MPI Sessions: What is that and what are implementations?

PHILIP REDECKER, FernUniversität in Hagen, Germany

Abstract:

The traditional MPI world model has inherent limitations, such as rigid initialization constraints and a lack of flexibility in resource management across application components. To address these constraints, the MPI Sessions model, introduced in the MPI Standard 4.0, enables applications or their components to instantiate MPI resources tailored to specific communication needs. This paper explores the concept of MPI Sessions, highlights its benefits over the world model and examines its implementation in OpenMPI, demonstrating its practical application and potential impact on parallel computing architectures.

CCS Concepts: • **Computer systems organization** → Parallel architectures; Grid computing; • **Computing methodologies** → Distributed programming languages.

Additional Key Words and Phrases: MPI Sessions, parallel architectures, Message Passing Interface (MPI), OpenMPI, parallel programming, implementations of MPI

1 Introduction

Ever since the introduction of multicore processors the demand for scalable and optimized parallel programming architectures has grown exponentially. Parallel programming now has an ever expanding range of applications in both science and industry. As the scale of tasks for parallel programs and the capabilities of available hardware increased significantly the challenges for parallel programming architectures have also become more complex.

Today systems are generally categorized as shared memory systems, distributed memory systems or hybrid systems that combine features of both paradigms. The Message Passing Interface (MPI) standard, established in May 1994, serves as a cornerstone for designing parallel programs on distributed systems. While MPI 3.0 introduced capabilities for shared memory optimizations, its primary focus remains on distributed computing platforms [15]. The MPI Forum, which is responsible for formulating and publishing the standard, has regularly introduced updates to the MPI standard to keep pace with advances in hardware and application requirements over the years.

However, the original MPI world model has inherent limitations that have become increasingly apparent in light of recent developments. For example MPI cannot be initialized more than once, it cannot be reinitialized after a call to `MPI_Finalize` (which marks the end of a traditional MPI program) and it cannot be initialized across different components of an application without prior coordination. To address these issues, the MPI Sessions model was introduced in MPI version 4.0 [14].

The MPI Sessions model offers solutions to many of these challenges and it provides greater flexibility and modularity in managing MPI resources. However its introduction has also raised new challenges, some of which remain unresolved since the release of MPI 4.0 in 2021. The most recent update, MPI version 4.1, has begun to address some of these challenges with further refinements expected in future releases. Additionally the introduction of MPI Sessions poses implementation challenges for MPI libraries [3, 15].

One such implementation is OpenMPI, an open-source project collaboratively developed and maintained by a consortium of academic, research and industry organizations. In the latest version of OpenMPI (version 5.0.x) support for the MPI Sessions model has been introduced and the project continues to refine this implementation [17].

Author's Contact Information: Philip Redecker, philipredecker@gmail.com, FernUniversität in Hagen, Hagen, Germany.

This paper aims to provide a clear overview of the MPI Sessions concept and its significance for parallel computer architectures. It explores the challenges posed by this paradigm shift, examines its implementation in OpenMPI as a case study and analyzes the practical implications and ongoing challenges. By contextualizing MPI Sessions within the broader evolution of the MPI standard this work seeks to contribute to a deeper understanding of their role in modern parallel computing.

2 Related works

Previous research on MPI Sessions has laid important groundwork for understanding both the theoretical foundations and practical implementation challenges. Holmes et al. (2016) [9] provided an initial conceptual framework for MPI Sessions, introducing them as a solution to scalability challenges in exascale computing. Their work outlined how MPI Sessions could leverage runtime infrastructure more effectively than the traditional MPI world model and particularly focuses on resource management and initialization costs. The authors presented MPI Sessions as a fundamental shift in the MPI programming model and thus proposing it as a more flexible and scalable approach to parallel communication.

Building upon this theoretical foundation Hjelm et al. (2019) [8] presented the first comprehensive evaluation of an MPI Sessions implementation in OpenMPI. Their work moved from concept to practice and documents the challenges and solutions encountered during the actual implementation process. This research provided crucial insights into the practical aspects of integrating MPI Sessions into an existing MPI library, including performance implications and architectural considerations. The authors' findings demonstrate both the feasibility of implementing MPI Sessions in a production MPI library and highlighted areas requiring further development.

While Holmes et al. (2016) [9] focused on the theoretical underpinnings of MPI Sessions and Hjelm et al. (2019) [8] provided initial insights into their practical implementation, the following work builds upon these foundational studies. It presents their contributions and examines the current state of MPI Sessions with a particular focus on their implementation in OpenMPI and the challenges that have arisen as the paradigm continues to evolve.

3 Background

Parallel programming is a foundational concept in modern computing and it enables multiple processes to execute concurrently so it becomes possible to solve complex problems more efficiently. This approach has become increasingly vital with the rise of high performance computing, driven by advancements in data intensive applications such as machine learning, scientific simulations and big data processing. Traditional methods for improving computational performance, such as increasing processor clock frequencies, are reaching their practical and physical limits [20] due to power consumption, heat dissipation and other challenges. As a result parallel programming has emerged as a powerful alternative allowing tasks to be distributed and executed simultaneously across multiple cores or nodes.

The classification of parallel computing systems typically includes shared memory models, distributed memory models and hybrids that incorporate characteristics of each. Among these distributed memory systems present unique challenges such as managing explicit communication between processes running on separate nodes. These challenges are particularly relevant in modern computing where efficiency and scalability are critical for both scientific research and industrial applications.

The Message Passing Interface (MPI) was developed to address these challenges, providing a standardized framework for process communication in distributed systems. The following section introduces MPI, its world model, and the limitations that have driven the development of MPI Sessions in the MPI standard.

3.1 Overview of MPI

The endeavor to standardize a Message Passing Interface (MPI) began in the early 1990s and culminated in the publication of the first MPI standard in 1994. Developed by the MPI Forum, that consists of a consortium of experts from academia, industry and research, MPI was created to address the challenges inherent to communication in distributed systems. Unlike shared memory systems, in which processes can directly access a common memory space, distributed systems require explicit communication between processes because each of them operates within its own address space (see Fig. 1). MPI provides a standardized framework for managing this communication and so enables the development of portable and scalable parallel applications [3, 15, 20].

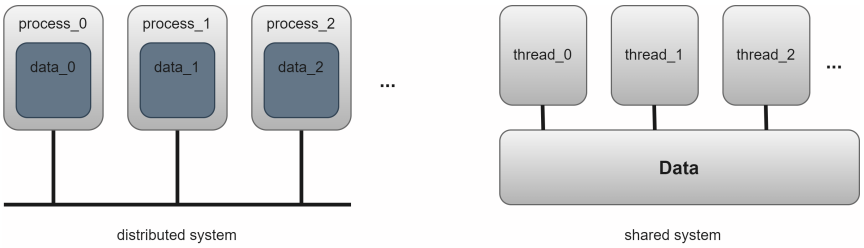

Fig. 1. The basic difference between distributed and shared memory systems, adapted from [20].

