POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Critical Path Analysis for MPI Applications with Trace Compass

FAWAZ JUMAAH

Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de Maîtrise ès sciences appliquées

Génie informatique

Décembre 2023

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Ce mémoire intitulé :

Critical Path Analysis for MPI Applications with Trace Compass

présenté par **Fawaz JUMAAH**

en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

a été dûment accepté par le jury d'examen constitué de :

Guillaume-Alexandre BILODEAU, président

Michel DAGENAIS, membre et directeur de recherche

Maxime LAMOTHE, membre

DEDICATION

"And mankind has not been given of knowledge except a little."

Holy Qur'an / Al-Israa (85)

To the memory of my father, whether he is dead or still alive …

To my mother, the beautiful smile in my life…

To my wife, the soul mate of my heart…

To my daughter, the happiness of my dreams …

To my beloved friends for their encouragement and help

Thank you for being there with me …

In the memory of my sister Sura, forever in my heart …

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my deepest thanks and appreciation to my supervisor, Prof. Dr. Michel Dagenais, for being a great supervisor, motivator as well as teacher throughout the long journey it took to complete this research.

My special thanks to my colleagues at DORSAL lab for their dedicated and valuable support. They have helped me during my research journey in scientific and mental ways. Special thanks to Arnaud Fiorini for his endless support to complete this study. He was always there ready to help.

Furthermore, I would like to praise Ecole Polytechnique de Montreal and the department of Computer and Software Engineering for their support during my study journey. It was a great opportunity to be a student there.

My special and hearty gratitude goes to my family, especially my mother and my lovely wife, for their unreserved love and support. My parents sacrificed their lives since I was child until now to raise me up. Nothing I can say or do that can reward you for what you have done. My beloved wife was beside me throughout my study facing many hard moments. I really appreciate your patience, dedication, and true support.

RÉSUMÉ

Avec la croissance rapide des systèmes multicœurs dans les supercalculateurs modernes, il devient essentiel de mettre en place des applications évolutives et les outils nécessaires pour les développer. Les applications modernes évoluent vers la complexité, ce qui rend les outils de profilage et de débogage traditionnels difficiles à utiliser. En conséquence, les performances des programmes parallèles deviennent difficiles à déboguer et à optimiser. De plus, l'efficacité des systèmes à grande échelle est limitée par l'intercommunication entre les nœuds. Pour comprendre la cause profonde de ces retards, une analyse de traçage est nécessaire. Cette étude propose un cadre pour les calculs du chemin critique des communications Message Passing Interface (MPI) basé sur l'application Trace Compass.

Le cadre proposé a été développé en deux étapes, premièrement, la mise en œuvre de l'algorithme de graphe d'exécution MPI et l'intégration avec l'environnement de développement de l'incubateur Trace Compass, deuxièmement, l'extension de l'algorithme générique de chemin critique au niveau du noyau dans Trace Compass pour tenir en compte les intervalles du graphique d'exécution MPI associés et calculer le chemin critique.

Le cadre proposé a été testé et validé à l'aide de cinq programmes MPI allant d'applications simples, exécutées sur deux nœuds de processeurs, à une application complexe, fonctionnant sur cinquante cœurs de traitement avec des communications intensives entre les cœurs.

ABSTRACT

With the rapid growth of multi-core systems in modern supercomputers, it becomes essential to establish scalable applications and the required tools to develop them. Modern applications are moving towards complexity, which makes the traditional profiling and debugging tools hard to use. As a result, the performance of parallel programs becomes difficult to debug and optimize. Furthermore, the efficiency of large-scale systems is constrained by the intercommunication between nodes. To understand the root cause of these delays, tracing analysis is needed. This study proposes a framework for critical path computations of Message Passing Interface (MPI) communications based on the Trace Compass application.

The proposed framework was developed in two stages, first, the implementation of the MPI execution graph algorithm and the integration with the Trace Compass incubator developer environment, second, the extension of the generic kernel-level critical path algorithm in Trace Compass to take into account the related MPI execution graph intervals and compute the critical path.

The proposed framework was tested and validated using five MPI programs scaled from simple applications, running on two processors nodes, to a complex application that runs on fifty processing cores with intensive communications between cores.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
AST	Abstract Syntax Tree
CTF	Common Trace Format
CUDA	Compute Unified Device Architecture
HPC	High-Performance Computing
LTTng	Linux Trace Toolkit: next generation
MPI	Message Passing Interface
OTF2	Open Trace Format version 2
PAG	Program Activity Graph
ROCm	Radeon Open Compute platform
SAST	Static Application Security Testing
TSDL	Trace Stream Description Language
UI	User Interface
UST	User Space Tracing

CHAPTER 1 INTRODUCTION

1.1 Performance Analysis Challenges

Parallel programs that communicate through the Message Passing Interface library (MPI) represent the prevalent High-Performance Computing (HPC) applications, to tackle a variety of complex and challenging problems. In such large-scale applications, which are running on clusters of multi-core computers, it is challenging to debug, optimize, and mitigate the performance issues. The performance analysis tools developed for this purpose can be categorized into profiling and tracing. Profiling tools provide an abstracted performance fingerprint of an application. These tools involve information about the execution time of the code sections of an application. On the other hand, tracing tools provide recorded events activities with timestamps, which allow software engineers to investigate the application performance in more details. These events are visualized in the form of charts, execution graphs, and tables. Tracing is a method that involves the use of an instrumentation tool to capture low-level events within a system. During the events recording, the software developer can set specific instrumentation points in the application source code and store the recorded events in a trace file. After that, the developer can use trace viewers and analyzers to investigate the source of performance issues and bottlenecks during the application execution.

Through a proper set of profiling tools, the developer can collect some information about the application performance during its execution. However, this information might not be sufficient to fully understand the system behavior. Similarly, it is challenging to use a debugger within such system environments, due to the multiple communicating cores. As a result, tracing is needed to overcome these issues [1].

In this chapter, we present the basic definitions and concepts used throughout the study. The problem statement of this research is defined. Furthermore, the objectives of this study are presented. In addition, the research contributions are illustrated in terms of the MPI critical path computation in Trace Compass, and its validation.

1.2 Definitions and Basic Concepts

1- MPI: MPI is a library of calls that can be used for developing parallel programs in C/C++. It provides models and approaches for messages passing between the available processor cores in an efficient, practical, and portable manner.

2- Instrumentation: it refers to the ability of measuring and monitoring program performance and debugging issues when executing the program. It provides the developer with the required tools and libraries to debug, profile, and trace programs to achieve design goals and mitigate performance issues.

3- Tracing: it is a technique used to monitor the execution of a program running on a system, and understand its behavior, by recording the events and calls that occur during the program execution. Through tracing, software engineers can detect performance issues and analyze it for better system performance and optimization. By adding the proper instrumentation to the program code, the developer can record trace events, and analyze them through trace analysis tools.

4- Profiling: is a technique that can be used to measure the performance of an application. This can be achieved by recording performance counters, frequency, memory utilization, function calls, etc …

5- Debugging: it is a technique used to analyze programs during the execution to distinguish their behavior. It aids the developers to identify the underlying causes of program failures and flaws.

6- Trace Compass: is an open-source tool used for identifying performance issues via analyzing the trace files of a program or system. It provides the necessary analysis tools, graphs, diagrams, and metrics that help the software engineers to obtain useful information from the trace and log files.

7- Score-P: it is an instrumentation set of libraries that can be used for profiling and tracing programs while executing them on hardware. By applying the required triggers and tools, Score-P can record trace files after execution completion. These trace files can be imported into trace analysis tools such as Trace Compass, TAU, and Vampir.

8- Trace Event: it is an event that represents the execution of a code location in a program. This event can be a system call, input, output, interrupt, memory access, or network activities.

9- Execution graph: it is the program control flow that occurs during the execution. It represents a set of events and calls from the beginning till the end of the program. This graph is important since it is used to compute the critical path through the processing cores or threads.

10- Critical Path: it is one of the essential performance characteristics of parallel programs. It represents the longest sequence of execution without any waiting time.

1.3 Problem Statement

In modern multi-core systems, that involve multiple processes, there is an increase in the difficulty of optimizing and debugging the applications as the technology grows. With such parallel programs, it is required to use efficient message passing protocols to achieve optimum performance and resource utilization. However, in MPI applications, which involve multiple cores, delays in peer-to-peer, root-to-all, and all-to-root communications have a significant impact on critical path [2]. Therefore, the efficiency of large-scale systems is often constrained by this intercommunications between cores [3]. To understand the root cause of these delays, trace analysis is needed. It is therefore important to have MPI performance analysis tools to help the HPC applications developers to improve their codes [4].

The critical path is an essential application characteristic, which refers to the longest sequence of operations in a system that can be executed without any interruption or waiting time [5]. In simpler terms, the critical path signifies the time required to finish a parallel application task, considering the execution on the targeted process itself, but replacing any waiting delay by what we are waiting for, from other processes possibly on other processor cores. Identifying the critical path of parallel applications will help the developers to define the performance optimization possibilities in their applications. Furthermore, it provides more details on the application behavior and its communication, with respect to scaling and performance changes, which will help in identifying the bottlenecks of the application performance. In addition, the number of available processor cores might reach in the hundreds, which makes it impossible to manually analyze the critical path. As a result, it is required to have tracing analysis tools to analyze the MPI critical path and visualize the results.

The purpose of this research is to extend the critical path algorithms in order to take into account the semantics of the MPI communications, both point-to-point and global. This can be achieved

through the development of an MPI execution graph module. It will extend the available Linux operating system critical path algorithm of Trace Compass, to compute the critical path of MPI applications. This framework uses the Trace Compass libraries to read the MPI trace files in CTF format, then extract the required information and visualize the critical path via the Trace Compass User Interface (UI) modules.

1.4 Research Objectives

This study focuses on the following objectives to come up with the proposed MPI critical path framework:

1- Investigate the MPI tracing tools and analyzers to survey the state of the art and identify gaps.
2- Derive a new critical path computation framework, for MPI applications, based on Trace Compass.
3- Validate the proposed extended algorithm, integrating it within the Trace Compass framework, and validate it with trace files from representative MPI applications.
4- Discuss the results of the proposed framework when used on representative MPI applications.

1.5 Research Contributions

The primary achievement of this research is to extend the existing critical algorithm to MPI communications, and develop and implement a framework for analyzing the critical path of MPI applications. This framework includes the algorithms needed to handle MPI trace files and to create the MPI execution graph. Furthermore, the available generic critical path algorithm was extended to analyze MPI events, which are different from the Linux operating system events and calls. Finally, the derived algorithms are integrated with the Trace Compass incubator environment for validation and testing.

1.6 Thesis Outline

Chapter 2 presents the latest tools, algorithms, and methods used to create MPI execution graph and critical path computation methods. It examines the latest methods and frameworks used to analyze and compute the critical path of the MPI applications.

Chapter 3 demonstrates the proposed framework methodology to achieve the research objectives. Through this chapter, the details of the proposed MPI execution graph algorithm, and its implementation, are presented. Furthermore, the extension of the available generic kernel-level Linux operating system critical path algorithm is discussed.

Chapter 4 presents the article “Multi-level Critical Path Analysis Framework for MPI Applications“. This article shows the details of the research contributions, proposed methodology, and the validation results.

Chapter 5 provides a general discussion and revisits the research milestones. Furthermore, the limitations of this research are presented.

Chapter 6 summarizes the study conclusions, contributions, and future work.

CHAPTER 2 LITERATURE REVIEW

This chapter presents an overview of the current MPI tracing tools and algorithms found in related work. We survey the characteristics of the available algorithms and frameworks to solve similar problems. We then discuss the strengths and weaknesses of each, and identify the remaining gaps in the performance analysis of MPI applications.

2.1 Message Passing Interface (MPI)

MPI is a standard programming paradigm that enables communication among processes and nodes deployed in distributed systems. It offers an Application Programming Interface (API) that clearly outlines the syntax and comprehensive semantics of a software library, serving as a uniform set of procedures for building complex programs [6]. The international MPI initiative was established in 1992 by Oak Ridge National Laboratory and the parallel computing center at Rice university. The MPI standard was introduced in 1994 by the MPI Forum. The major target of this initiative was to determine an interface for message passing that can be used efficiently in parallel and distributed computing systems [7]. The main function of MPI is to connect processes (or endpoints) while performing synchronization and data exchanges through message passing. An MPI program consists of a series of procedures that initialize the processes based on the available number of nodes in the multi-core system, thereby facilitating communication among them. All of the available nodes in the distributed system are running the same program, but with different data sets [8].

MPI is not a library implementation, but rather an Application Programming Interface (API) with also characteristics that the library should embody to ensure it is scalable, efficient, and portable. Several initiatives have been undertaken to implement the MPI standard, including OpenMPI, Intel-MPI, MPICH, and MVAPICH2. These are some examples of open-source implementations of MPI. MPICH, developed by Argonne National Laboratory, acts as a benchmark implementation of MPI, serving as the source code basis for other versions such as MVAPICH. The key difference between MVAPICH and MPICH lies in the fact that MVAPICH is specifically tailored for high-speed interconnection networks such as iWARP and InfiniBand. Open MPI is indeed another popular MPI library based on an open-source implementation. It is widely utilized by numerous

supercomputers listed in the Top500.org list of supercomputers, further demonstrating its effectiveness and reliability. IBM Spectrum MPI, Intel MPI, and Cray MPI are commercial implementations of the MPI standard. These versions are typically optimized for specific hardware or use cases, offering enhanced performance and additional features compared to open-source implementations. For a job to be executed in MPI, it is necessary for the processes executing on the different cores and nodes to have the ability to interact with each other. This implies that data must be moved from the memory space of one process to the memory space of another process. This transfer is crucial for efficient inter-process communication and data sharing [2].

MPI is compatible with the Compute Unified Device Architecture (CUDA), a parallel programming platform for Graphics Processing Units (GPU), which was introduced by NVIDIA. GPU-Aware MPI was introduced by integrating the classical MPI calls with the CUDA platform, to enable data communications among GPUs, without passing through CPU host [9]. CUDA-Aware MPI can be adopted to pass GPU memory pointers to MPI calls, without explicit handling of CPU-GPU data movements at the application level [10]. The node in CUDA-Aware MPI is defined as a common node, if two or more subdomains of a structure are sharing it. MPI communications can be implemented using the same MPI calls but with GPU buffers instead of host buffers [9].

2.2 MPI Communication Types

MPI supports two types of communications, namely: point-to-point (or two-sided) communications, and collective (or group) communications. Two concepts were established in MPI to handle these communications: groups and communicators. These concepts assist the developers to identify the context and the scope of communications. A group consists of a sequence of processes or threads, each linked with a unique processor node known as rank, ranging from 1 to *N*. When initializing an MPI program, the system assigns these unique rank numbers to each processor node. The communicator is a set of groups that are participating in each communication. MPI allows the developers to talk to each rank within the same communicator in terms of sending or receiving data. The following sections describe each MPI communication type in details.

2.2.1 Point-to-Point Communications

The Point-to-point communication approach is a fundamental method of interaction, where a message is transmitted from a sender to a recipient, with the order of messages being preserved. The sender uses a buffer data structure to hold the message, with an envelop containing information utilized by the receiver side. This information helps the receiver node to select the required message and store it in a buffer. An example of point-to-point communication can be seen in Figure 2.1. In this case, it is the responsibility of the MPI implementation to determine how data is handled. Usually, a dedicated buffer area within the system is set in the destination process to store data that is in the process of being transferred.

