accessed November 6, 2025, <https://stackoverflow.com/questions/2427399/mpich-vs-openmpi>
41. Open MPI v5.0.x — Open MPI 5.0.x documentation, accessed November 6, 2025, <https://docs.open-mpi.org/>
42. Open MPI head of development — Open MPI main documentation, accessed November 6, 2025, <https://docs.open-mpi.org/en/main/index.html>
43. FAQ: General information about the Open MPI Project, accessed November 6, 2025, <https://www.open-mpi.org/faq/?category=general>
44. Recent improvement to Open MPI AllReduce and the impact to application performance, accessed November 6, 2025, <https://aws.amazon.com/blogs/hpc/recent-improvement-to-open-mpi-allreduce-and-the-impact-to-application-performance/>
45. MPICH: A High-Performance Open Source MPI Library for Leadership-class HPC Systems, accessed November 6, 2025, <https://cass.community/assets/artifacts/2024-06-bof-days/CASS-BOF-2024-MPICH.pdf>
46. 11.2.6. CUDA — Open MPI main documentation, accessed November 6, 2025, <https://docs.open-mpi.org/en/main/tuning-apps/networking/cuda.html>
47. An Introduction to CUDA-Aware MPI | NVIDIA Technical Blog, accessed November 6, 2025, <https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/>
48. UPC++ - NERSC Documentation, accessed November 6, 2025, <https://docs.nersc.gov/development/programming-models/upcxx/>
49. UPC++ v1.0 Programmer's Guide, Revision 2020.10.0 - eScholarship, accessed November 6, 2025, <https://escholarship.org/content/qt8vf7658w/qt8vf7658w_noSplash_21f7de4442a650db9241d38c604b8c09.pdf>
50. UPC++ v1.0 Programmer's Guide, Revision 2023.9.0, accessed November 6, 2025, <https://upcxx.lbl.gov/docs/html/guide.html>
51. Comparing One-Sided Communication With MPI, UPC and SHMEM - Cray User Group, accessed November 6, 2025, <https://cug.org/proceedings/attendee_program_cug2012/includes/files/pap195.pdf>
52. Demonstrating UPC++/Kokkos Interoperability in a Heat ... - Bitbucket, accessed November 6, 2025, <https://bytebucket.org/berkeleylab/upcxx/wiki/pubs/PAW-ATM21-upcxx-kokkos.pdf?rev=d50b0f83e17178f987760f9509fdfb3fde3d8db1>
53. Towards a Scalable and Efficient PGAS-based Distributed OpenMP - arXiv, accessed November 6, 2025, <https://arxiv.org/html/2409.02830v1>
54. Demonstrating UPC++/Kokkos Interoperability in a Heat Conduction Simulation (Extended Abstract) - eScholarship, accessed November 6, 2025, <https://escholarship.org/uc/item/1x06t965>