It is important to note that MPI is a specification, not an implementation. While the standard outlines the interface and functionality, actual implementations are created by third party organizations. This separation ensures flexibility and empowers hardware and software developers to tailor their implementations to specific requirements, such as maximizing performance, minimizing power consumption or optimizing for application specific goals. As such MPI has become a foundational tool for developing applications in distributed memory environments since it offers both portability and the potential for hardware acceleration. The MPI standard includes official language bindings for C and Fortran, reflecting its broad adoption across different programming paradigms. For clarity and consistency all code examples in this paper will be presented in the C programming language using the OpenMPI implementation introduced later.

Before any communication can occur in an MPI program, MPI must first be initialized. This is accomplished through the `MPI_Init` function, which prepares the environment necessary for processes to communicate. At the end of the program a call to `MPI_Finalize` marks its completion and releases any resources used by MPI (see Fig. 2). During initialization `MPI_Init` triggers the runtime system to perform essential tasks like process discovery and setup. A critical component of this process is the `add_procs` method, which identifies and establishes the initial set of processes available for communication. This functionality is pivotal in enabling seamless communication across distributed nodes and underlines MPI's ability to handle large scale applications efficiently [9, 15].

An important enhancement introduced with MPI 2.0 was thread support. The `MPI_Init_thread` function allows applications to specify the desired level of thread safety and that can range from basic single-threaded support (`MPI_THREAD_SINGLE`) to full multithreading capabilities (`MPI_THREAD_MULTIPLE`). This advancement broadens MPI's applicability to multithreaded environments and provides developers with tools to better manage concurrency and parallelism in complex systems [8, 15, 18].

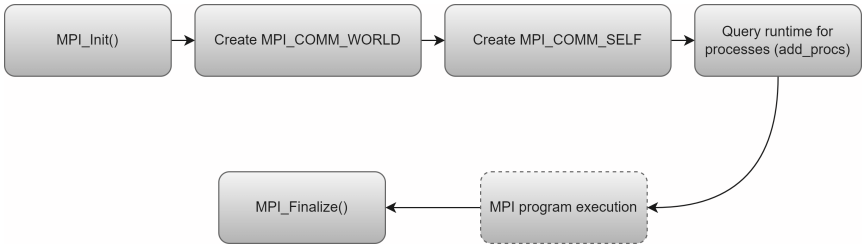

Fig. 2. The general course of a standard MPI program, starting with initialization and ending with finalization.

The core of MPI's communication model revolves around the concept of communicators. A communicator is a group of processes that can communicate with one another and each process within a communicator is assigned a unique rank starting from 0 [20]. This rank based addressing allows processes to target specific peers for communication. Upon initializatio MPI defines two default communicators: `MPI_COMM_WORLD` which includes all processes present at the start of the program and `MPI_COMM_SELF` which includes only the calling process. Communicators also play a critical role in managing complex applications. For example processes in the main program can be grouped separately from those used by libraries and like that it can be ensured operations in one group do not interfere with the other. During initialization MPI sets up internal data structures including communicator identifiers (CIDs). CIDs are unique identifiers assigned to communicators to ensure proper message routing and matching. Generating these CIDs requires collective communication among all processes, which adds to the initialization overhead. In the traditional world model, `MPI_COMM_WORLD` is assigned the first CID, and subsequent communicators receive unique CIDs through a collective agreement process [8, 9].

MPI facilitates communication between processes through explicit message passing operations [15]. Two core routines, `MPI_SEND` and `MPI_RECV`, are central to this functionality. These operations allow processes to exchange data and coordinate their actions. For example a process sending data to another must often wait for the receiving process to acknowledge receipt to ensure synchronization. This synchronization is essential in distributed systems where processes may operate at different speeds or encounter varying workloads.

Beyond basic communication MPI supports advanced features such as collective communication and derived data types. Collective operations such as `MPI_Bcast` for broadcasting data, `MPI_Gather` for gathering data to a single process and `MPI_Reduce` for performing operations like summation across processes enable efficient workload distribution in parallel applications. Derived data types allow for the transmission of complex, structured data and through this MPI's flexibility and usability across various applications is extended [15].

MPI also introduced the concept of `MPI_Info` objects, which are key-value pairs designed to provide hints and parameters for the underlying MPI implementation. These objects offer a mechanism to tailor MPI's behavior to specific runtime environments that make it possible to better optimize communication protocols or resource allocation strategies. Although their usage during initialization is limited in the world model however `MPI_Info` objects serve as a powerful tool for fine grained control and customization of MPI operations [8, 15].

One of the primary motivations for standardizing MPI was to simplify the development of scalable parallel programs [15]. Distributed memory systems inherently require lower-level communication mechanisms but the abstraction provided by MPI allows developers to focus on higher-level program logic. At the same time the standard encourages consistent performance optimizations across implementations and thus benefiting a wide range of applications, from

scientific simulations to industrial computations [20]. While these features make MPI a robust and versatile standard for parallel computing, they also highlight challenges and limitations of the MPI world model which will be explored in the next section.

3.2 The MPI world model and its limitations

At the core of MPI lies the world model, a framework that has defined its structure and functionality since its inception. Before the introduction of MPI Sessions in version 4.0 of the MPI standard the world model was the only available approach for using MPI. This model is based on initializing MPI with a call to `MPI_Init` and finalizing it with `MPI_Finalize` after completing computations. Upon initialization two default communicators `MPI_COMM_WORLD` and `MPI_COMM_SELF` become available to facilitate communication among processes. While developers can create additional communicators the majority of communication typically occurs through `MPI_COMM_WORLD` [9, 20].

The MPI world model relies heavily on the `MPI_COMM_WORLD` communicator which tightly couples all processes in the application. These processes are initialized collectively at program startup via `MPI_Init` and the model assumes a fixed number of processes for the program's entire lifetime. This design eliminates the overhead of dynamic process creation and allows MPI libraries to optimize communication by precomputing necessary configurations [15, 20]. The desired number of processes can be specified through the program's execution command (see Figure). However this fixed process model introduces significant limitations such as the inability to adjust the number of processes at runtime. While MPI 2.0 introduced `MPI_Comm_spawn` enabling the creation of additional processes during execution, these new processes operate within separate communicators and are not included in `MPI_COMM_WORLD` leaving some restrictions unresolved [15]. Another notable characteristic of the MPI world model is its collective coupling of processes. All processes must start, synchronize and terminate as a single unit, which complicates modularity. This behavior that is inherent in the design of `MPI_Init` and `MPI_Finalize` means all processes share the `MPI_COMM_WORLD` communicator further restricting modularity and flexibility.

Communication within the world model is explicit. Processes interact through point-to-point communication or collective operations with each process identified by a unique rank within a communicator [20]. Point-to-point operations like `MPI_Send` and `MPI_Recv` require explicit specification of both when and how data is exchanged between processes. This explicit nature makes MPI particularly suited for distributed systems, in which optimizing communication is critical. In addition to point-to-point communication MPI supports collective operations which involve multiple processes within a communicator. Examples include `MPI_Bcast` for broadcasting data, `MPI_Gather` for gathering data to a single process and `MPI_Reduce` for performing operations such as summation across processes. These collective operations facilitate efficient workload distribution in parallel applications.