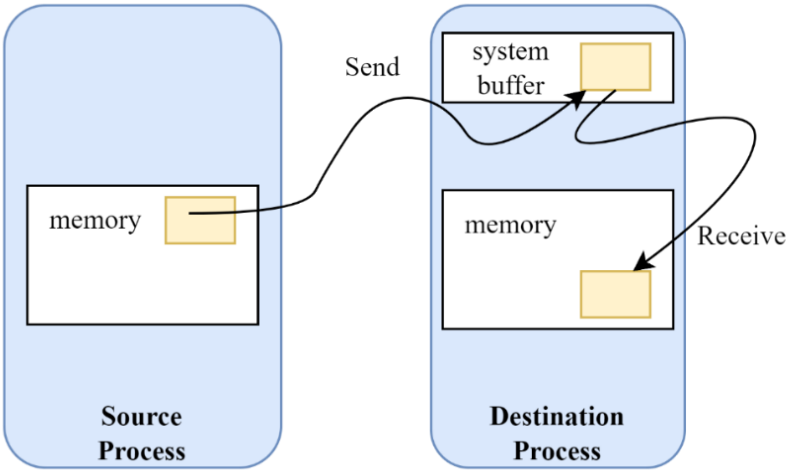


Figure 2.1 MPI point-to-point communication example

Point-to-point communication can be facilitated through two distinct methodologies: blocking and non-blocking. The commonly used APIs for send operations in MPI point-to-point communications are: mpi_send and mpi_isend, while the receive operations APIs are: mpi_recv and mpi_irecv.

When using the blocking approach, the calling process must be blocked until it is able to use the sender buffer again. On the receiver side, the receiver is blocked until the communication operation is completed. In the non-blocking approach, the operation of the sender resumes once the data has been successfully copied into the sender buffer. Concurrently, the receiver operation returns as soon as the request to receive messages has been posted. As a result, the non-blocking approach is useful, since it provides the ability to overlap communications with other processing for hardware

parallelism, at the cost of further waiting and polling techniques to verify the communication completion [11].

Non-blocking point-to-point communications are used in scientific programs. This approach offers the possibility of enhancing performance on numerous systems by allowing simultaneous computation and communication. However, achieving this overlap is challenging, due to constraints within the MPI internal progress engine and the network operation. The traditional solution to this issue involves the development of an asynchronous-progress engine, which relies on either extended hardware interrupts or threads [12]. Andreas proposed a model to enhance the quantitative point-to-point models via a novel methodology which involves both heterogeneity and contention [13]. Another approach was proposed by Min Si to allow developers to use the PMPI Casper framework to manage point-to-point communications using a process-based asynchronous progress approach. This approach can be applied to an indefinite number of cores within a multicore architecture [12].

2.2.2 Collective Communications

The collective communication mechanism is an MPI communication approach where the communication messages can be exchanged among a communication group of ranks, instead of two. This allows the processes to communicate in root-to-all, all-to-root, and all-to-all topologies. Common MPI examples for collective communications are barrier, allgather, broadcast, and allreduce. Collective operations are frequently used by MPI application developers. They often have a significant impact on the application performance [2]. An example of collective communication can be seen in Figure 2.2.

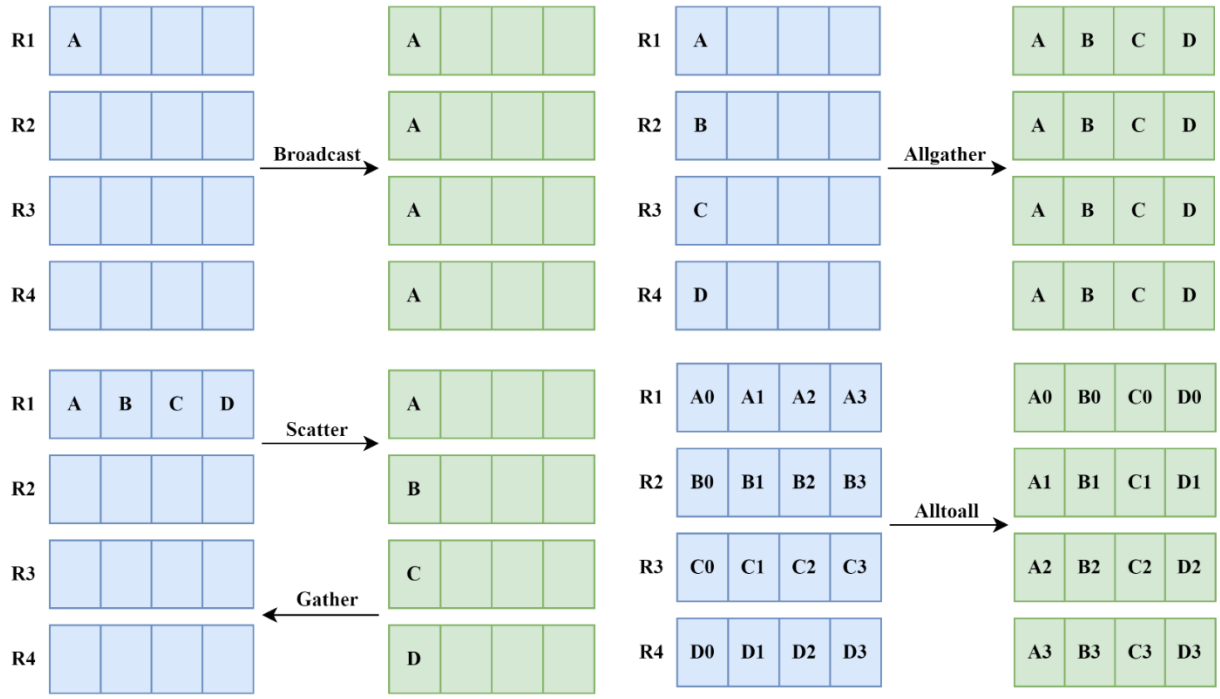


Figure 2.2 MPI collective communication

One of the standard collective communication methods is called Broadcast. In this approach, a single root process disseminates identical data to all processes within the same communicator. This is commonly seen when configuration parameters need to be distributed to all system processes, or when user input data must be transmitted to a parallel program. While the MPI Scatter method and Broadcast might appear similar, they have a crucial difference. Broadcast sends the same data to all processes, whereas Scatter distributes different parts of an array to various processes. On the other hand, MPI Gather operates differently by collecting data from multiple processes and consolidating it into one. This technique is particularly useful in applications that require parallel searching and sorting. Another collective communication method is called Allgather, where all the data elements are gathered to all the available processes. In other words, Allgather functions akin to MPI gather followed by MPI broadcast.

The MPI libraries have a collection of algorithms that allow the developer to use the required MPI collective operation [14]. Collective operations are recognized for both standard and connected communicators using the conventional interfaces, such as MPI_Bcast. Communicators with associated virtual topologies and intra-communicators are distinguished using specific, tangible interfaces, such as MPI_Neighbor_allgather [15]. Collective operations were expanded in MPI 4.0

to comprise persistent collectives; this enables the developer to get the benefit of repeating the same collective operation with new data, for better data movement optimization [16].

2.3 Performance Analysis

The term “Instrumentation” in computer programs refers to the tools and codes added to software programs for program analysis. This extra code is inserted before or after program compilation, to collect further information in form of metrics about the program execution. These metrics are used in analysis tools to understand system performance and behavior [17].

Sun’s initial distribution of Open MPI software came with the Sun HPC ClusterTools 7 release. This marked a transition from a commercial code base, originating from the Globalwork software of Thinking Machines Corporation to the Open MPI software with its open-source implementation. The Sun HPC Cluster-Tools package generates an executable after compiling the Open MPI source code through the compilers of Sun Studio. It also includes installation scripts for deploying these packages over a network of nodes in a multi-core system. The team of Sun HPC Cluster-Tools made substantial contributions by developing new plugins for the uDAPL Byte Transfer Layer module and the Sun Grid Engine, which served as the Solaris operating system InfiniBand solution [18]. Furthermore, the package facilitates the utilization of DTrace as a tool for MPI applications debugging and performance analysis [18].

Performance analysis tools can be divided into static and dynamic tools. The static analysis tools, such as Semantic Diff [19], Datalaude [20], and FUSE [21], are employed to improve the quality and productivity of software developers tasks. Another source code analysis tool called Static Application Security Testing (SAST) helps developers to find security flaws [22]. Static analysis tools are limited to the source code being validated, and hence, they might not be able to reproduce the complex multi-threaded system behavior [23]. To overcome this issue, the dynamic analysis approach is presented, where developers can record events while observing programs executions [23]. Dynamic analysis tools help to develop efficient parallel programs, and come in the form of profilers, debuggers, or tracers [24].

Parallel programming, using the MPI distributed memory model, is often perceived as a task that is susceptible to errors. Considerable efforts were achieved to parallelize applications and libraries through the utilization of MPI [25]. Nevertheless, in terms of software maintenance and

optimization, when porting the program to other implementations and platforms, or when deploying it onto a new hardware, developers may face further obstacles. These can include errors arising from deadlocks due to MPI communication characteristics, difficult-to-identify timing-critical bugs, and behavior that is defined by the MPI implementation or even dependent on the hardware [25]. In the realm of application instrumentation, there are mainly two approaches: external and embedded. External methods entail managing the execution substrate (program, developer, client) to segregate programs from the resource consumption perspective. While this strategy provides comprehensive measurements, it requires a cooperative execution substrate, which can sometimes deviate considerably from the standard one. On the other hand, embedded techniques depend on an internal view of the program while executing it. This method has the benefit of explaining the definite substrate. However, it causes concerns about measurement disturbance. Embedded instrumentation has the advantage of offering a representation of the actual behavior, albeit slightly modified, as seen on the execution substrate. This enables the capture of insights that might not be possible with virtualization [26].

The method of storing measurements in a trace record file, for later analysis, is known as a post-mortem method. This method has the benefit of facilitating intricate analyses, as they are distinct from instrumentation. Furthermore, it opens possibilities for post-processing and iterative or comparative examination. Typically, post-mortem instrumentation records events that are either “unreduced” or minimally reduced. This separates the process of instrumentation from analysis and offers flexibility around a central component: the trace format. This approach involves recording events from parallel programs in an external trace file for independent processing. This method facilitates intricate analyses without interfering with the execution process. It also paves the way for multiple rounds of analysis on the same data set.

There are several approaches and techniques proposed in literature to analyze MPI applications statically [27]–[29]. These methods work mainly on matching the send/receive statements, which in turns, limits the coverage. Furthermore, there is a limitation in terms of context sensitivity, this is due to the loss of communication edges during data propagation [29]. Consequently, these tools do not offer an adequate method for generating a complete communication graph.

2.3.1 Software Profiler

Profiling is a technique used to acquire program performance information to evaluate application performance and detect bottlenecks. This technique is used based on the occurrence of an event in the program and updates the statistics of the execution. After that, the gained information is stored in profile data for further analyses. Profilers provide the necessary information to understand the functions calls and call hierarchy while executing the program. They help the developers to define the fast and slow execution parts of the program, and hence, they allow developers to optimize and improve program performance. Profilers typically aggregate information over several executions of a code section or function, thus providing an average metric. For instance, profilers will often provide a profile of the functions: the average duration of a function call for each function. Because of this aggregation, profilers will not identify infrequent performance problems that do not affect the average [23]. Yet, an operation that takes more time in one node among a hundred may still be on the critical path for a group communication and cause all the other nodes to wait.

2.3.2 Software Debugger

A debugger is a dynamic analysis tool used to analyse programs during their execution to interactively examine the program behavior. It aids the developers to identify the underlying causes of program failures and flaws. As a result, debugging is difficult, and might take a longer time than that required to develop the code in the first place [30]. Debugging can be accomplished by placing breakpoints within the code. Subsequently, the developer can run the program up to that point, maintaining full control over execution and the function stack. To do this, debuggers need to be able to accurately represent a program erroneous state. They typically achieve this by utilizing the ptrace system call [31], which enables a parent process to control and monitor one of its child processes. This single system call offers sufficient functionality for constructing a fully equipped debugger, like DBX [32], IDB [33], and GDB [34]. However, debuggers typically significantly increase the program execution duration due to the added breakpoints and the execution conditions needed to stop and resume the program execution [1]. Furthermore, handling parallel payloads presents a complex task of visualizing multiple process states. This task necessitates a design that is scalable not only in terms of data collection but also in its ability to display the collected data [26].

2.3.3 Software Tracer

Tracing is a method used for recording the system events in a detailed log. For each event occurrence, a timestamp and its attributes are stored in a log, to show the execution progression of a program. Tracing can be considered as a better solution in parallel program analysis compared to profiling and debugging. It may result in less overhead, and can be utilized to both assess the application performance metrics, and investigate the details of individual inefficiencies, as it is capable of logging low-level events within the system, without loosing details through aggregation [1]. Tracing can be divided into static and dynamic tracing. Static tracing requires source code modifications and recompilation before the execution of a program, while dynamic tracing inserts trace points into the running code directly. Each trace point is associated with an event to record the behavior of the system when executing a program [23].

While tracing can provide detailed information for performance analysis, it has a few challenges in terms of data analysis, trace size, and data complexity. Since tracers can record a tremendous amount of program events, the size of the created trace file may be huge. As a result, it is preferred to visualize the trace data in the form of charts and execution diagrams, instead of detailed event lists, up to the desired understanding of the system behavior [35]. Furthermore, tracers may collect a variety of data including machine instructions, memory references, system calls, and message transfers, and hence, several methods and techniques are needed to handle this data. Finally, the data analysis requires intensive knowledge of the low-level mechanisms of systems and applications [23]. For instance, when tracing MPI applications, it is necessary to understand the underlying level of communication among MPI ranks.

2.4 Software Tracing and Instrumentation

2.4.1 Software Tracing Tools

The performance attributes of a parallel computing program may require significant changes when migrating to a new hardware, or scaling up within the same hardware [36]. As a result, it is important to find the right performance measurement tools to overcome these challenges. Score-P comprises an instrumentation framework with multiple runtime libraries and auxiliary tools. It focuses on the data collection aspect, through gathering performance data that records information from different types of parallelism, using a single tool [36]. The instrumentation feature of Score-

P allows embedding measurement probes into Fortran and C/C++ programs. When these probes are activated during measurement runs, it starts collecting data about the performance of the program. The running program must be linked with the available runtime environment, to gather performance-related data, such as hardware counters, communication metrics, times, and events. These libraries cater to different execution modes, including OpenMP or MPI parallelism, serial execution, or even hybrid combinations [37]. The prefix *scorep* command is used to compile and link the executable applications. For example, instead of using *mpicc -c foo.c*, you would use *scorep mpicc -c foo.c*. As a result, the programming paradigm is defined for MPI, and this will add the required flags for compiling and linking the application before the build command execution. The current supported instrumentation methods comprise MPI library interposition, compiler instrumentation, and easy-to-use macros for user instrumentation [37].

William et al. introduced a cyclical process for using Score-P instrumentation in his tutorial [36]. This process begins with the preparation phase, followed by measurement and analysis, and concludes with optimization. The primary objective of the first three phases is to prepare the trace files for subsequent analyses [36]. The preparation phase involves two key steps: compiling the program with the appropriate instrumentation hooks, and the performance experiment derivation. During the measurement phase, it is required to start the investigation on measurements by giving the commonly available measurement systems overview for HPC applications. These systems are then classified based on two main dimensions: data collection method (instrumentation versus sampling), and the produced output (profiles versus execution traces). In the analysis stage, the objective is to gain a comprehensive understanding of the overall trajectory of the parallel program, and to evaluate the accuracy of the measurement itself.

The Linux Trace Toolkit: next generation (LTTng) is a high performance open-source tool used for Linux kernel and user space tracing [38]. It targets the minimal instrumentation overhead through the significant efforts applied when observing the behavior of a system, where each event or call can be activated. LTTng uses a memory buffer to record the instrumented system events, with each core being assigned to a memory buffer for scalability [39]. LTTng provides a library for user-level tracing named LTTng User Space Tracing (LTTng UST). This library allows the developers to instrument user-level events.

2.4.2 Trace Analysis and Viewing Tools

A few tools found in literature can be used to analyze and visualize trace files, these tools are: Trace Compass, Vampir [41], TAU [42], Jumpshot [43], and HPCToolkit [44]. Some of these tools provide analysis and visualization only, and others provide instrumentation libraries in terms of tracing and profiling. Eclipse Trace Compass is a freely available tool that tackles issues of performance and reliability by reading and analyzing system logs and traces. The aim of this tool is to visualize traces through views, charts, metrics, and more to assist in gleaning useful insights from traces. Hence, Trace Compass offers a user-friendly and informative alternative to extensive text dumps. The Trace Compass architecture can be seen in Figure 2.3.

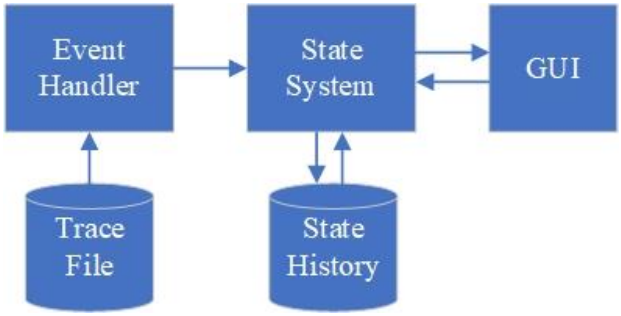


Figure 2.3 Trace Compass architecture summary

The state system, which is a key component of the ‘statesystem’ package, functions as a processor for all incoming events and has three main components: the core state system, the state provider, and the storage backend. The core state system utilizes the states to supply the requested information for viewing. It has a connection with the state history storage to fetch the requested data query. The state system acts as a receiver for event inputs from the trace, and forwards them to the state provider. After that, the state provider examines these events and implements the necessary state modifications. This way the complete state history of the traced application is modeled in Trace Compass. Finally, Trace Compass provides a view that presents the analyzed trace data for viewing. For example, the execution state (running, preempted, waiting) of each rank can be displayed as a function of time.

Another important component of Trace Compass is the analysis module, as seen in Figure 2.4. This module implements the trace data analysis, before passing the analyzed data to data providers. The data provider module passes the data to the viewer module, as part of the UI component. Typically,

an analysis module includes a single state system which houses all the data derived from the traces, along with some meta data like the analysis id. The analysis module has the capability to be queried from multiple data providers. However, it is common for one analysis module to be linked to just one data provider, to offer high degree of extensions flexibility. Finally, the data providers are connected directly to data viewers.

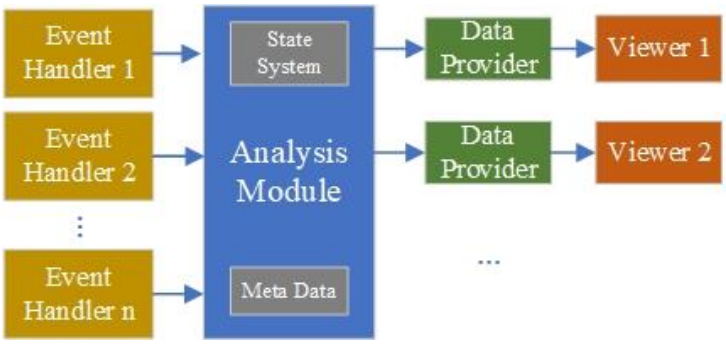


Figure 2.4 Trace Compass analysis module

The incubator project of Trace Compass includes a version tailored for developers, known as Trace Compass Incubator, where the creation and testing of new features take place. The Incubator enables developers to add new trace types by executing the relevant API functions. Once a feature has been fully developed and undergone testing, the new module may be approved for inclusion in the project. Consequently, users can utilize these new features by installing a corresponding plug-in into Trace Compass [45].

Vampir is a commercial post-mortem software trace analysis tool developed by KFA Jülich for parallel programs performance analysis. Vampir depends on Score-P profiling and instrumentation libraries. It offers several analyses and views for MPI message executions, focusing on histograms and message graphs [41]. The visualization of Vampir can be categorized into the Process State, the Statistics, and the Timeline. In the Process State, Vampir can view all the MPI processes in a box, along with the process state at a given timestamp. The Statistics view presents statistical information about the trace file in terms of event counts, distribution, and utilization. The Timeline view shows the process state over time, with its communication between other ranks or processors [46].

Tuning and Analysis Utilities (TAU) is a parallel performance toolkit used to profile, analyze, and visualize HPC applications. This tool uses an API library to trace applications, which gives the

developers the ability to configure their tracing environments, and control the overhead caused by instrumentation [41]. Furthermore, it provides portable profiling and tracing facilities along with data mining and management. TAU provides a set of visualization tools in single and aggregated forms of cores, threads, and contexts. In addition, it generates trace files of MPI applications that can be viewed in Vampir [42].

Jumpshot is another post-mortem trace analysis tool based on Java. It provides a graphical timeline that represents the trace file of an execution recorded by an instrument called MPE [43]. Furthermore, it provides a profiling library for MPI applications. This tool can be deployed as a web-browser computing environment applet. There are multiple views and graphs available for logfile data through Jumpshot, primarily the timeline series. Another view of histograms and mountain range can be generated by Jumpshot, to visualize the state duration, and the process number in each state time [47].

HPCToolkit is a set of integrated tools that can be used for performance analysis and measurements for parallel programs. The main components of HPCToolkit are: hpcviewer, hpcrun, hpcprof, and hpcstruct. The hpcviewer is the UI of HPCToolkit, that provides performance data in three views: flat, top-down, and bottom-up. The hpcrun is the measurement tool used to collect performance data and traces. The hpcprof is the profiler tool that is used to measure performance counters. Finally, hpcstruct is the software binary analysis tool that relates information between the binary executable file, and the source code file [44], [48].

2.4.3 Related Trace Formats

2.4.3.1 The Open Trace Format 2 (OTF2)

Event tracing is structured to gather and preserve highly specific data. While individual event logs, that depict single occurrences, are relatively compact, the sheer quantity of these events often leads to substantial data volumes. In general, it is preferable to have minimal impact during the process of generating traces, whether it is in terms of altering the runtime or consuming memory. This is especially crucial for performance evaluation, as it minimizes the disruption of the behavior being examined. One of the elements involved in event tracing is the format of the event trace data. This format is related to the in-memory buffer through the record stage and determines the format of the trace file when moving data to a permanent storage, when completing the trace record.

The initial trace format was the Open Trace Format (OTF), which was designed from an interoperability perspective. It was a joint effort by the Technische Universitat Dresden (Germany), the Lawrence Livermore National Lab (USA), and the University of Oregon (USA). The OTF includes a library for reading and writing traces, a documented API, and several auxiliary tools. Being the pioneer in its field, it was issued as a standalone software package deployed by multiple tools such as Vampir, TAU, and the MPI trace collection layer of the Microsoft HPC Server. After that, OTF2 was an updated version of OTF produced by Jülich Supercomputing Centre (Germany), Technische Universitat Dresden (Germany), GNS GmbH Braunschweig (Germany), Technische Universitat München (Germany), German Research School for Simulation Sciences Aachen (Germany), RWTH Aachen University (Germany), and University of Oregon (USA). This version was integrated into Score-P [49, p. 2].

The main characteristics of OTF2 are akin to other event trace formats that are record-based. It encompasses global definitions and events, and it spreads data storage over several files. These files are an anchor file, ‘n’ local definition files, a global definitions file, and ‘n’ local event files for ‘n’ threads or processes. The event records are organized in chronological order per event file. The OTF2 core is not just about the format specification, but it also includes a read/write API and a library that enables selective access. OTF2 is a standalone software package that has a vital role in the Score-P measurement runtime system, buffering event trace collection. The same binary encoding is utilized for both file representation and the in-memory buffers [37]. To view the trace file after finishing the instrumentation, the *otf2-print* command can be used to view the trace file in multiple sections. The first section starts with information about the trace anchor file, as seen in Figure 2.5.

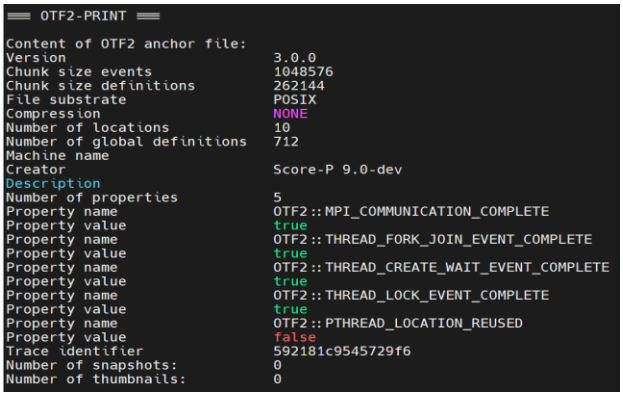


Figure 2.5 OTF2 information section overview

The next section is the global definitions, where each trace event has its own definition, ID, and attributes, as seen in Figure 2.6.

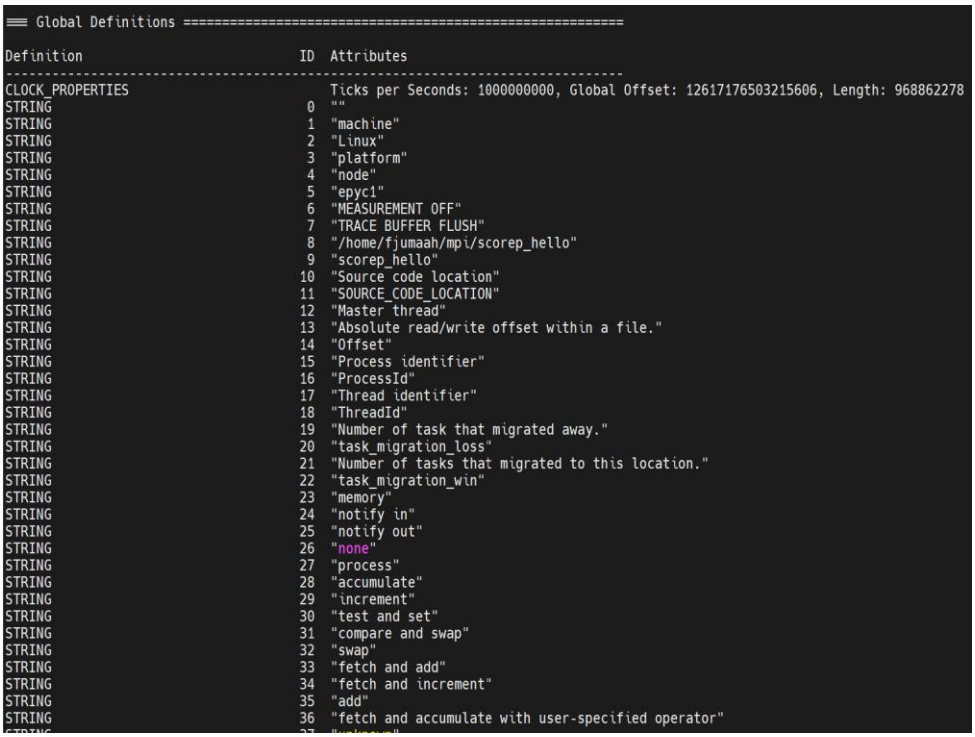
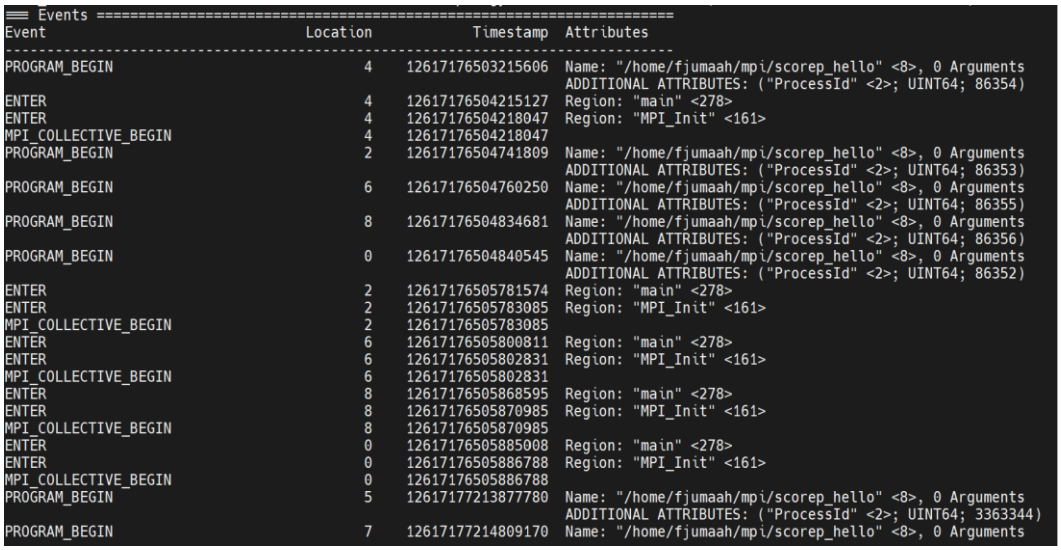


Figure 2.6 OTF2 global definitions section overview

The last section of the OTF2 file is the event records, where each event of the application has a name, location, timestamp of the event, and a few attributes. From the attributes, the reader can point to the region of this event, to gather the needed information about it. A snippet of this region can be seen in Figure 2.7.