Synchronization and buffering are essential aspects of MPI communication. While some functions like `MPI_Send` block execution until a matching receive operation occurs, non-blocking variants such as `MPI_Isend` and `MPI_Irecv` allow for computation and communication overlap and can help with enhancing performance [15, 20]. Explicit synchronization such as through `MPI_Barrier` ensures all processes reach a specific point in execution before proceeding providing robust control over execution flow.

Finally the shared nature of `MPI_COMM_WORLD` introduces challenges for isolating components, especially when integrating libraries that also depend on MPI. Because `MPI_COMM_WORLD` is a global resource encompassing all initialized processes, it complicates modular development and interoperability [20]. A critical analysis of the MPI world model's characteristics exposes several limitations inherent to its design.

Firstly there is a lack of modularity and it is especially difficult to use MPI in component based systems due to the global initialization and reliance on the `MPI_COMM_WORLD` communicator. Also it is not possible to initialize MPI from such components and that results in the need for developers to initialize MPI in the main application and create components so that these components assume the initialization of MPI from an outside entity [9]. Moreover `MPI_Info` objects, which could provide critical runtime-specific hints for optimizing initialization or resource allocation, are underutilized in the traditional world model [10]. The inability of `MPI_Init` to accept `MPI_Info` parameters limits the flexibility of the initialization process, restricting dynamic configurations that could otherwise improve modularity and efficiency [8]. Furthermore In the MPI world model once `MPI_Finalize` has been called MPI cannot be reinitialized. This happens because once `MPI_Finalize` is called the resources allocated during initialization are released and the MPI environmentis considered terminated. That means also the finalizing of MPI has to happen in the main part of the program and components cannot be allowed to do so because errors could happen if a component finalizes MPI although the main program still has to continue using it. Another limitation connected to components and libraries is that in a dynamic or adaptive workflow environment it might be desirable to perform multiple phases of MPI computation in a way that these are connected to non-MPI tasks. This is also not possible since MPI cannot be reinitialized and like that the application must remain in the MPI environment throughout its runtime even when MPI is not needed which may lead to inefficiencies [9, 15].

Secondly the process management within the MPI world model is inflexible and it is not possible to add or remove processes. Also MPI does not support multiple initializations within the same program execution and both of these characteristics make it difficult to run applications that need dynamic process management [9]. The `add_procs` method, which is responsible for process discovery and setup during initialization, adds significant overhead particularly in large-scale applications. This overhead arises from the need to gather and distribute process information across all nodes and therefore further limiting the scalability of the fixed process model [8, 12]. Overall it has been suggested many times that the performance of MPI could be enhanced by increasing the possibility to interact with runtime systems. For instance leveraging information about the physical topology can help with mapping MPI processes to allocated nodes more effectively and that could help to optimize the performance of communication. This is however not really possible withn the scope of the traditional MPI world model [6, 15, 16].

Thirdly a big challenge in the MPI world model is the scalability because the number of processes have drastically increased in recent years [12]. All of these processes in the MPI world model will be contained in the `MPI_COMM_WORLD` communicator after initialization and thus create a big overhead. While MPI 2.0 introduced thread support via `MPI_Init_thread`, this functionality adds complexity and potential overhead especially at higher threading levels like `MPI_THREAD_MULTIPLE`. The inconsistent implementation of threading support across MPI libraries further complicates efforts to utilize multithreading in scalable applications [8, 18]. Considering that it is likely that in future applications more and more processes might be needed for computation the overhead will increase accordingly and therefore this poses a challenge for the MPI world model because lowering the overhead is always desirable [9, 20].

Fourthly errors during or prior to initialization can disrupt the execution of the application and any non-trivial error affects the `MPI_COMM_WORLD` communicator, because its functionality relies on the entire set of processes involved in the job. Since `MPI_COMM_WORLD` encompasses all processes the failure of even a single process can propagate and compromise the entire communication structure. This highlights fault tolerance as a significant challenge for the MPI world model because it lacks inherent mechanisms to restart or replace failed processes within a running job [11, 15].

Fifthly MPI has applications beyond traditional high-performance computing (HPC) including interfaces for languages like Java, Python and R which are commonly used in scientific computing. However MPI sees limited use in broader

non-scientific contexts despite supporting client-server models. Features like `MPI_COMM_DISCONNECT` provide some fault tolerance by allowing disconnection from broken connections without process failures. Yet this is restricted to distinct MPI jobs that do not share the `MPI_COMM_WORLD` communicator limiting broader applicability and risking unnecessary termination of functional processes. Because of that introducing limited connection groups could prevent unnecessary termination of processes hosting functional servers or clients [9].

The challenges and limitations inherent in the MPI world model underscore the need for advancements in future iterations of the MPI standard. These challenges form the basis for defining the requirements that new approaches must address. The following sections introduce MPI Sessions, a feature introduced in version 4.0 of the MPI standard. MPI Sessions offer an alternative to the traditional world model and the following analysis will examine how MPI Sessions address the limitations of the MPI world model and evaluate their effectiveness in meeting the evolving demands of modern parallel computing [14].

4 MPI Sessions

The main idea of MPI Sessions is to offer a more flexible approach to managing MPI environments allowing for better modularity, dynamic process management and improved integration with other programming paradigms. Unlike the rigid MPI world model MPI Sessions provide a framework for managing multiple communicators and distinct communication contexts within a single application. This new approach aims to address the challenges of scalability, modularity and process management making it more suitable for modern parallel computing environments [9].

4.1 Introduction to MPI Sessions

MPI Sessions introduce a paradigm shift in how applications interact with the MPI environment, offering a more flexible and modular alternative to the traditional MPI world model. Unlike the world model, which tightly couples initialization, resource management and communicator creation MPI Sessions decouple these aspects and like that it is possible to provide developers with greater control and adaptability.

A core feature of MPI Sessions is the ability to initialize the MPI environment without immediately setting up communication resources. In the traditional MPI world model the creation of the global communicator `MPI_COMM_WORLD` is coupled with the initialization of the environment, which creates a static and rigid framework for communication [9, 15]. In contrast MPI Sessions relax this constraint since through sessions each session is allowed to act as its own local handle to the MPI library. This means that communication resources, such as communicators, can be set up dynamically and only when they are actually needed by the application. This incremental approach significantly reduces overhead and enables more efficient use of system resources, especially in modular or dynamically evolving workflows.

Each session handle provides access to various runtime functions and configuration options. For instance developers can use the session handle to query the current system resources available or to gather runtime information about specific tasks. This querying capability opens up possibilities for tailored resource management, including setting different configurations for error handling, communication optimization and process management based on the specific needs of the application. The ability to manage these configurations independently for each session provides a much more flexible and efficient way to handle communication and resources compared to the rigid structure of the world model.

In MPI Sessions communicators are not created automatically when a session is initialized, as is the case with `MPI_COMM_WORLD` in the traditional model. Instead developers must explicitly define the set of processes that will participate in communication within each session. This is done by first creating an MPI Group from the session

handle using the function `MPI_Group_from_session` [9]. An MPI group is essentially a collection of processes that are logically grouped together for communication purposes. The flexibility in defining groups means that developers can design communication contexts that are highly specific to the requirements of their application. For example, a traffic simulation might require separate communicators for different regions or time steps and MPI Groups make it possible to specify these needs precisely [15]. The initialization procedure and actions taken during the lifetime of a MPI Session are shown in Fig. 3.

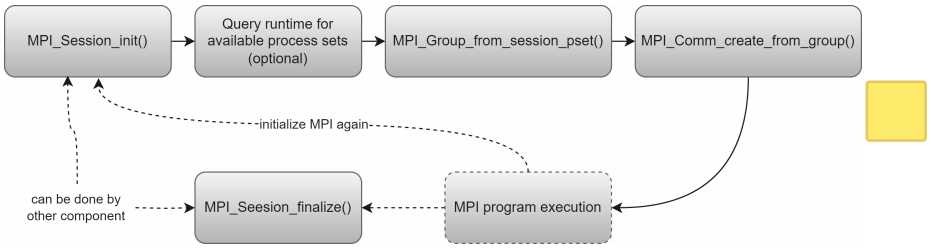

Fig. 3. The general course of a MPI Sessions program, showcasing the improved modularity that they offer, adapted from [8].

Once an MPI group is defined, it can then be used to create custom communicators. These communicators are scoped only to the specific session and group they are part of, thus ensuring that no global state or data is shared between different sessions or communicators. This isolation allows for more modular application design, where different parts of the application can operate independently without interfering with one another. The ability to create communicators on demand based on specific application needs stands in stark contrast to the world model, where `MPI_COMM_WORLD` is created automatically and all processes are tightly coupled under this global communicator [15]. By decoupling the creation of communicators and allowing them to be scoped within sessions, MPI Sessions provide greater modularity and that enables developers to create more dynamic parallel workflows.

Another major advancement introduced by MPI Sessions is the ability to support multiple independent sessions within the same application or even within the same process [2, 15]. This feature addresses a significant limitation of the traditional MPI world model, where all processes must be part of a single global communicator `MPI_COMM_WORLD`. In MPI Sessions each session operates independentlywith its own set of communicators and process groups. This makes it possible to run parallel tasks within a single application that operate independently of one another through using separate MPI Sessions for different tasks or stages of computation.

The independence of sessions is facilitated through the use of session handles and their associated APIs. Each session is created using the `MPI_Session_init` function, which initializes the session and assigns it a unique handle. This handle serves as a reference for all subsequent operations within that session, such as querying runtime information, defining groups or creating communicators. Since sessions are independent, it is possible to run different parts of an application under separate MPI Sessions each with its own set of configurations, optimizations and communication resources. This ability to manage multiple independent sessions in the same process represents a major leap forward in how MPI can be used to address complex, heterogeneous parallel applications [15].

In MPI Sessions fine-grained control over resource management is possible through the use of `MPI_Info` objects, which are associated with session handles. `MPI_Info` provides a mechanism for specifying detailed configuration parameters, such as memory management, error handling and communication settings. By combining session handles

with MPI_Info developers can create highly optimized communication paths that adapt to the specifics of the runtime environment. This customization extends beyond simple communicator creation and allows applications to respond dynamically to varying resource availability, system architecture and workload requirements [8, 9].

Overall the introduction of MPI Sessions marks a significant evolution in the MPI standard. By decoupling initialization, enhancing modularity and offering greater scalability MPI Sessions address many of the traditional world model's limitations. These innovations allow developers to create parallel applications that are more flexible, adaptable and efficient while also facilitating the creation of more complex and dynamic workflows. In the following section it will explored how these features directly tackle the constraints of the original world model and how they are offering solutions to its long standing challenges and paving the way for a more adaptable and efficient parallel programming paradigm.

4.2 Addressing the limitations of the MPI world model with MPI Sessions

In the previous section the limitations of the traditional MPI world model were outlined highlighting areas such as its lack of modularity, inflexible process management, scalability challenges (particularly with the MPI_COMM_WORLD communicator) and limited usability in non-HPC tasks as well as its global error handling constraints. MPI Sessions which were introduced in version 4.0 of the MPI standard provide a powerful alternative designed to overcome these obstacles and better align with the demands of modern parallel computing.

One of the most significant limitations of the MPI world model is its rigidity. The tight coupling of initialization, resource management and communicator creation makes it challenging to develop modular, reusable and dynamic applications. MPI Sessions address this by decoupling these processes and thus offering a more flexible framework for managing MPI resources. With MPI Sessions developers can tailor the MPI environment to their specific needs by defining groups and communicators dynamically. This allows for the creation of specialized communicators that meet the distinct requirements of individual application modules. For example an application can define separate communicators for computational tasks, data aggregation and fault tolerant communication and so enable cleaner separation of concerns, improved modularity and dynamic process management [15].

The MPI world model assumes a static number of processes throughout the program's execution, which can hinder applications requiring dynamic scaling. In contrast MPI Sessions allow processes to operate independently and maintain multiple session handles to the MPI library. This flexibility enables developers to create, manage and terminate processes dynamically. Additionally session specific communicators facilitate the separation of concerns which allows distinct tasks or modules to operate independently without interfering with each other. The reliance on MPI_COMM_WORLD in the world model introduces scalability challenges as it requires all processes to participate in a single tightly coupled global communicator. This design often results in communication overheads and inefficiencies as the number of processes increases. By contrast MPI Sessions eliminate the automatic creation of MPI_COMM_WORLD at initialization allowing applications to scale more effectively. Developers can define communicators that span only the required processes. This enables the efficient allocation of resources and it reduces bottlenecks in large-scale systems. For instance in distributed simulations or complex workflows, MPI Sessions allow for the division of processes into logically separate groups which can greatly improve scalability and ease task distribution [9, 10, 15].

Broader applicability enhancements introduced by MPI Sessions extend MPI's usability beyond traditional HPC applications. The ability to dynamically create groups and communicators has proven particularly valuable in non-HPC contexts, such as server-client architectures or data-intensive applications. For instance MPI can now be employed in systems requiring limited connection groups, where critical server or client tasks must remain operational without

being affected by failures in unrelated processes. These improvements have broadened MPI's appeal to a wider user base and they have enabled its adoption in fields such as data analytics, and real-time computing. By addressing usability challenges, MPI Sessions allow a larger portion of the computing community to benefit from MPI's performance advantages [9].

The MPI world model's global communicator `MPI_COMM_WORLD` makes error handling cumbersome. Failures in one process could propagate leading to the termination of the entire application. The introduction of MPI Sessions represents a significant architectural evolution in the MPI standard and they offer opportunities to manage errors more effectively [19]. Unlike the traditional world model where the global communicator `MPI_COMM_WORLD` inherently ties all processes together MPI Sessions allow for the modularization of process groups and communicators. This modularity creates the potential for errors to be isolated and handled within specific sessions rather than impacting the entire application. For example with session specific communicators failures can theoretically be confined to localized subsets of processes thus reducing disruption [9, 15].