In summary, OTF2 is a widely used trace format. It now incorporates several strategies to compress the trace data and is therefore reasonably compact. One limitation, however, is that the definition of event types is static. It is more complicated to add event types, for example to provide more advanced analysis in trace viewers.



```
=== Events ===================================================================
Event                                Location           Timestamp  Attributes
-------------------------------------------------------------------------------
PROGRAM_BEGIN                               4    12617176503215606  Name: "/home/fjumaah/mpi/scorep_hello" <8>, 0 Arguments
                                                                    ADDITIONAL ATTRIBUTES: ("ProcessId" <2>; UINT64; 86354)
ENTER                                       4    12617176504215127  Region: "main" <278>
ENTER                                       4    12617176504218047  Region: "MPI_Init" <161>
MPI_COLLECTIVE_BEGIN                        4    12617176504218047
PROGRAM_BEGIN                               2    12617176504741809  Name: "/home/fjumaah/mpi/scorep_hello" <8>, 0 Arguments
                                                                    ADDITIONAL ATTRIBUTES: ("ProcessId" <2>; UINT64; 86353)
PROGRAM_BEGIN                               6    12617176504760250  Name: "/home/fjumaah/mpi/scorep_hello" <8>, 0 Arguments
                                                                    ADDITIONAL ATTRIBUTES: ("ProcessId" <2>; UINT64; 86355)
PROGRAM_BEGIN                               8    12617176504834681  Name: "/home/fjumaah/mpi/scorep_hello" <8>, 0 Arguments
                                                                    ADDITIONAL ATTRIBUTES: ("ProcessId" <2>; UINT64; 86356)
PROGRAM_BEGIN                               0    12617176504840545  Name: "/home/fjumaah/mpi/scorep_hello" <8>, 0 Arguments
                                                                    ADDITIONAL ATTRIBUTES: ("ProcessId" <2>; UINT64; 86352)
ENTER                                       2    12617176505781574  Region: "main" <278>
ENTER                                       2    12617176505783085  Region: "MPI_Init" <161>
MPI_COLLECTIVE_BEGIN                        2    12617176505783085
ENTER                                       6    12617176505800811  Region: "main" <278>
ENTER                                       6    12617176505802831  Region: "MPI_Init" <161>
MPI_COLLECTIVE_BEGIN                        6    12617176505802831
ENTER                                       8    12617176505868595  Region: "main" <278>
ENTER                                       8    12617176505870985  Region: "MPI_Init" <161>
MPI_COLLECTIVE_BEGIN                        8    12617176505870985
ENTER                                       0    12617176505885008  Region: "main" <278>
ENTER                                       0    12617176505886788  Region: "MPI_Init" <161>
MPI_COLLECTIVE_BEGIN                        0    12617176505886788
PROGRAM_BEGIN                               5    12617177213877780  Name: "/home/fjumaah/mpi/scorep_hello" <8>, 0 Arguments
                                                                    ADDITIONAL ATTRIBUTES: ("ProcessId" <2>; UINT64; 3363344)
PROGRAM_BEGIN                               7    12617177214809170  Name: "/home/fjumaah/mpi/scorep_hello" <8>, 0 Arguments
```

Figure 2.7 OTF2 regions section overview

2.4.3.2 Common Trace Format (CTF)

CTF is a format utilized for trace storage. It is self-explanatory, binary, and designed for ease of writing. CTF is a binary trace format that is engineered to be extremely quick to write, while still maintaining a high degree of flexibility. It enables traces to be naturally produced by any application or system written in C/C++ [50]. The CTF file has one metadata stream that describes the data in JSON format. Furthermore, CTF has one or more data streams which contain trace events from a tracer tool. Through Trace Compass, both files are needed, the metadata file to know how to read the events, while the data streams are used to read the events. Trace Compass has a CTF parser tool to read the trace files. The metadata is composed in a subset of the C language, known as Trace Stream Description Language (TSDL) [51]. To interpret it, it is first depacketized, then the raw text is parsed using an ANother Tool for Language Recognition (ANTLR) grammar. This parsing process occurs in two stages. Initially, a lexer segregates the metadata text into tokens. These tokens are then pattern-matched using the parser to construct an Abstract Syntax Tree (AST). This AST is navigated using “IOStructGen.java” to populate streams and traces in the parent trace object.

The OTF2 traces can be converted into the CTF format, readable by Trace Compass, using a converter tool developed by the Dorsal Lab at Polytechnique Montreal [52]. This project facilitates the conversion of OTF2 traces into CTF traces utilizing the OTF2 C API and BareCTF library. To

optimize the conversion speed, multiple threads are employed. Each thread is assigned several locations (i.e. trace streams from different nodes), and local event readers are used for concurrent reading of the OTF2 traces. For each OTF2 location, a corresponding CTF stream is generated. Additionally, a CTF stream that contains global definitions is also created. The converter is still under development and does not support the conversion of all types of events in an OTF2 trace, it focuses mainly on the MPI events.

2.5 Critical Path Analysis

To analyze large traces of multi-threaded applications, one can use the critical path analysis, which examines a specific task and its dependencies across multiple threads. The critical path algorithm identifies the threads that determine the total execution time. Debugging performance could be challenging, because of the concealed nature of processing, and the discrepancies between different runtime environments. By integrating the critical path knowledge along with the traditional tracing tools, the root causes of issues in multi-threaded applications can be identified. Developers of multi-threaded programs must be familiar with the program components that can be optimized. Therefore, critical path analysis provides a holistic view of the parallel computation performance and the interplay between various threads, as opposed to metrics that simply aggregate values for individual threads. Trace Compass offers the functionality of critical path computation analysis to identify the expended time during application execution, across different processes [53]. The available critical path analysis is available for Linux operating system traces. However, this analysis does not support the semantics of MPI applications with their sophisticated group communications.

There are a limited number of critical path algorithms and methods available in the literature, as summarized in Table 2.1. Gilardeau et al. [54] devised an algorithm that effectively isolates the execution segments contributing to task latency. Using certain Linux kernel events (such as sched_wakeup), this algorithm can identify the threads associated with a specific task. However, the tool falls short in linking this information with user-space functions, which restricts its applicability for analyzing application code [23]. Hollingsworth et al. introduced an approach for calculating the critical path of activity graphs in real-time. This method eliminates the need to construct the entire graph and store events. Instead, it involves appending the current critical path value into a transferred message. Once this message is received, the value is replicated into a local

variable. After that, the processed value is incorporated, then returned as a reply. Once the message is received, a comparison between the local variable value and the received one is achieved, and the greater value is preserved [55]. Miller et al. introduced an algorithm to determine the critical path value in distributed applications that rely on message passing. As a cyclic directed graph, the Program Activity Graph (PAG) is characterized with nodes symbolizing the events marking the begin and the end of the activity. The critical path is defined as the longest path for which the weights of all the segments are aggregated. The research contrasts a compacted algorithm, along with a distributed version that can be used in computing the critical path offline. The primary objective of the study is to use a parallel algorithm for expediting the offline calculation of the critical path [56]. Ali et al. has proposed a new method to apply the critical path analysis to intricate systems that consist of multiple interconnected state machines. This approach eliminates the need for time-consuming initial modeling, through the utilization of control-flow tracing, to uncover hidden software state machines automatically. Additionally, the method uses iterative refinement to integrate necessary manual annotations with minimal effort [57].

Table 2.1 Critical path computation methods summary

Author	**Proposal**	**Methodology**	**Discussions**
Gilardeau et al. [54]	Algorithm	An algorithm that isolates the execution segments contributing to task latency	This algorithm can identify the threads associated with a specific task
Hollingsworth et al. [55]	Algorithm	Calculating the critical path of activity graphs in real-time	This method eliminates the need to construct the entire execution graph. However, it adds extra overhead from storage perspective for the critical path value local variables
Miller et al. [56]	Algorithm	As a cyclic directed graph, PAG is characterized with nodes symbolizing the events marking the begin and the end of the activity	This algorithm determines the critical path value in distributed applications that rely on passing messages
Ali et al. [57]	Method	The application of critical path analysis to intricate systems that consist of multiple interconnected state machines	This approach eliminates the need for time-consuming initial modeling through the utilization of control-flow tracing to uncover hidden software state machines

2.5.1 MPI Critical Path Related Work

Several studies have been investigated in the literature to understand the tools and frameworks proposed to measure the critical path of an MPI application, as seen in Table 2.2. Schulz et al. in his research presented a method for calculating the critical path of a simple MPI application with send/receive activities [5]. The proposed approach starts with collecting the trace files, while executing the MPI application, creating subgraphs with critical path information. After that, he used post-mortem analysis to merge the subgraphs into the global execution graph, to extract the critical path. Hilbrich et al. proposed a method to detect MPI deadlocks through the utilization of MPI execution graphs [58]. The proposed method involves a general deadlock model for MPI applications, which visualises these deadlocks. Bak et al. presented the MPI task graph scheduling algorithm, which provides an efficient synchronization and communication between MPI tasks [59]. This approach helped in reducing the critical path of MPI applications for a mixed sequence of computations and communications.

A new metric was proposed by Denis et al. to measure the MPI overlap between communications and computations, along with a benchmark suite to evaluate MPI overlapping strategies [60]. Chen et al. presented a new method for detecting the MPI critical path candidates, using a potential critical path pool to capture the critical activities of MPI applications. Furthermore, he proposed a scalable performance modeling framework to record event traces, execution graphs, and critical path candidates [61]. Friese et al. suggested another method to create MPI models based on hierarchical critical path analysis. This approach captures the sequence of dependent operations while disregarding any overlap [62]. This method begins by constructing a histogram of critical tasks, then parameterizes the instance counts and arguments using a scaling function. The algorithm employed to identify the critical path is known as “the longest path method”.

Hendriks et al. introduced a graph-based representation of execution traces that takes into account the temporal order of tasks, critical path analysis, and trace distance computation [63]. Dietrich et al. introduced an enhancement to the MPI performance measurement tool “Score-P” that allows it to monitor dependencies between the regions of MPI on the target device and host [64]. Furthermore, he defined the required rules to detect these dependencies and implement them in the CASITA analysis tool for critical path analysis enablement.

Denys et al. proposed a distributed critical path analysis algorithm that relies on the pre-computation of the execution graph from multiple trace files, without the need to aggregate all traces in one node [65]. Moraru et al. proposed a new approach, based on hash-table algorithm that enhances the critical path time of an MPI application through the message-matching process [66]. This approach was developed mainly for point-to-point MPI communications.

Table 2.2 MPI critical path analysis literature summary

Author	**Proposal**	**Methodology**	**Discussions**
Schulz et al. [5]	Method	An approach to calculate the MPI critical path with mpi_send and mpi_recv	This method cannot be used for collective communications
Hilbrich et al. [58]	Model	Deadlocks detection through the utilization of MPI execution graph	This model was derived mainly for deadlocks detection
Bak et al. [59]	Algorithm	An algorithm derived for MPI task graph scheduling for efficient synchronization and communication between MPI tasks	Useful for communication synchronization, and hence, improving the critical path reduction
Denis et al. [60]	Metric	The proposed metric to qualify the MPI communication overlapping strategies	This work can be used to reduce the critical path of an MPI
Chen et al. [61]	Framework	A new method to detect MPI critical path candidates from potential paths. It provides a pool of potential critical paths	Further work might be needed for pool search and select algorithms to enhance the framework performance
Friese et al. [62]	Model	A new model to calculate MPI critical path based on hierarchical critical path analysis	This model captures the sequence of dependent operations while disregarding any overlap
Hendriks et al. [63]	Model	A graph-based representation of execution traces in terms of temporal order of tasks, critical path analysis, and trace distance computation	This model focuses on the trace distance computation. It provides a promising motivation towards deriving MPI execution graph algorithms
Dietrich et al. [64]	Model	An enhancement to the MPI performance measurement tool “Score-P”	This model allows Score-P to monitor dependencies between the regions of MPI of the target device and host
Denys et al. [65]	Algorithm	A distributed critical path algorithm depends on the pre-computation of the execution graph from multiple trace files	This algorithm provides a solution to calculate the MPI critical path from multiple trace files without the need of aggregation
Moraru et al. [66]	Method	A new method derived from hash-table algorithm	This method was developed for point-to-point MPI communications enhancement

2.6 Summary

This chapter presented an overview of the MPI communication topologies and related tools for analyzing and visualizing MPI applications. An overview of the performance analysis tools was presented, to show the state of the art of the current developments that were achieved to analyze MPI applications. However, most of the related work focused on the standalone development to compute the MPI critical path. This is not enough to give the developer detailed information about the program behavior. As a result, the critical path computation of MPI must be provided with sufficient graphs to visualize the events activities during the execution. This motivated us towards developing a critical path framework that is integrated with a trace analysis and visualization tool, to provide detailed analysis up to the operating system kernel level. Furthermore, the related work focused mainly on the MPI point-to-point communication. This is not sufficient, since the MPI collective communications are important and used in complex HPC applications. The proposed MPI execution graph algorithm considers this communication topology, along with the point-to-point one, for critical path analysis.

CHAPTER 3 RESEARCH METHODOLOGY

3.1 Introduction

The main objective of the critical path analysis is to examine and optimize the execution of software applications. This is accomplished by tracking the events that occur during the application operation and recording them in a trace file. Subsequently, a tool analyzes the trace file to conduct the analysis.

This chapter outlines the proposed methodology for creating the critical path analysis framework and establishing the data collection process. Additionally, it provides details on the experimental setup, test prerequisites, and testing scenarios.

3.2 Research Methodology Phases

The proposed methodology of this research has three phases, as seen in Figure 3.1. Phase I involves defining the problem statement, identifying the research objectives, and investigating the state-of-the-art literature. In Phase II, the proposed framework modules are developed. The execution graph algorithm is developed and implemented. Furthermore, the generic Linux operating system critical path algorithm in Trace Compass was extended to analyze MPI events. Phase III involves collecting trace files and analyzing them. Furthermore, the proposed algorithm is validated through several case studies and tests.

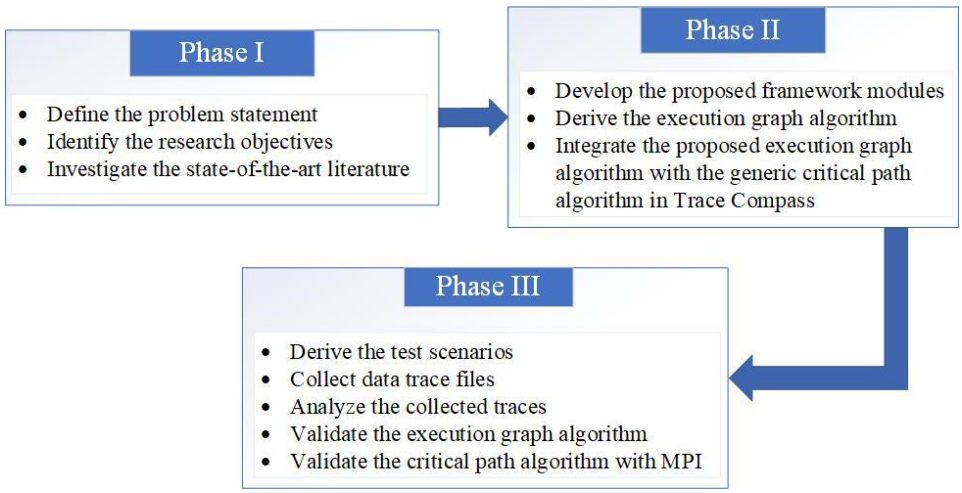


Figure 3.1 Proposed research methodology

3.3 Proposed Framework

The proposed framework of this study was derived based on three main modules: the MPI execution graph algorithm, the critical path algorithm extension, and the experimental setup of the development environment, the proposed algorithm validation, and the data collection.

3.3.1 MPI Execution Graph Algorithm

To calculate the critical path for an MPI application, it is necessary to generate the execution graph. The proposed MPI execution graph algorithm can be seen in Figure 3.2. The development of this algorithm was motivated by the Linux kernel-level operating system execution graph algorithm in Trace Compass. The flow starts with the Trace Compass parser; this is the initial stage where the system parses the trace data where the trace file is the input.

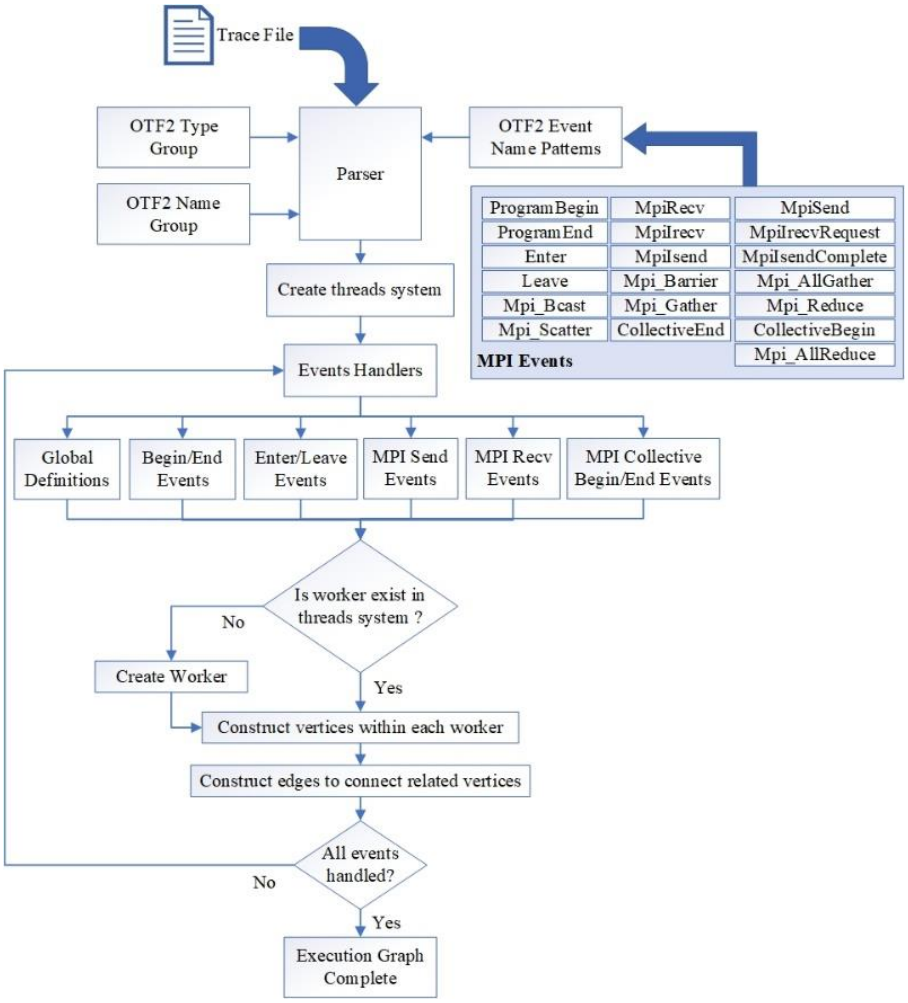


Figure 3.2 The proposed MPI execution graph

OTF2 Group and OTF2 Events are components of the OTF2 format, along with the OTF2 event names patterns. The events are Program Begin, MPI_Send, MPI_Recv, MPI_Gather, MPI Collective, and Collective Begin; these represent various stages in an MPI program. MPI_Send and MPI_Recv are used for message send and receive among processes. MPI_Gather collects data from all processes and combines it into one buffer. MPI Collective represents collective communication operations. Next, we look at the worker creation. The worker is a data structure in Trace Compass that is used to track the events occurring in every MPI rank. For instance, if the traced application was executed on an 8 ranks system, we will create 8 workers in the threads system of Trace Compass; this involves creating worker processes, assigning tasks to each worker (represented as vertices), tracing the execution graph, and checking if all events have been processed. The proposed MPI execution graph algorithm can be seen in Figure 3.3.

The proposed algorithm has a main procedure for checking all the incoming events in the trace file. Based on the patterns provided to the parser, the incoming event is analyzed. The global definition region in the OTF2 format has information about the system, which includes events names, regions, locations, ranks, communicators, and groups. This information is needed in other analysis; hence, it is required to collect it before starting the MPI events analysis. After that, the MPI event Program Begin and Program End analysis starts; these are the trace events that indicate the beginning and end of MPI execution. These events are represented by a horizontal vertex with the RUNNING status. Next, MPI_Send, and MPI_Isend events are analyzed. It is important to save these events in a list, so that it is easy to link the receive events with its sender. These events are represented by a horizontal vertex with a RUNNING edge. After that, the MPI_Recv and MPI_Irecv events are analyzed. There are two main functions to consider, first, the horizontal link where we connect these events with the previous vertex via a horizontal edge, RUNNING, second, the vertical link, where we connect the receive vertex with its send vertex via a vertical edge, NETWORK.

```
Input: trace file T
Output: MPI Execution Graph G
1   for all event e ∈ T do                    ► Main procedure
2     eventName ← e.getName
3     eventType ← matcher.group
4     graph G
5     switch (eventType)                      ► Global definitions
6       case (GlobalDef)
7         switch (eventName)
8           case (otf2_string)
9             Info← e.getCont(stringRef, stringValue)
10            break
11          case (otf2_region)
12            Info← e.getCont(RegionRef, stringValue)
13            break
14          case (otf2_location)
15            locationGroupId ← e.getCont(locGrp)
16            break
17          case (otf2_comm)
18            CommGroup ← (e.getCont(commRef, grRef))
19            break
20          case (ProgramBeginEnd)            ► MPI events
21            getInfo(e)
22            createWorker(hostThread,srcRank, ts)
23            Link_Horizontal (mpiWorker, RUNNING, ts)
24            break
25          case (EnterLeave)
26            getInfo( e)
27            resolveEnterLeave(e, otf2MpiEvent, ts)
28            break
29      break
30      case (mpiSend | mpiIsend)
31        getInfo( e)
32        createWorker(hostThread,srcRank, ts)
33        Link_Horizontal (mpiWorker, RUNNING, ts)
34        sendEvents ← (mpiWorker, srcRank, ts)
35        break
36      case (mpiRecv | mpiIrecv)
37        getInfo( e)
38        createWorker(hostThread,srcRank, ts)
39        Link_Horizontal(mpiWorker, RUNNING, ts)
40        findSrcEvent(sendEvents, mpiWorker, ts)
41        Link_Vertical(sendEvent, e, NETWORK)
42        break
43      case (MpiCollectiveBegin)
44        getInfo( e)
45        createWorker(hostThread,srcRank, ts)
46        Link_Horizontal(mpiWorker, RUNNING, ts)
47        break
48      case (MpiCollectiveEnd)
49        getInfo( e)
50        createWorker(hostThread,srcRank, ts)
51        Link_Horizontal(mpiWorker, RUNNING, ts)
52        CollEndVert ← (mpiWorker, ts)
53        opCode ← content.COLLECTIVE_OPERATION
54          switch (opCode)
55            case (BCAST | SCATTER)
56              mpiRtoA( e )
57              break
58            case (GATHER | REDUCE)
59              mpiAtoR( e )
60              break
61        break
62  end for
```

```
1   function mpiAtoR (e, otf2MpiEvent)      ► Utilities
2     getInfo( e)
3     root ← content.getFieldValue(OTF2_ROOT)
4     comm ← content.getFieldValue(OTF2_COMM)
5     srcLocID ← getLocation (root, comm)
6     for op ∈ RtoAQueue) do
7         if (op.isaOp(opCode, srcLocID, comm))
8             aOp = op;
9     end for
10    srcEvent = null
11    if (assocOp == null)
12      srcEvent = destEvent
13      members = getMembCommRef(comm)
14      pendingLocations = new (members)
15      aOp= new RtoA(opCode, commu, srcLocID, srcEvent)
16    createWorker(hostThread,srcRank, ts)
17    Link_Horizontal(mpiWorker, RUNNING, ts)
18    Link_Vertical(sendEvent, e, NETWORK)
19  end function
20  function RtoA (e, otf2)
21    getInfo( e)
22    root ← content.getFieldValue(OTF2_ROOT)
23    comm ← content.getFieldValue(OTF2_COMM)
24    destLoID ← getLocation (root, comm)
25    for op ∈ AtoRQueue) do
26        if (op.isassocOp(opCode, destLoID, comm))
27            assocOp = op;
28    end for
29    srcEvent = null
30    if (assocOp == null)
31      srcEvent = destEvent
32      members = getMembCommRef(comm)
33      pendingLocations = new (members)
34      assocOp=new AtoR(opCode, comm, destLID, destEvent)
35    createWorker(hostThread,destRank, ts)
36    Link_Horizontal(mpiWorker, RUNNING, ts)
37    Link_Vertical(sendEvent, e, NETWORK)
38  end function
39  function getInfo( e)
40      content ← e.getCont
41      srcRank ← e.getFieldValue(locGrpId)
42      hostThread ← new HostThread(e.getHostId, srcRank)
43      ts ← e.getTimestamp
44      return event information (content, rank, hostThread, ts)
45  end function
46  function createWorker(hostThread,srcRank, ts)
47    system ← getSystem
48    mpiWorker ← system.findWorker(host, srcRank)
49    if (mpiWorker == null)
50      name ← e.getName
51      mpiWorker ← new MpiWorker(src, name, now)
52      system.addWorker(mpiWorker, srcRank);
53    return mpiWorker
54  end function
55  function Link_Horizontal(mpiWorker, MPI_STATUS, ts)
56    find graph G
57    vertex ← (mpiWorker, ts)
58    G ← vertex(MPI_STATUS)
59  end function
60  function resolveEnterLeave(e, otf2MpiEvent, ts)
61    createWorker(hostThread,srcRank, ts)
62    Link_Horizontal(mpiWorker, RUNNING, ts)
63  end function
64  function findSrcEvent(sendEvents, mpiWorker, ts)
65      sendEvent ← find source event sendEvents(mpiWorker, ts)
66      return sendEvent
67  end function
68  function Link_Vertical(mpiWorker, MPI_STATUS, ts)
69    find graph G
70    edge ← (mpiWorker, ts)
71    G ← edge(MPI_STATUS)
72  end function
```

Figure 3.3 MPI execution graph implementation

3.3.2 MPI Critical Path Algorithm

The generic critical path algorithm used in this research was developed by [54], as seen in Figure 3.4. The author defines the active path of execution as “the execution path where all blocking edges are replaced by their corresponding subtask”.

```
Input: execution graph G, task T, v_s, v_e
Output: path P
 1: v ← v_s
 2: while v is not v_e do                          ▷ Forward iteration
 3:    E ← v.right
 4:    append PROCESS(E) to P
 5:    v ← E.to
 6: end while
 7: function PROCESS (edge)
 8:    if edge.label is blocking) and
 9:          edge.to has incoming vertical edge then
10:       return RESOLVE(edge)
11:    else
12:       return E
13:    end if
14: end function
15: function RESOLVE (edge)
16:    TMP ← ∅
17:    v ← edge.from                                ▷ Follow incoming
18:    while v.ts > edge.from.ts do                 ▷ Backward iteration
19:       E ← v.left
20:       if v.down.label is network then
21:          prepend RESOLVE(E) to TMP
22:       else
23:          prepend PROCESS(E) to TMP
24:       end if
25:       v ← E.from
26:    end while
27:    return TMP
28: end function
```

Figure 3.4 Generic active path computation algorithm [54]

As seen in Figure 3.4, the main task states progress in order, with each traversed edge being incorporated into the active path. Upon encountering a blocked state, the trace of the incoming wake-up edge is initiated, leading to the commencement of the backward iteration. As we move backwards, the traversed edges are added to a local path at the beginning. If a packet is received, its source is traced backwards. A recursive repetition is needed when facing a blocking edge during a backward iteration. Upon reaching the beginning of the blocking interval, the backward iteration stops. After that, the forward iteration resumes, after accumulating the added path into the active path.

In this study, further contexts are required to enable the MPI critical path analysis through this algorithm. We consider the generic block state as a network state, where the MPI sender ranks sends the message, while the MPI receiver ranks is waiting for it. The running state of the algorithm is kept for normal MPI execution, which includes all MPI calls, except the receiving calls. The unknown state was ignored since there is no unknown state within MPI execution. The critical path algorithm, after extending it for MPI applications, can be seen in Figure 3.5.

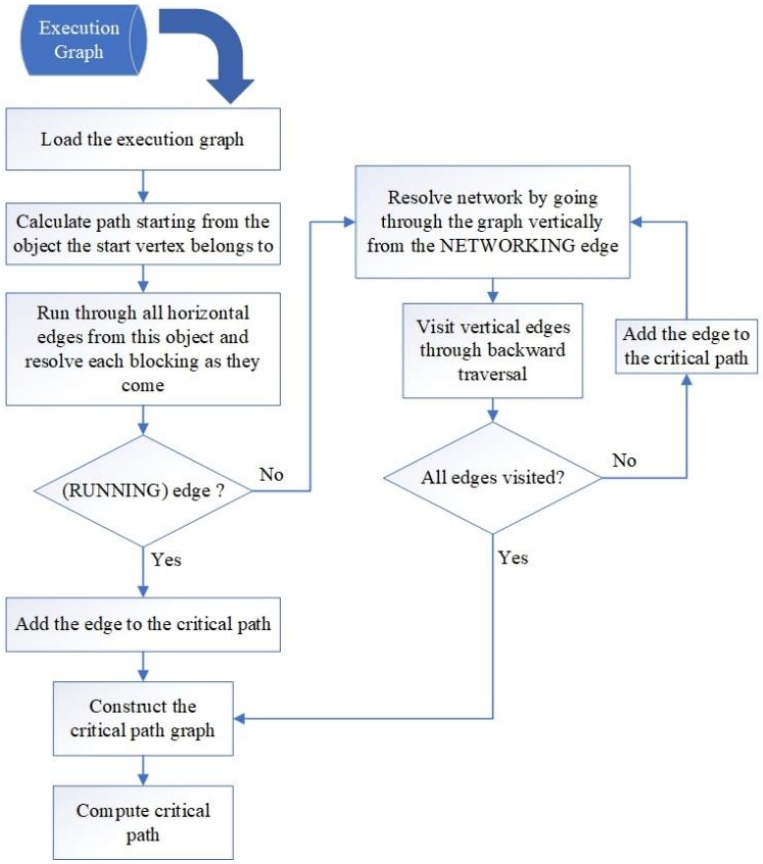


Figure 3.5 MPI critical path algorithm

3.3.3 Environment Setup

The required setup for this study can be seen in Figure 3.6. The Slurm work environment was set to have two nodes, with a variable number of cores to represent the number of ranks needed for MPI applications (from 2 ranks, up to 50). The command used for this operation can be seen below:

```
salloc --time=1:0:0 --nodelist=epyc1,epyc2 --cpus-per-task n
```

where: --time: the required time to run the experiments. --nodelist: the names of the assigned nodes. --cpus-per-task: the number of the processor cores (ranks)

After setting the Slurm environment, it is required to load the needed libraries for MPI and Score-P executions, i.e. mpich, scorep, cubelib, and cubew.

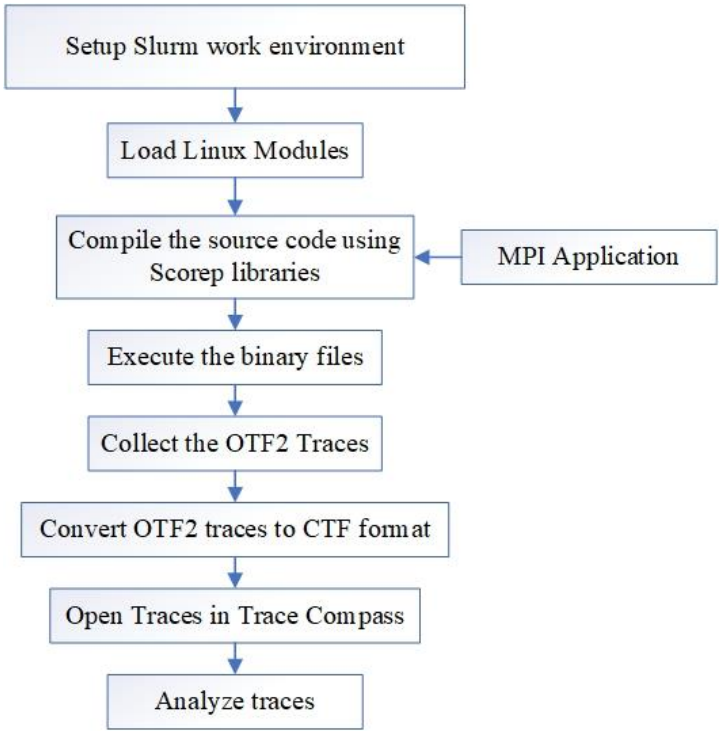


Figure 3.6 Environment setup for experiments

Next, the MPI applications are compiled using Scorep libraries, to add the required instrumentation for tracing the execution of the application. Once the compilation is completed, we can run the binary executable files. After the execution end, the OTF2 trace files are ready for analysis. To use the trace files in Trace Compass, it is needed to convert them into the CTF format, using the OTF2-to-CTF converter tool from the DORSAL laboratory, which can be found online [52]. Finally, the converted trace files can be visualized and analyzed in Trace Compass.