Through modularity, dynamic process managability, broader applicability and enhanced error handling MPI Sessions address the core limitations of the traditional MPI world model. These advancements not only align with the evolving needs of modern parallel computing but also pave the way for more adaptable, efficient and reliable applications. While MPI Sessions address many of the longstanding limitations of the traditional MPI world model, they are not without their own challenges and trade-offs. The introduction of this new paradigm has solved critical issues but also introduced new complexities, some of which remain open questions within the MPI standard. The following section explores some of these challenges and will evaluate the areas where MPI Sessions fall short or introduce additional considerations.

4.3 Challenges and open questions in MPI Sessions

One of the feats of the MPI Sessions model is the reduction of overhead when initializing a session. The `MPI_COMM_WORLD` is no longer created here and therefore no overhead happens through this. However it is thinkable that MPI Sessions create their own performance overhead depending largely on the component their are used in. The session management is one of the potential sources of such overhead since building and destroying sessions incurs its own computational cost and each session requires its own metadata management and tracking. Basically a more dynamic session handling is also likely to lead to more administrative work for the MPI implementation. Apart from that additional overhead might also be produced during runtime, for example through switching between different sessions or through updating the communication context which is necessary for each session. Overall it is of cource important to note that the impact of this might greatly depend on the actual application [8, 9] (see example in Fig. 4).

Although it might seem like an obvious side effect of introducing a new concept in a programming standard a challenge worth noting is that an introduction like that inevitably brings new cognitive challenges for developers. While sessions offer more flexible communication contexts, they simultaneously require programmers to learn new conceptual frameworks, understand complex lifecycle management and develop different mental models for between process communication. This increased flexibility comes at the cost of a steeper learning curve which shows that there is an inherent trade-off between innovative programming tools and developer productivity one has to consider.

Another important point to make regards fault tolerance. As of MPI 4.1, the fault tolerance capabilities within MPI Sessions remain limited [19]. While the introduction of MPI Sessions provides a theoretical mechanism for more sophisticated error handling and resource management, the standard does not yet comprehensively address fault tolerance in terms of systematic recovery. MPI Sessions enable more granular error isolation by decoupling process management and allowing processes to operate within specific communicators, which potentially mitigates the risk of global failures

triggered by errors in individual processes. However robust fault tolerance in the traditional sense (such as automatic handling of network or node failures) has not been substantively implemented. Ongoing research and discussions within the MPI community are actively exploring strategies to improve fault tolerance with emerging proposals focusing on integrating more sophisticated error detection and recovery mechanisms within the MPI standard. At present, while MPI Sessions create a promising foundation for more effective error management, the specific methodologies for comprehensive fault recovery remain an unresolved challenge that has yet to be thoroughly addressed in the MPI standardization process [10, 19].

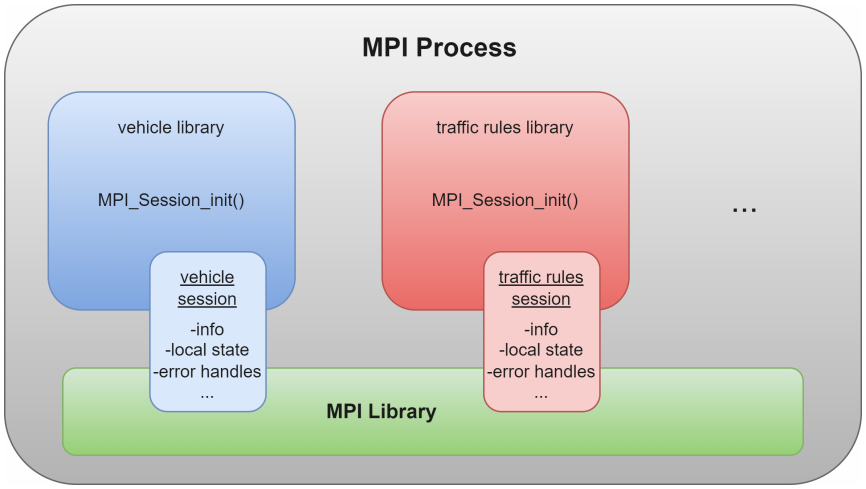

Fig. 4. An example of how MPI Sessions allows modularity within MPI processes. Every library can create their own session handles (light colored squares) to the MPI library through session initialization, adapted from [2].

Finally from an implementation perspective MPI Sessions introduce multilayered complexities that challenge existing library architectures. The dynamic nature of session creation, management and termination requires more sophisticated internal mechanisms compared to the static world communicator model. Library implementers must now design robust systems that can efficiently track and manage multiple concurrent communication contexts and handle complex lifecycle events while also being able to maintain performance guarantees across different session configurations. Compatibility presents an equally nuanced challenge. Migrating existing MPI-based applications which were developed under the traditional communication model to the session based approach poses a challenge. Developers face significant refactoring efforts that require careful translation of global communication patterns into session specific constructs. This transition is not merely syntactic but also conceptual and thus demands a fundamental rethinking of parallel communication strategies [8]. These implementation and compatibility challenges naturally lead us to examine how specific MPI implementations, such as OpenMPI, are addressing these complexities. The following chapter will delve into OpenMPI's approach to implementing the MPI Sessions model and their strategies for managing the intricate technical landscape created by this new communication paradigm will be explored.

5 Implementation in OpenMPI

As one of the most prominent implementations of the MPI standard OpenMPI combines high performance, scalability and flexibility through its innovative modular architecture and strong community driven development. Before exploring how OpenMPI incorporates the MPI Sessions paradigm, it is essential to understand the foundational principles and design choices that underpin its general implementation. OpenMPI is an open source project and is developed collaboratively by multiple institutions and consistently adopts the latest MPI standard revisions, including MPI standard 4.0 and 4.1. OpenMPI is broadly adopted across HPC applications and it offers cross platform functionality and optimization on performance in various architectures [4, 5, 17].

A defining feature of OpenMPI is its Modular Component Architecture (MCA), which allows for the dynamic loading of components at runtime. This architecture enables OpenMPI to support a variety of communication protocols (e.g., TCP/IP, InfiniBand, shared memory) and hardware architectures without requiring recompilation. The modularity also simplifies the integration of custom components, making OpenMPI highly extensible and adaptable to emerging technologies. To ensure optimal performance OpenMPI employs advanced memory management systems. Two of the key subsystems include the Memory Pool (MPool) which efficiently allocates memory for message passing and therefore reducing the overhead typically associated with dynamic memory allocation in performance-critical scenarios and the registration cache (RCache) which minimizes the cost of memory registration especially within networks like InfiniBand that require explicit registration of memory regions for remote direct memory access (RDMA) [10, 13]. OpenMPI also includes optimized point-to-point and collective communication algorithms and like that it can be ensured that it can take full advantage of hardware capabilities like multi-core processors, NUMA (non-uniform-memory-access) architectures and GPU accelerators [4, 7, 17].