3.4 MPI Applications

In this study, we used several MPI application examples published by John Burkardt from South Carolina university, Department of math science [67]. These examples are published and distributed under the GNU LGPL license. These examples are:

1- **MPI Hello World** is a program written in the C language that displays the message “Hello, World!”. It uses MPI, a standardized and portable system for message-passing that is designed to work efficiently across various parallel computing architectures. This application showcases the send/receive capabilities of MPI.

2- **MPI Ring** is a program developed in the C programming language that calculates the estimated duration required to transmit a vector containing N values of double precision through each process in a circular configuration.
3- **MPI Monte Carlo** is a C application used to compute the value of Pi using the Monte Carlo method.
4- **MPI Random** is a program implemented in the C language that illustrates a method to generate an identical sequence of random numbers for both sequential and parallel execution under MPI.
5- **MPI Communicator** is a program written in the C language that establishes new communicators that involve a subset of the initial group of MPI processes present in the default communicator, MPI_COMM_WORLD.
6- **Wave** is a program that can be used to solve 1D wave equations using MPI parallel execution. This program was tested on 10 ranks (Wave10), and 50 ranks (Wave50).

3.5 Summary

This chapter presented the proposed framework to compute the MPI critical path analysis. The framework was developed in three stages: the development of the MPI execution graph, the integration with Trace Compass, and the extension of the available kernel-level critical path algorithm. Furthermore, the environment setup used to initiate the test experiments, along with the MPI test programs, were presented. In the following chapter, we present the article “Multi-level Critical Path Analysis Framework for MPI Applications“. Through this article, the validation results of the proposed framework are illustrated with further discussions.

CHAPTER 4 ARTICLE 1: MULTI-LEVEL CRITICAL PATH ANALYSIS FRAMEWORK FOR MPI APPLICATIONS

Authors: Fawaz Jumaah, Arnaud Fiorini, Michel Dagenais

Department of Computer and Software Engineering, Ecole Polytechnique de Montreal, Montreal, H3T 1J4, Canada

Submitted to: Concurrency and Computation: Practice and Experience, November 24, 2023

4.1 Abstract

With the rapid growth of multi-node multi-core systems in modern supercomputers, it has become essential to establish scalable applications with the required libraries and tools. As the applications tend toward more complexity, this makes the performance of parallel programs to be difficult to optimize and debug. Furthermore, the efficiency of large-scale systems is constrained by the intercommunication between nodes. To understand the root cause of these delays, tracing analysis is needed. This study proposes a framework for critical path computations of Message Passing Interface (MPI) communications. It is based on the Trace Compass trace analysis and viewing tool. A new algorithm was developed and implemented to extract the MPI execution graph from a trace, and integrate it within the Trace Compass incubator developer environment. The proposed new algorithm extends the generic kernel-level critical path algorithm used in Trace Compass, to read the generated MPI execution graph intervals and compute the critical path. With the critical path view, any imbalance or bottleneck in an MPI application immediately becomes obvious, speeding up the debugging of performance problems.

Keywords: Execution Graph, MPI, Critical Path, ScoreP, Trace Compass, OTF2, Tracing

4.2 Introduction

In recent times, the scale of supercomputers and data centers for Cloud computing has enjoyed continuous growth. As a result, High-Performance Computing (HPC) libraries and tools are used in an increasing number of applications, from simulation to machine learning. These applications aim to tackle large-scale problems, necessitating the use of high-performance systems [68], [69]. This encouraged the developers of these applications to adapt their programming in order to exploit the available multi-node multi-core heterogeneous systems. To optimize performance of such applications, it is essential to determine and study the crucial segments of the application code. In particular, it is necessary to ensure efficient communications among these multiple nodes. Furthermore, it is challenging to understand and mitigate the performance issues because of the number of nodes involved, and hence, the debugging of such applications becomes challenging [65], [70].

When using muti-node platforms, MPI is deployed for communications between nodes due to its proven scalability [69]. MPI is a programming model for message communications in parallel computers, which allows the developers to share information between different cores and nodes in large-scale distributed systems. Parallel programming, using the distributed memory model of MPI, is often perceived as a task that is susceptible to errors [71]. Considerable efforts were made to parallelize applications and libraries through the use of MPI. Nevertheless, when porting the program to other platforms, developers may face further obstacles in terms of software maintenance and optimization. These can include errors arising from deadlocks due to MPI communication characteristics, difficult-to-identify timing-critical bugs, and behavior that is defined by the MPI implementation [25].

The analysis tools developed for the purpose of optimizing application performance, and detecting bottlenecks, can be categorized into profiling and tracing. Profiling tools provide an abstracted, aggregated, performance fingerprint of an application. These tools involve information about the execution time of the functions of an application. On the other hand, tracing tools provide recorded event activities with timestamps, allowing software engineers to investigate the application performance in more details [23]. These events are visualized in the form of charts, flame graphs, execution graphs, and tables. Tracing is a method that involves the use of a tracing tool to capture low-level events within a system, when executing a program. During the events recording, the

software developer can activate specific instrumentation points in the application source code, and store the recorded events in a trace file. After that, the developer can use trace viewers and analyzers to investigate the source of performance issues, during the application execution [17], [23], [72]. To analyze large traces of multi-node multi-threaded applications, one can use the critical path analysis, which examines a specific task and its dependencies across multiple threads and processes [69]. The critical path is an essential application characteristic, which refers to the longest sequence of operations in a system that can be executed without any interruptions or waiting time [5]. In simpler terms, the critical path signifies the time required to finish a parallel application task, replacing the waiting states of the target thread by the activity from the other processes on which we are waiting. By identifying the critical path of parallel applications, applications developers can determine the possible performance optimization in their applications [69]. Furthermore, it provides more details on the application behavior and its communications, with respect to scaling and performance changes, which will help in identifying the bottlenecks of the application performance. Since the available number of processor cores might reach in the hundreds or the thousands, it is clearly impossible to manually analyze the critical path. As a result, it is required to have tracing analysis tools to analyze the MPI critical paths and visualize the results.

In MPI applications, which involve multiple nodes and processor cores, delays in peer-to-peer, root-to-all, and all-to-root communications have a significant impact on the critical path [2]. The critical path computation takes place everywhere a process waits for unblocking events from other processes or resources. This happens in simple programs when processes wait for data from other processes, or wait for the termination of other programs. Therefore, the efficiency of large-scale systems is often constrained by the intercommunication between nodes and processes [3]. To understand the root cause of these delays, tracing analysis is needed. As a result, it is important to have MPI performance analysis instruments, helping the HPC application developers to improve their code [4].

The purpose of this research is to develop a new framework for the critical path computation of MPI applications, through the development of a new MPI execution graph extraction algorithm. Then, an extension to the available generic kernel-level critical path algorithm of Trace Compass was derived to compute the critical path of MPI programs. This framework uses Trace Compass

libraries to read the MPI trace file in CTF format, then extract the required information and visualize the critical path via the Trace Compass User Interface (UI) modules.

4.3 Literature Review

This literature survey focuses on MPI communication approaches, the critical path analysis, and the MPI execution graph computation. Then, the proposed new framework aims to cover the gaps found in existing work from several perspectives. First, the data collection procedure was not adequately detailed in several previous studies. This study exploits Score-P to collect trace data from multiple MPI applications. After that, the trace files are converted from the OTF2 to the CTF format, so that it can be imported in Trace Compass for trace analysis. Secondly, the proposed framework computes the critical path of MPI applications from the point-to-point, and collective communications. Finally, the proposed framework relies on a generic critical path algorithm for kernel-level tracing. This provides the framework more scalability towards future developments and enhancements to trace kernel-level MPI events.

4.3.1 MPI Communication Types

MPI supports two types of communications, namely: point-to-point (or two-sided) communications, and collective (or group) communications. Two concepts were established in MPI to handle these communications: groups and communicators. These concepts assist the developers to identify the context and the scope of communications. A group consists of a sequence of processes or threads, each linked with a unique processor node known as rank, ranging from 1 to N. When initializing an MPI program, the system assigns these unique rank numbers to each processor node. The communicator is a set of nodes that may participate in each communication. MPI allows the developers to talk to each rank, within the same communicator, in terms of sending or receiving data.

4.3.2 Point-to-Point Communications

The point-to-point communication is a fundamental method of interaction, where a message is transmitted from a sender to a recipient, with the order of messages being preserved. The sender uses a buffer data structure to hold the message, with an envelope containing information utilized by the receiver side. This information helps the receiver node to select the required message and store it in a buffer. An example of point-to-point communication can be seen in Figure 2.1. In this

case, it is the responsibility of the MPI implementation to determine how the data is handled. Usually, a dedicated buffer area within the system is set in the destination process, to store data that is in the process of being transferred.

4.3.3 Collective Communications

The collective communication is an MPI communication type where the communication messages can be exchanged among a communication group of ranks, instead of only two. This allows the processes to communicate in root-to-all, all-to-root, and all-to-all topologies. The popular available MPI calls for collective communication are barrier, allgather, broadcast, and allreduce. Collective operations are frequently used by MPI application developers. This may lead to a significant impact on the application performance [2]. An example of collective communication can be seen in Figure 2.2. One of the standard collective communication methods is called Broadcast. In this method, a single root process disseminates identical data to all processes within the same communicator. This is commonly seen when configuration parameters need to be distributed to all system processes, or when user input data must be transmitted to a parallel program, while the MPI Scatter method distributes different parts of an array to various processes. MPI Gather operates differently, by collecting data from multiple processes and consolidating it into one. This technique is particularly useful in applications that require parallel searching and sorting. In the Allgather method, all the data elements are gathered to all the available processes. In other words, Allgather is similar to MPI gather followed by MPI broadcast.

4.4 Critical Path Analysis

The critical path highlights the calls and events from the complete execution graph of a program. Then, it generates a smaller execution graph visualized as a time graph to present the system events that have an impact on the program execution duration. Events are presented as state changes blocks, and the communications among processes or threads are presented as arrows. As a result, the developer can understand the behavior of the program execution, and study the source of performance issues and bottlenecks [65].

The critical path analysis can be used to analyze large traces of multi-threaded applications. It examines a specific task and its dependencies across multiple threads. The critical path algorithm identifies the threads that determine the total execution time [73]. Debugging performance could

be challenging, because of the concealed nature of processing and the discrepancies between different runtime environments [74]. By integrating the critical path knowledge along with the traditional tracing tools, performance bottlenecks in multi-threaded applications can be identified [69]. Developers of multi-threaded programs must be familiar with the program components that can be optimized. Therefore, the critical path analysis provides a holistic view of the parallel computation performance, and the interplay between various threads, as opposed to metrics that simply aggregate values for individual threads [75]. Trace Compass offers the functionality of critical path computation, to identify the expended time during the application execution across different processes [53]. The available critical path analysis is used for Linux operating system events and system calls. However, this analysis does not support MPI communication events.

There is a limited number of critical path algorithms and methods available in the literature. Gilardeau [54] devised an algorithm that effectively isolates the execution segments contributing to task latency. Utilizing certain Linux kernel events (such as sched_wakeup), this algorithm can identify the threads associated with a specific task. However, the tool falls short in linking this information with user space functions, which restricts its applicability for analyzing application code [23]. Hollingsworth introduced an approach for calculating the critical path of activity graphs in real-time. This method eliminates the need to construct the entire graph and store events. Instead, it involves appending the current critical path value into a transferred message. Once this message is received, the value is replicated into a local variable. After that, the processed value is incorporated, then returned as a reply. Once the message is received, a comparison between the local variable value and the received one is achieved, and the greater value is preserved [55]. Miller introduced an algorithm to determine the critical path value in distributed applications that rely on message passing. As a cyclic directed graph, the Program Activity Graph (PAG) is characterized by nodes symbolizing the events marking the beginning and the end of the activity. The critical path is defined as the longest path for which the weights of all the segments are aggregated. The research contrasts a compacted algorithm, along with a distributed version that can be used in computing the critical path offline. The primary objective of that study is to use a parallel algorithm for expediting the offline calculation of the critical path [56]. Ali has proposed a new method to apply critical path analysis to intricate systems that consist of multiple interconnected state machines. This approach eliminates the need for time-consuming initial modeling, through the utilization of control-flow tracing, to uncover hidden software state machines automatically.

Additionally, the method uses iterative refinement to integrate necessary manual annotations with minimal effort [57].

4.4.1 MPI Critical Path Computation

The critical path computation of an MPI program takes place when a process in a rank waits for other unblocking events, from other processes or ranks, to complete. It is the execution path for which any time reduction improvement, along any component of the path, will result in a global time reduction improvement. Any linked time segment that has no idle nor wait time contributes to the critical path. Figure 4.1 illustrates an example of an MPI critical path computation. The MPI_Recv on the receiver rank (R0) is starting before the MPI_Send on the sender rank (R1) starts. The time spent on R0 before MPI_Send is called on R1 is an idle time. Once MPI_Send is called on R1, a compute time would start on R0. Both the idle and compute time in MPI_Recv are not contributing to the critical path. The time slot of MPI_Recv on R0, when MPI_Send on R1 starts sending the message, is considered as communication time. During this time, the receiver rank is busy receiving the message from R1. This time slot is contributing to the critical path.

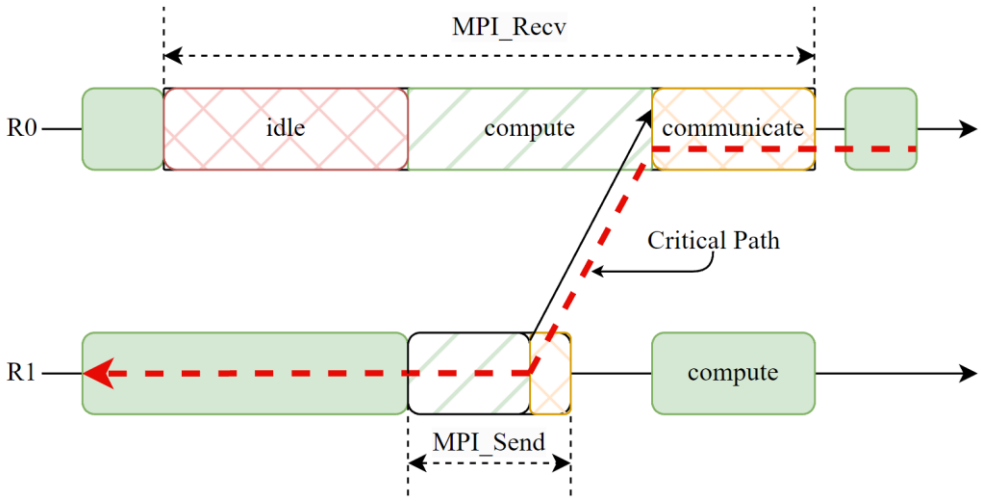


Figure 4.1 MPI critical path computation

Several studies were investigated in literature to understand the tools and frameworks proposed to measure the critical path of an MPI application. Schulz in his research presented a method for calculating the critical path of a simple MPI application with send/receive activities [5]. The proposed approach starts by collecting the trace files while executing the MPI application, creating sub-graphs with critical path information. After that, a post-mortem analysis was applied to merge

the sub-graphs into a global execution graph to extract the critical path. Hilbrich proposed a method to detect MPI deadlocks through the utilization of MPI execution graphs [58]. The proposed method involved a general deadlock model for MPI applications, which visualises these deadlocks.

Bak presented the MPI task graph scheduling algorithm which provides efficient synchronization and communication between MPI tasks [59]. This approach helped in reducing the critical path of MPI applications for a mixed sequence of computations and communications. A new metric was proposed by Denis to measure the overlap between MPI communications and computations, along with a benchmark suite to evaluate MPI overlapping strategies [60].

Chen presented a new method for detecting the MPI critical path candidates, using a potential critical path pool to capture the critical activities of MPI applications. Furthermore, a scalable performance modeling framework was proposed to record event traces, execution graphs, and critical path candidates [61]. Friese suggested another method to create MPI models based on a hierarchical critical path analysis. This approach captures the sequence of dependent operations, while disregarding any overlap [62]. This method begins by constructing a histogram of critical tasks, then parameterizes the instance counts and arguments using a scaling function. The algorithm employed to identify the critical path is known as “the longest path method”.

Hendriks introduced a graph-based representation of execution traces that takes into account the temporal order of tasks, critical path analysis, and trace distance computation [63]. Dietrich introduced an enhancement to the MPI performance measurement tool “Score-P” that allows it to monitor dependencies between the regions of MPI activity on the target device and host [64]. Furthermore, he defined the required rules to detect these dependencies, and implemented them in the CASITA analysis tool for critical path analysis.

Denys proposed a distributed critical path analysis algorithm that relies on the pre-computation of the execution graph from multiple trace files, without the need to aggregate all traces in one node [65]. Moraru proposed a new approach, based on an hash-table algorithm, that enhances the critical path time of an MPI application through the message-matching process [66]. This approach was developed mainly for point-to-point MPI communications.

4.4.2 MPI Software Trace Analysis Tools

MPI libraries can be traced and analyzed in different ways. The Radeon Open Compute platform (ROCm) can be used to trace MPI events in CTF format. the proposed framework in [41] utilized ROC-Profiler and ROC-Tracer libraries to capture MPI events running in CPU-GPU platforms. TAU is a tracing and profiling tool that can be used to trace MPI programs on multiple nodes [76]. This tool provides all the needed libraries to instrument, measure and analyze MPI applications. Score-P comprises an instrumentation framework with multiple runtime libraries and auxiliary tools. It focuses on the data collection aspect through gathering performance data that records information from different types of parallelism using a single tool. The instrumentation feature of Score-P allows embedding measurement probes into Fortran and C/C++ programs. When these probes are activated during a run, it starts collecting data about the performance of the program. Vampir is a trace visualization tool that can read Score-P traces, and visualize them for performance analysis [36]. Some MPI calls interact directly with computing devices like GPUs. These heterogeneous platforms can be traced at the runtime and device levels using Radeon Open Compute platform (ROCm) libraries. One approach described in [41] is to trace each source independently and combine every trace in the analysis phase, the other, supported by Tau, is to connect to the runtime libraries directly and combine everything in one trace.

4.5 MPI Critical Path Framework

The framework proposed in this study is based on three main modules: the MPI execution graph algorithm implementation, the critical path algorithm extension, and the experimental setup (development environment, data collection, validation).

4.5.1 MPI Execution Graph Algorithm

To calculate the critical path for an MPI application, it is necessary to generate the complete execution graph of the MPI program. The proposed MPI execution graph algorithm is shown in Figure 3.2. The flow starts with the Trace Compass parser; this is the initial stage where the system parses the trace data input file. OTF2 Group and OTF2 Events are components of the OTF2 format, along with the OTF2 event names patterns. The events are ProgramBegin, MPI_Send, MPI_Recv, MPI_Gather, MpiCollectiveBegin, and MpiCollectiveEnd; these represent various events stages in an MPI program. MPI_Send and MPI_Recv are used for message send and receive among

processes. MPI_Gather collects data from all processes and combines it into one buffer. MPICollectiveBegin and MPICollectiveEnd represent collective communication operations. A worker data structure is then created for each rank, to track their state. The worker is a data structure in Trace Compass that is used to track the events occurring in every MPI rank. For instance, if the traced application was executed on 8 ranks system, we will create 8 worker threads in Trace Compass; this involves creating worker processes, assigning tasks to each worker (represented as vertices), tracing the execution graph, and checking if all events have been processed.

The implementation of the proposed MPI execution graph algorithm is depicted in Figure 3.3. The proposed algorithm has a main procedure for checking all the incoming events in the trace file. Based on the patterns provided to the parser, the incoming event is analyzed. The global definition region in the OTF2 format has information about the system which includes events names, regions, locations, ranks, communicators, and groups. This information is needed in other analysis; hence, these events should be processed before starting the MPI events analysis. After that, the MPI event ProgramBegin and ProgramEnd analysis start; these are the trace events that indicate the beginning and end of MPI execution on a single rank. These events are represented by a horizontal vertex with (RUNNING) status. Next, MPI_Send, and MPI_Isend events are analyzed. It is important to save these events in a list, so it would be easy to link the receive events with its sender. These events are represented by a horizontal vertex with a RUNNING edge. After that, the MPI_Recv and MPI_Irecv events are analyzed. There are two main functions to consider, first, the horizontal link where we connect these events with the previous vertex via a horizontal edge, RUNNING, second, the vertical link, where we connect the receive vertex with its send vertex, via a vertical edge, NETWORK.

4.5.2 MPI Critical Path Algorithm

The generic critical path computation algorithm, at the operating system level, used in this research, was developed by [54], as seen in Figure 3.4. The author defines the active path of execution as “the execution path where all blocking edges are replaced by their corresponding subtask”.

4As seen in Figure 3.4, the main task states progress in order, with each traversed edge being incorporated into the active path. Upon encountering a blocked state, the trace of the incoming wake-up edge is initiated, leading to the commencement of the backward iteration. As we move backwards, the traversed edges are added to a local path at the beginning. If a packet is received,

its source is traced backwards. A recursive repetition is needed when facing a blocking edge during backward iteration. Upon reaching the beginning of the blocking interval, the backward iteration stops. After that, the forward iteration resumes after accumulating the added path into the active path. In this study, we consider the blocked state as the network state, where the MPI send is sending the message, while MPI receive is waiting for it. The running state of the algorithm is kept for normal MPI execution, which includes all MPI calls except the receive calls. The unknown state was ignored since there is no unknown state within MPI executions. The critical path algorithm, after adapting it for MPI applications, can be seen in Figure 3.5.

4.5.3 Environment Setup

The required setup for this study can be seen in Figure 3.6. The Slurm work environment was setup to have two servers, with a variable number of cores to represent the number of ranks needed for MPI applications. After setting up the Slurm environment, it is required to load the necessary libraries for MPI and Scorep, i.e., mpich, scorep, cubelib, and cubew. The commands used in the Slurm environment setup can be seen below:

```
salloc --time=1:0:0 --nodelist=epyc1,epyc2 --cpus-per-task n
```

where --time is the required time to run the experiments, –nodelist is the names of the assigned nodes, and –cpus-per-task is the number of processor cores.

Next, the MPI applications are compiled using the Scorep libraries to add the required instrumentation for tracing the execution of the application automatically. Once the compilation is completed, we can run the binary executable files. After the end of execution, the OTF2 trace files are ready for analysis. To use the trace files in Trace Compass, it is needed to convert them into the CTF format using the OTF2-to-CTF-converter tool from the DORSAL laboratory, which can be found online[52]. Finally, the converted trace files can be analyzed and visualized in Trace Compass.

4.5.4 MPI Applications

Please refer to section 3.4.

4.6 Results

4.6.1 MPI Execution Graph

The MPI execution graph algorithm was verified through the MPI CallStack analysis, available in Trace Compass. After running each test, the generated execution graph was tracked, based on the function calls monitored in the CallStack flame graph analysis. the summary of the generated execution graphs for the MPI test programs can be seen in Table 4.1.

Table 4.1 MPI execution graphs summary

Test	Tree File Size (Kilo Bytes)	Workers	Intervals
Multitask	69	3	36
Communicator	70	10	68
Hello	70	10	108
Ring	201	8	465
MonteCarlo8	266	8	580
Wave10	270	50	616
MonteCarlo50	2200	50	1104
Wave50	2233	50	1730

The MPI workers represent the IGraphWorker structure interface in Trace Compass. The worker is used to record the information about each rank, in terms of event starting time, rank number, MPI call name, current and previous event status. The intervals represent the number of horizontal and vertical transitions of the execution graph. As the number of ranks and the message transfer activities increase, an increase in the number of intervals is expected. A sample of a MPI execution graph view can be seen in Figure 4.2.

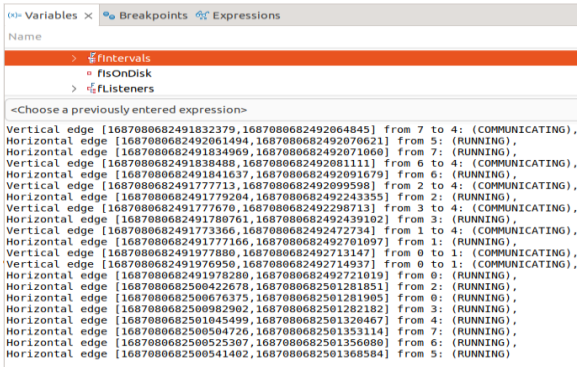


Figure 4.2 MPI execution graph view

The distribution of MPI events in the test applications can be seen in Figure 4.3. The most common events observed from the ScoreP tracer are Enter and Leave events. These events are used to reference OTF2 regions, where lower-level information can be found in the trace file. In every tested program, these events represent more than 60% of the trace events. As the number of ranks utilized to execute the MPI application increases, there is no change in the distribution of trace events; this can be seen in the MonteCarlo8/MonteCarlo50 and Wave10/Wave50 programs.

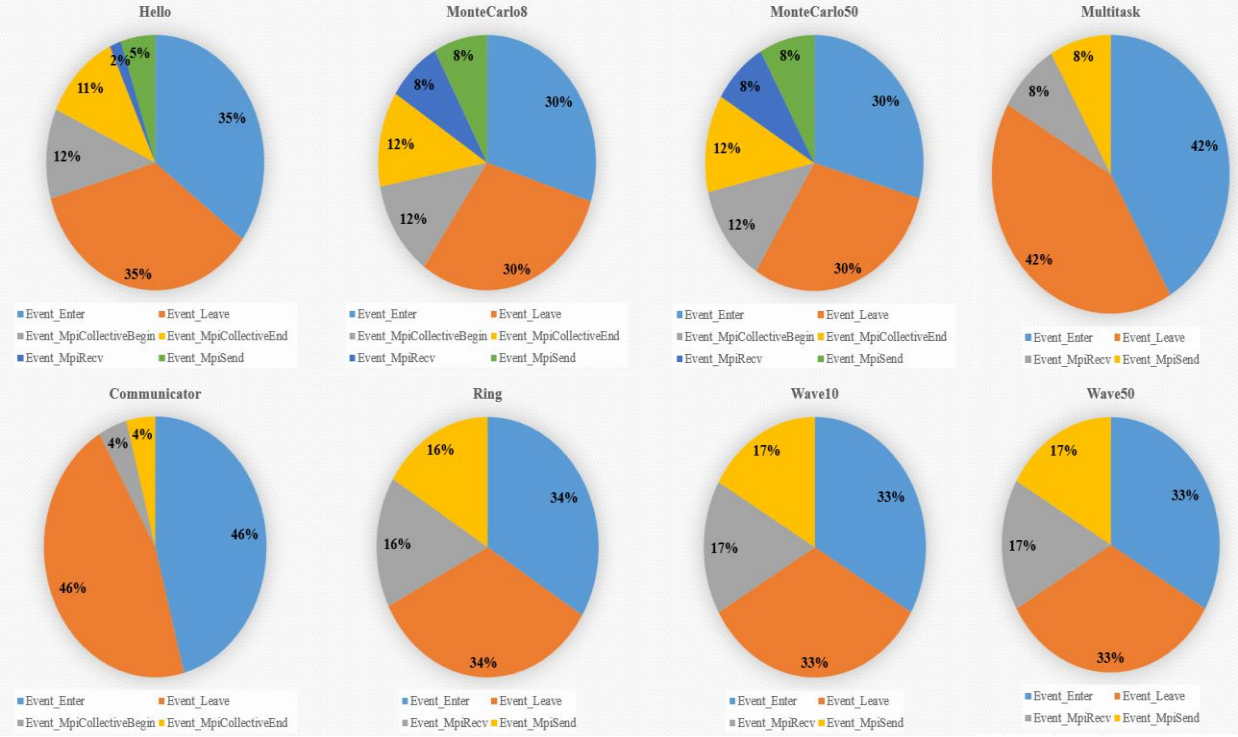


Figure 4.3 MPI events distribution per test application

The CallStack flame chart of test programs can be seen in Figure 4.4. This analysis was used to validate the MPI execution graph, by checking the timestamp of each vertex in the execution graph, for small-scale MPI programs that have MPI communication calls (MPI_Send, MPI_Isend, MPI_Recv, MPI_Scatter, MPI_Gather, MPI_Reduce, MPI_Ssend, MPI_Bsend, MPI_AllGather, and MPI_AllReduce). For larger MPI applications, like Wave50 and MonteCarlo50, random sampling was used to check the timestamp of events and compare it with the corresponding Trace Compass events table and the CallStack flame graph analysis. The overall time of the program execution was normalized to take samples every 10% of the events time. On that particular time slot, the events were checked against the CallStack flame graph and the MPI execution graph to match the produced execution graph nodes.