OpenMPI's cross-platform design ensures it works across various environments, including differend operating systems. Its implementation in C avoids C++ dependencies and like that portability is enhanced and compatibility issues are reduced. This design choice has contributed to its widespread adoption as it easily integrates with existing toolchains and development workflows. Overall C, C++ and Fortran are supported languages but there are language bindings available for other languages like Java, Python and R. Furthermore OpenMPI dynamically detects and optimizes the use of hardware resources during initialization. For example it can determine the topology of the underlying system (e.g., NUMA nodes, network connections) and adjust communication patterns accordingly. This feature helps maximize performance while minimizing resource contention [5, 17]. An example of a simple MPI program, its output and how it can be run from the command line (once it has been compiled) is shown in Fig. 6, 7 and 5 respectively.

```
# Compile the program
mpicc -o example_mpi example_mpi.c
```

```
# Run the program
mpirun -np 4 ./example_mpi
```

Fig. 5. A command (on the left) that compiles the MPI program example_mpi and a command (on the right) that runs the previously compiled MPI program with 4 processes (as specified with the -np command), adapted from [20].

Through these features OpenMPI has established itself as a highly versatile and performance-oriented MPI implementation since it enables researchers and developers to push the boundaries of scientific discovery and industrial innovation. The next section will analyze how OpenMPI has dealt with the conecpt of MPI Sessions in particular and it will be examined what the implementation of MPI Sessions as formulated in the MPI standard looks like.

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    int world_rank, world_size;

    // Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // Get the rank of the process
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Print a message from each process
    printf("Hello from process %d out of %d!\n",
      world_rank, world_size);

    // Finalize the MPI environment
    MPI_Finalize();

    return 0;
}
```

Fig. 6. A simple MPI program that shows basic MPI commands and syntax, adapted from [20].

```
Hello from process 0 out of 4!
Hello from process 1 out of 4!
Hello from process 2 out of 4!
Hello from process 3 out of 4!
```

Fig. 7. The output of the above example MPI program when it is run with 4 processes.

5.1 OpenMPI and MPI Sessions

With the release of OpenMPI v5.0.x support for MPI Sessions has been introduced to OpenMPI aligning with the MPI standard's emphasis on greater flexibility and modularity in parallel application development. This implementation required notable changes to the internal architecture of OpenMPI to accommodate the decoupled initialization and resource management principles introduced by MPI Sessions. One key aspect of this integration is the support for multiple independent sessions within a single process that allows developers to leverage distinct communication contexts without having to rely on the global communicator `MPI_COMM_WORLD`. OpenMPI achieves this by redesigning its initialization routines to handle session based initialization independently. Moreover the library's runtime environment was updated to manage session specific resources efficiently ensuring compatibility with existing features while enabling the modularity and flexibility inherent to the MPI Sessions model. Additionally the OpenMPI developers incorporated mechanisms to dynamically create and manage communicators through session specific inquiries. These

update positions OpenMPI as a leading implementation that is actively driving the adoption of MPI 4.x features in both academic and industry contexts [17].

OpenMPI's implementation of MPI Sessions, introduced in version 5.0.x, represents a significant architectural evolution in the library's approach to communication contexts. This implementation aligns with the MPI 4.0 standard's move toward more flexible communication models. While the implementation is recent, OpenMPI's modular design has enabled this addition without disrupting existing functionality, ensuring that applications using traditional MPI constructs continue to work while providing a path forward for new applications leveraging MPI Sessions. The integration of MPI Sessions into OpenMPI's Modular Component Architecture (MCA) showcases the flexibility of OpenMPI's design philosophy. The implementation leverages several key OpenMPI subsystems, including the Process Management Interface (PMIx) for process coordination, the Byte Transfer Layer (BTL) for underlying communication and the Process Management Layer (PML) for message passing. These existing components have been extended or modified to support the MPI Sessions model while maintaining their original functionality [8, 10, 17]. From a technical perspective OpenMPI implements MPI Sessions through sophisticated mechanisms for creation, management and termination. Session creation initializes session-specific contexts, establishes resource tracking mechanisms and sets up communication endpoints. The process set management system maintains dynamic mappings of processes and handles group creation and modification, while resource management handles allocation, tracking and cleanup of session-specific resources. The current feature coverage in OpenMPI's MPI Sessions implementation includes basic session initialization and termination, process set management, group creation within sessions and fundamental error handling mechanisms. However development continues on advanced features such as comprehensive fault tolerance mechanisms, dynamic process management within sessions and extended error recovery options [17, 19]. This ongoing development reflects the evolutionary nature of the MPI Sessions implementation. Performance considerations play a crucial role in OpenMPI's implementation. The design focuses on reducing initialization overhead compared to the traditional `MPI_INIT` adn therefore optimizing runtime performance through efficient context switching between sessions and managing memory usage through session-specific allocation and resource sharing where appropriate [10]. These optimizations aim to maintain the high-performance characteristics expected of MPI implementations while providing the additional flexibility of the MPI Sessions model. For practical usage OpenMPI 5.0.x provides configuration options through environment variables and runtime parameters. Developers need to adapt their code to use the MPI Sessions API that replaces traditional MPI initialization calls with session-specific alternatives. The implementation maintains backward compatibility while establishing new patterns for session-based parallel applications. This includes considerations for session-specific error handling, resource management strategies, and process group management [17]. A simple example for a MPI Sessions program can be seen in Fig. 8.

The implementation continues to evolve, with ongoing work focused on improving performance characteristics, adding new features, enhancing debugging and profiling tools and extending error handling capabilities. The very latest OpenMPI v5.0.6 introduces several notable enhancements to its functionality and performance. In session management improvements include better handling of multiple initializations and finalizations, proper finalization of classes and enhanced support for session-based initialization models in line withMPI 4.x standards. The release also addresses key issues in CUDA compatibility, such as handling stream-ordered allocations and virtual memory manager pointers and adds support for NVIDIA's nvfortran compiler. For MPI communication enhancements to the `MPI_Info_dup` function ensure consistent key management, while resource leak fixes and improved locking protocols bolster stability and memory efficiency. Apart from that the `MPI_T` interface has been refined to enable dynamic tuning of collectives at runtime, offering greater flexibility for applications. These updates underscore OpenMPI's commitment to advancing

stability, compatibility, and performance in modern HPC environments [1, 17]. Overall this evolutionary approach ensures that OpenMPI's MPI Sessions implementation can grow to meet the needs of modern parallel applications while maintaining the reliability and performance that users expect from the OpenMPI library.

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
  MPI_Group process_group;
  MPI_Comm process_comm;
  int process_rank, process_size;

  // Initialize the MPI environment
  MPI_Session session;
  MPI_Info info = MPI_INFO_NULL;

  MPI_Session_init(info, MPI_ERRORS_RETURN, &session);

  // Get the default group from the session
  MPI_Group_from_session_pset(session, "mpi://WORLD", &process_group);

  // Create a communicator for the process group
  MPI_Comm_create_from_group(process_group, "process_comm",
          MPI_INFO_NULL, MPI_ERRORS_RETURN, &process_comm);

  // Get the rank of the process
  MPI_Comm_rank(process_comm, &process_rank);

  // Get the size of the communicator
  MPI_Comm_size(process_comm, &process_size);

  // Print a message from each process
  printf("Hello from process %d out of %d!\n", process_rank, process_size);

  // Free resources and finalize the session
  MPI_Group_free(&process_group);
  MPI_Comm_free(&process_comm);
  MPI_Session_finalize(&session);

  return 0;
}
```

Fig. 8. Using OpenMPI, this program demonstrates the initialization and usage of an MPI session to create a communicator and manage a group of processes.

5.2 Challenges and insights from OpenMPIs implementation

The implementation of MPI Sessions in OpenMPI represents a significant step forward in enhancing the modularity and scalability of MPI applications. However this transition also introduced a series of challenges that required innovative solutions and highlighted opportunities for further refinement.

One of the most significant challenges was the shift from the world communicator model to the MPI Sessions paradigm. Unlike the traditional model, which assumes the existence of a global communicator (`MPI_COMM_WORLD`), the MPI Sessions paradigm required OpenMPI to support dynamic process discovery and manage communicators independently. In traditional implementations the `add_procs` mechanism would query the runtime to discover all MPI processes during initialization, resulting in substantial memory overhead and high startup costs, especially in large-scale applications. To address this OpenMPI developers modified the `add_procs` mechanism to limit its scope to node-local processes, deferring the discovery of remote processes until they are required for communication. This change not only reduced the startup cost but also aligned the discovery process with the dynamic nature of MPI Sessions, where processes are managed more flexibly. Despite these improvements this shift introduced complexities in maintaining consistency across dynamic process sets particularly in scenarios involving frequent or irregular communication patterns [8, 15].

The management of communicator identifiers (CIDs) presented another major hurdle. In the traditional model, CIDs are derived from a parent communicator using a consensus algorithm that ensures consistency across all participating processes. The MPI Sessions paradigm, however, lacks predefined parent communicators and because of that a completely new approach to CID generation becomes necessary. OpenMPI addressed this by introducing an extended CID (exCID) system [8]. This system combines a 64-bit process group identifier (PGCID) from the PMIx runtime with additional subfields for derived communicators, allowing unique identifiers to be generated without relying on predefined structures. While this approach preserved backward compatibility with existing MPI workflows, it also introduced performance trade-offs. For example, the use of PMIx group APIs to generate PGCIDs involves inter-node communication, which can be relatively slow [8, 10]. Additionally the requirement to maintain compatibility with optimized tag-matching mechanisms in point-to-point messaging cmponents (PMLs) necessitated careful design to avoid performance degradation. These trade-offs highlight the challenges of balancing innovation with the need to maintain efficiency in high-performance computing environments.

Dynamic initialization and resource management posed further challenges, particularly in supporting the ability to initialize and finalize MPI Sessions multiple times within a single application execution. In traditional MPI implementations initialization routines are invoked once with all resources allocated at startup and released during finalization. This approach is incompatible with the MPI Sessions paradigm, which demands more granular and dynamic resource management. To adapt OpenMPI restructured its resource management systems to initialize subsystems incrementally as needed and track their usage with reference counts. Cleanup operations are managed through a callback framework and that ensures that resources are only released when they are no longer in use. This restructuring not only enabled the flexibility required by MPI Sessions but also improved resource efficiency by avoiding unnecessary allocations and deallocations. However, the added complexity of managing resources dynamically introduced challenges in ensuring consistent behavior across different execution contexts, particularly in applications with varying levels of concurrency or dynamic process management [8, 10, 15].

Supporting MPI functions before initialization added complexity, particularly in ensuring thread safety for operations on `MPI_Info` objects, error handlers and session attributes. The MPI Sessions proposal allows these operations to

be invoked before any initialization function (e.g., `MPI_Init` or `MPI_Session_init`) is called, requiring OpenMPI to implement thread-safe mechanisms that are always active. This contrasts with traditional implementations, where thread safety measures are only enabled after the initialization routine specifies the required thread level [8]. To address this OpenMPI developers ensured that locks for these objects are permanently enabled and that contributed in bringing robustness to pre-initialization operations. While this approach guarantees correctness, it also introduces slight overheads in scenarios where thread safety is not strictly required. Additionally maintaining consistency across a wide range of pre-initialization operations posed significant implementation challenges, particularly in ensuring compatibility with existing MPI components that were not originally designed for such usage patterns [18].

Integrating the MPI Sessions model with existing components such as the point-to-point message layer (PML), required substantial modifications. For example some PML components had to be updated to handle the new exCID system and like that consistent communicator identifiers across processes could be ensured while maintaining optimized matching support for existing workflows [8, 10]. The introduction of exCID required additional message headers to support identifier matching and because of that changes to both sending and receiving logic were necessary. To preserve performance for existing applications the implementation ensured that these modifications do not interfere with the optimized tag matching mechanisms used in the traditional MPI world model. However extending these updates to more PML components remains an area of ongoing work, as not all components currently support the new exCID structure.

These challenges underscore the complexity of implementing MPI Sessions within a mature library like OpenMPI. Despite these obstacles the process yielded valuable insights into both the technical intricacies and the broader implications of the MPI Sessions paradigm. For instance the extended CID mechanism demonstrated the importance of designing scalable and flexible identifier systems, while the restructuring of initialization routines highlighted the need for dynamic resource management in modern MPI implementations. Additionally the integration of PMIx group APIs provided a robust foundation for dynamic process management even with performance trade-offs that require further optimization.

While these efforts represent substantial progress in adapting OpenMPI to the MPI Sessions paradigm, they also highlight that these developments are still being tested and extended. As implementations mature and usage expands new challenges and unforeseen limitations are likely to surface and they can pave the way for further research and innovation.

6 Conclusion and outlook

The introduction of MPI Sessions marks a significant milestone in the evolution of the Message Passing Interface addressing key limitations of the traditional world model and offering a more modular and flexible approach to resource management and communication. By decoupling initialization from application-wide communicators and enabling independent communication contexts, MPI Sessions provide a framework better suited to the needs of modern parallel computing, especially in exascale systems. These advancements offer practical benefits, such as improved scalability and dynamic resource allocation, while also presenting new challenges for both developers and MPI library implementers.

The implementation of MPI Sessions in OpenMPI, introduced with version 5.0.x, illustrates how this new paradigm is being realized in practice. OpenMPI's adoption of MPI Sessions demonstrates its commitment to staying at the forefront of MPI standards, while highlighting the complexities involved in transitioning to this new model. These include challenges related to session management, metadata tracking and ensuring compatibility with existing MPI-based

applications. Despite these hurdles, OpenMPI's implementation represents a critical step toward integrating MPI Sessions into production-level systems and provides a foundation for further refinement and optimization.

At the same time several unresolved questions remain. Fault tolerance for example has yet to be fully addressed within the context of MPI Sessions. While the framework offers opportunities for more granular error handling and recovery, robust mechanisms for systematic failover and fault recovery are still under development. These issues underscore the need for ongoing research and collaboration within the MPI community to address the complexities and maximize the potential of MPI Sessions.

Looking ahead the development of future MPI standards such as a potential MPI 5.0 is expected to further refine the MPI Sessions model and address existing limitations. Ongoing work in optimizing session management, reducing performance overhead and integrating advanced fault-tolerance mechanisms will play a crucial role in achieving these goals. The continued evolution of implementations like OpenMPI will also contribute valuable insights into how MPI Sessions can be effectively integrated into diverse applications and environments.

In conclusion MPI Sessions represent a promising direction for the future of parallel programming, enabling greater flexibility and scalability in a rapidly advancing field. Their adoption by major implementations like OpenMPI signals a broader shift toward embracing modular and dynamic approaches in distributed computing. As research and development in this area continue MPI Sessions have the potential to become a cornerstone of high-performance computing and through them addressing the challenges of tomorrow's exascale systems are addressed while paving the way for even more innovative communication paradigms.

References

[1] Noah Evans, Jan Ciesko, Stephen L. Olivier, Howard Pritchard, Shintaro Iwasaki, Ken Raffenetti, and Pavan Balaji. 2020. Implementing Flexible Threading Support in Open MPI. In *2020 Workshop on Exascale MPI (ExaMPI)*. IEEE, Albuquerque, NM, USA, 21–30. https://doi.org/10.1109/ExaMPI52011.2020.00008

[2] Jeff Squyres (Cisco Systems) for insideHPC report YouTube channel. 2016. *How to Make MPI Awesome: A Proposal for MPI Sessions.* Youtube. https://www.youtube.com/watch?v=ItHxgOHPCJg Accessed: 2024-12-10.

[3] MPI Forum. 2024. MPI Forum: The Message Passing Interface Standard. https://www.mpi-forum.org/ Accessed: 2024-12-03.

[4] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*. Springer, www.open-mpi.org, Budapest, Hungary, 97–104.

[5] Richard L. Graham, Galen M. Shipman, Brian W. Barrett, Ralph H. Castain, George Bosilca, and Andrew Lumsdaine. 2006. Open MPI: A High-Performance, Heterogeneous MPI. In *2006 IEEE International Conference on Cluster Computing*. IEEE, Albuquerque, NM, USA, 1–9. https://doi.org/10.1109/CLUSTR.2006.311904

[6] F. Gygi, R.K. Yates, J. Lorenz, E.W. Draeger, F. Franchetti, C.W. Ueberhuber, B.R. de Supinski, S. Kral, J.A. Gunnels, and J.C. Sexton. 2005. Large-Scale First-Principles Molecular Dynamics simulations on the BlueGene/L Platform using the Qbox code. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE, Albuquerque, NM, USA, 24–24. https://doi.org/10.1109/SC.2005.40

[7] Nathan Hjelm, Matthew G. F. Dosanjh, Ryan E. Grant, Taylor Groves, Patrick Bridges, and Dorian Arnold. 2018. Improving MPI Multi-threaded RMA Communication Performance. In *Proceedings of the 47th International Conference on Parallel Processing* (Eugene, OR, USA) *(ICPP '18)*. Association for Computing Machinery, New York, NY, USA, Article 58, 11 pages. https://doi.org/10.1145/3225058.3225114

[8] Nathan Hjelm, Howard Pritchard, Samuel K. Gutiérrez, Daniel J. Holmes, Ralph Castain, and Anthony Skjellum. 2019. MPI Sessions: Evaluation of an Implementation in Open MPI. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Albuquerque, NM, USA, 1–11. https://doi.org/10.1109/CLUSTER.2019.8891002

[9] Daniel Holmes, Kathryn Mohror, Ryan E. Grant, Anthony Skjellum, Martin Schulz, Wesley Bland, and Jeffrey M. Squyres. 2016. MPI Sessions: Leveraging Runtime Infrastructure to Increase Scalability of Applications at Exascale. In *Proceedings of the 23rd European MPI Users' Group Meeting* (Edinburgh, United Kingdom) *(EuroMPI '16)*. Association for Computing Machinery, New York, NY, USA, 121–129. https://doi.org/10.1145/2966884.2966915

[10] Dominik Huber, Maximilian Streubel, Isaías Comprés, Martin Schulz, Martin Schreiber, and Howard Pritchard. 2022. Towards Dynamic Resource Management with MPI Sessions and PMIx. In *Proceedings of the 29th European MPI Users' Group Meeting* (Chattanooga, TN, USA) *(EuroMPI/USA '22)*.

Association for Computing Machinery, New York, NY, USA, 57–67. https://doi.org/10.1145/3555819.3555856

[11] Salvatore Orlando Jack Dongarra, Domenico Laforenza (Ed.). 2003. *Recent Advances in Parallel Virtual Machine and Message Passing Interface.* Springer, Heidelberg, Germany. https://doi.org/10.1007/b14070 Proceedings of the 9th European PVM/MPI Users' Group Meeting, Linz, Austria, September 29–October 2, 2002.

[12] Humaira Kamal, Seyed M Mirtaheri, and Alan Wagner. 2010. Scalability of communicators and groups in MPI. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. Association for Computing Machinery, New York, NY, USA, 264–275.

[13] Alexander Margolin and Amnon Barak. 2019. RDMA-Based Library for Collective Operations in MPI. In *2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI)*. IEEE, Albuquerque, NM, USA, 39–46. https://doi.org/10.1109/ExaMPI49596.2019.00010

[14] Message Passing Interface Forum 2021. *MPI: A Message-Passing Interface Standard Version 4.0.* Message Passing Interface Forum. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[15] Message Passing Interface Forum 2023. *MPI: A Message-Passing Interface Standard Version 4.1.* Message Passing Interface Forum. https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf

[16] Adam Moody, Dong H. Ahn, and Bronis R. de Supinski. 2011. Exascale algorithms for generalized MPI_comm_split. In *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface* (Santorini, Greece) *(EuroMPI'11)*. Springer-Verlag, Berlin, Heidelberg, 9–18.

[17] OpenMPI Project 2024. *OpenMPI v5.0.x Documentation.* OpenMPI Project. https://docs.open-mpi.org/en/v5.0.x/

[18] Thananon Patinyasakdikul, David Eberius, George Bosilca, and Nathan Hjelm. 2019. Give MPI Threading a Fair Chance: A Study of Multithreaded MPI Designs. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, Albuquerque, NM, USA, 1–11. https://doi.org/10.1109/CLUSTER.2019.8891015

[19] Roberto Rocco, Gianluca Palermo, and Daniele Gregori. 2023. Fault Awareness in the MPI 4.0 Session Model. In *Proceedings of the 20th ACM International Conference on Computing Frontiers* (Bologna, Italy) *(CF '23)*. Association for Computing Machinery, New York, NY, USA, 189–192. https://doi.org/10.1145/3587135.3592189

[20] Patricio Bulić Roman Trobec, Boštjan Slivnik and Borut Robič. 2018. *Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms* (1 ed.). Springer Cham, Switzerland. XII, 256 pages. https://doi.org/10.1007/978-3-319-98833-7