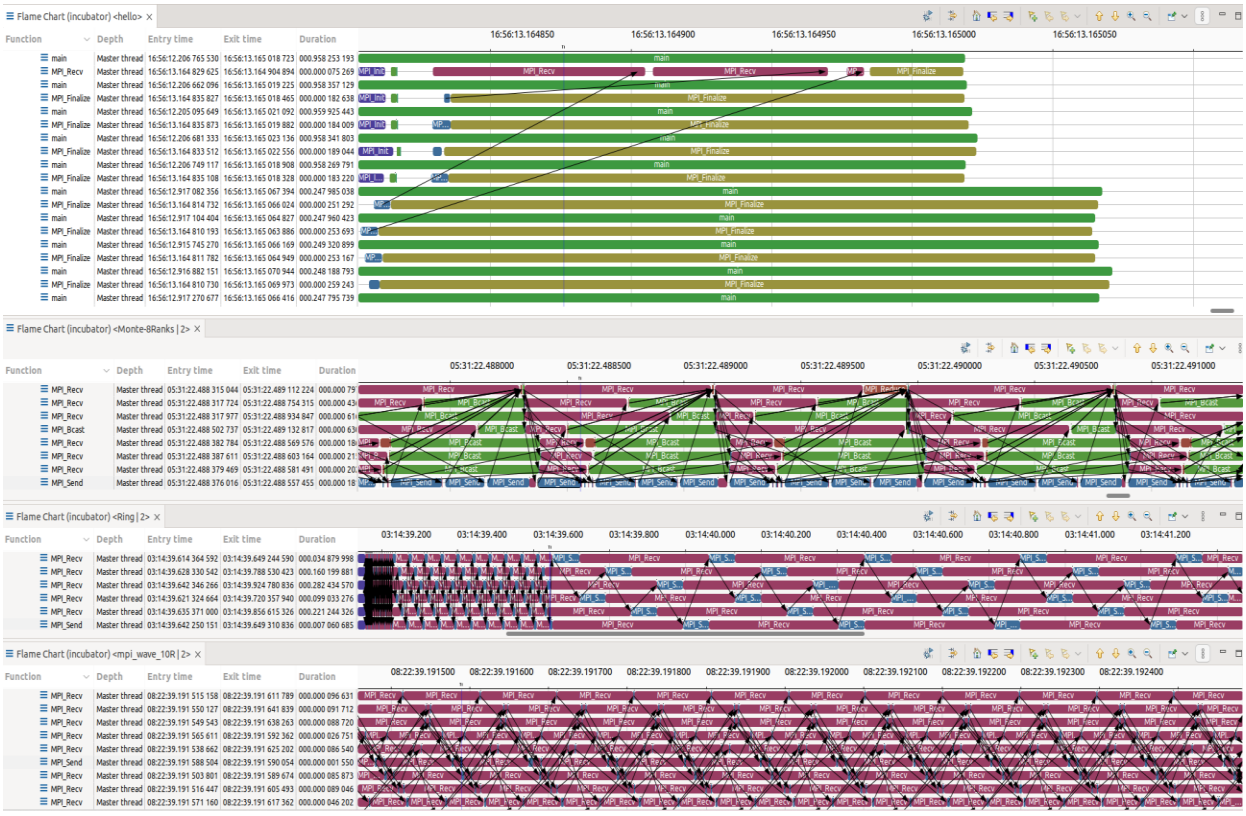


Figure 4.4 MPI CallStack flame chart

The duration of each MPI event, in seconds, can be seen in Figure 4.5. The longest event of an MPI program is MPI_Init. During this time, the rank is busy initializing the MPI execution environment.

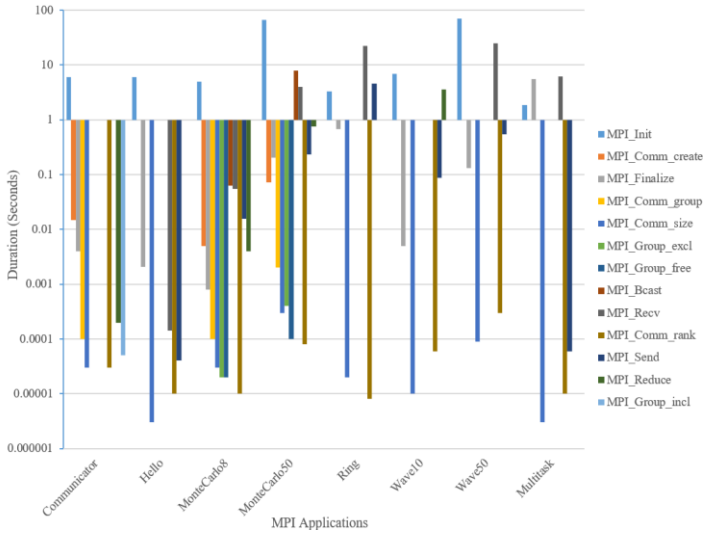
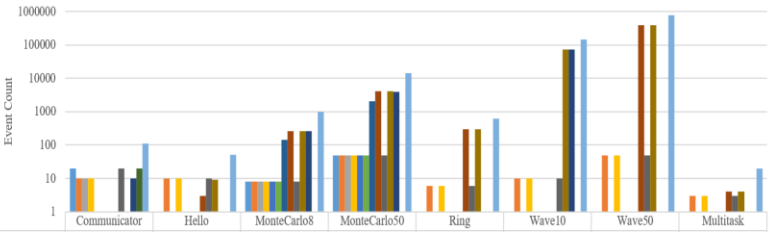


Figure 4.5 MPI calls durations in seconds (logarithmic scale)

The second most common MPI event is MPI_Recv. This event is called before a message transfer begins. The rank is working on preparing the buffers to receive the message and is waiting until the sender MPI_Send starts the communication. The MonteCarlo program used MPI_Bcast to broadcast messages from the root rank to all other ranks in the communicator group. There is a significant increase of this event occurrence between MonteCarlo8 and MonteCarlo50, due to the increased number of ranks from 8 to 50. The events count of each MPI call can be seen in Figure 4.6. Wave10 was executed on 10 ranks only. However, the number of events is higher than expected (compared to MonteCarlo50), due to the intensive send/receive activities among MPI ranks.



	Communicator	Hello	MonteCarlo8	MonteCarlo50	Ring	Wave10	Wave50	Multitask
MPI_Comm_create	20		8	50				
MPI_Finalize	10	10	8	50	6	10	50	3
MPI_Comm_group	10		8	50				
MPI_Comm_size	10	10	8	50	6	10	50	3
MPI_Group_excl			8	50				
MPI_Group_free			8	50				
MPI_Bcast			141	2059				
MPI_Recv		3	267	4019	300		392098	4
MPI_Comm_rank	20	10	8	50	6	10	50	3
MPI_Send		9	267	4019	300	72018	392098	4
MPI_Reduce	10		266	4018		72018		
MPI_Group_incl	20							
Total	110	52	1005	14515	624	144076	784396	20

Figure 4.6 MPI calls count (logarithmic scale)

4.6.2 MPI Critical Path

The critical path computation of MPI programs was achieved through the utilization of the Trace Compass generic kernel-level algorithm. The available algorithm was extended to support MPI states, which are “Running”, “Communicating”, and “Network”. The “Running” state refers to events that do not contribute delays to the program execution. When the execution graph detects a “Communicating” vertex, this flags that a communication between ranks occurs, and a vertical edge is added. The algorithm will resolve the network dependencies through a backward resolution, until it reaches the last vertex. During this time, any detected vertex will be labeled as “Network”. An example of MPI critical path visualization, for the MPI Multitask program can be seen in Figure 4.7.

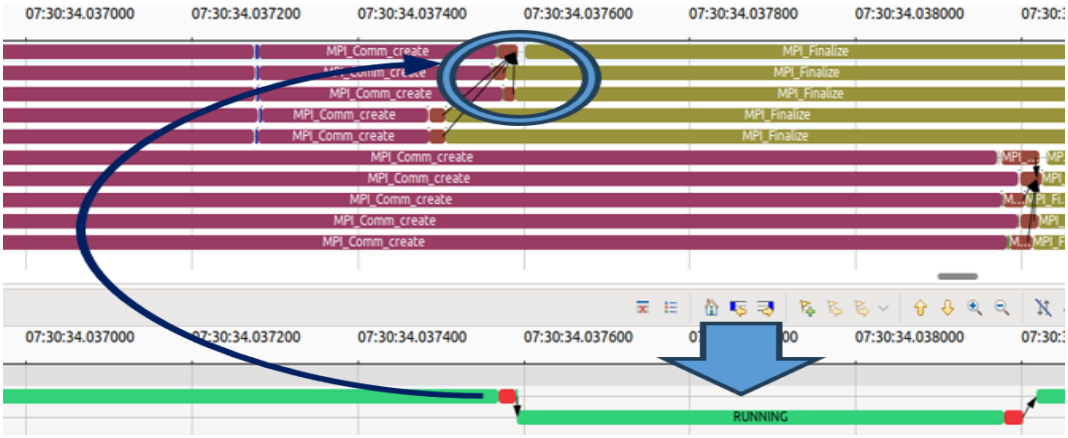


Figure 4.7 Critical path view for Multitask program

The green box represents the running state. During this time, the rank is busy with some computations. When it reaches the red box, this refers to the communicating state, where MPI_Recv receives messages from the other ranks. A black arrow denotes a “Network” state to another rank in the system. Another critical path view is illustrated in Figure 4.8, taken from the Communicator program execution. During MPI_Send, the state before the beginning of the arrow was marked green “Running”. When reaching the receiver rank, the MPI_Recv has started before the arrival of the message. However, this time was ignored by the critical path analysis, since it is a delay time. After the arrow area, the state reflects a communicating state, since the destination rank is receiving the message during that time.

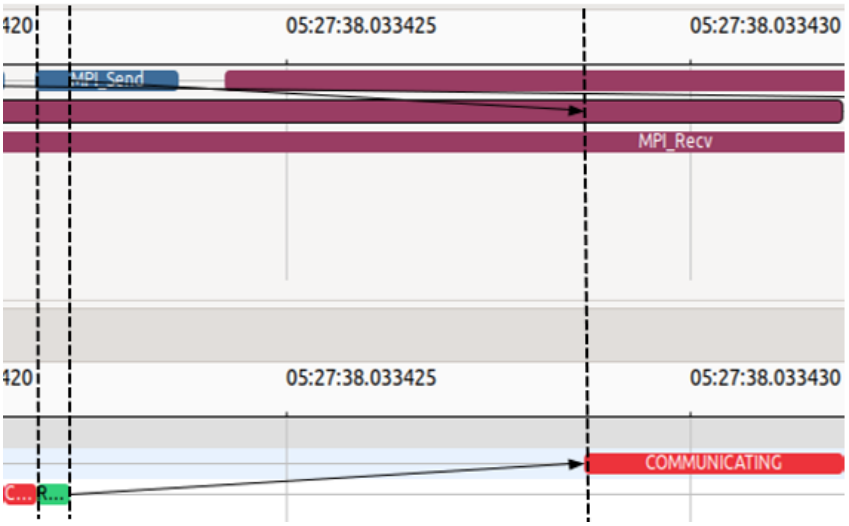


Figure 4.8 Critical path view of Communicator program

Another critical path view for the Communicator program can be found in Figure 4.9. This figure shows a snippet of the Communicator program, with the critical path computation at the bottom of the figure. Furthermore, another critical path computation can be found in the box on the left side, that shows the time spent on each rank, in percentage, during the execution of the program. Same view can be seen in Figure 4.10 and Figure 4.11 for the more complicated MPI programs of Ring and MonteCarlo50, respectively.

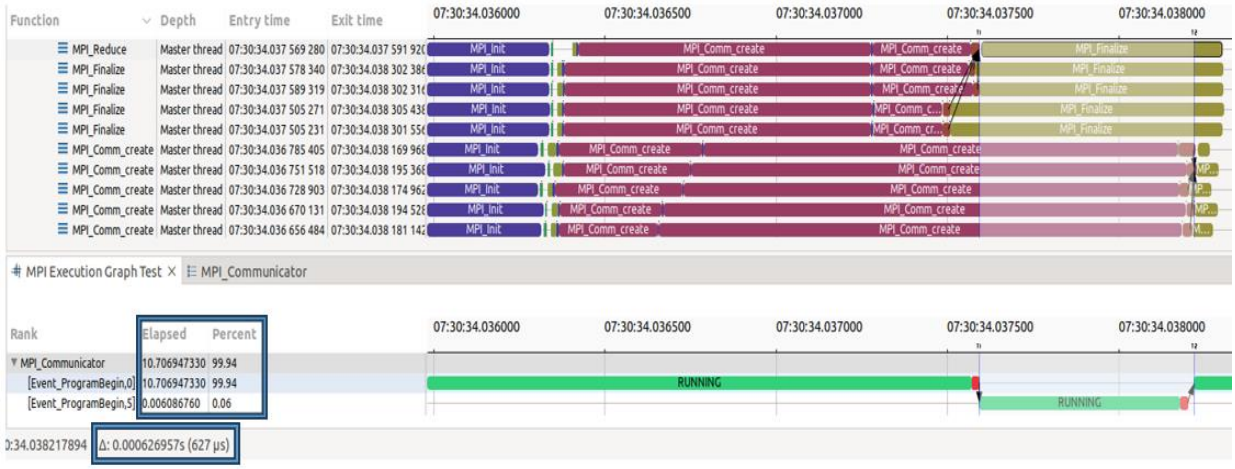


Figure 4.9 Critical path computation for Communicator program

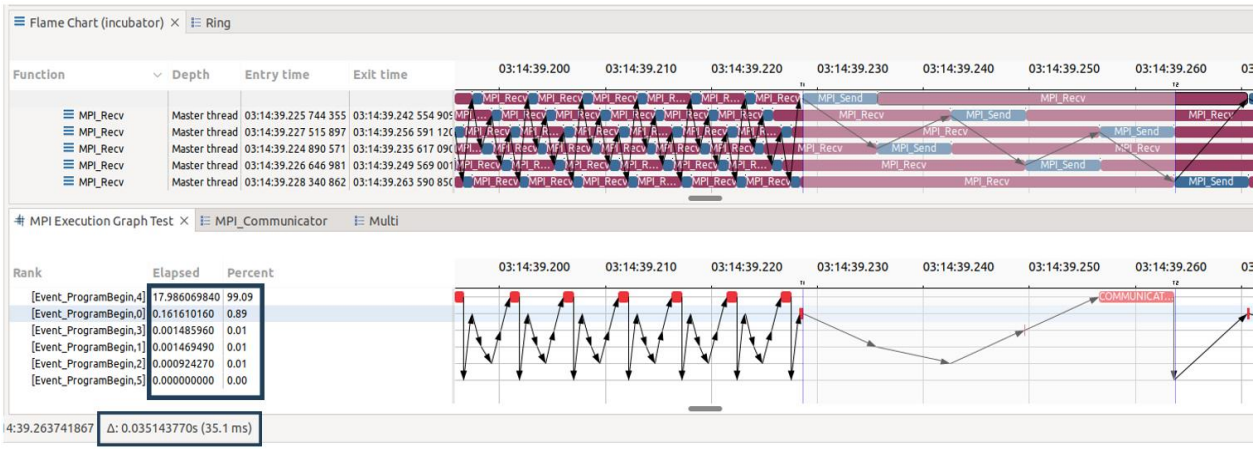
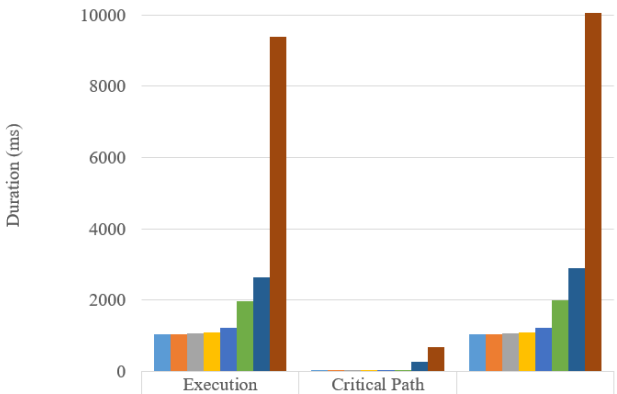


Figure 4.10 Critical path view for Ring program

Figure 4.11 Critical path view for MonteCarlo50 program

4.6.3 Framework Performance Analysis

The duration of the MPI execution graph construction, and of the critical path analysis, is illustrated in Figure 4.12. The proposed framework consumed more time during the execution graph analysis. This is expected, since the execution graph has a significant number of vertexes and edges, compared to the critical path graph. A simple MPI program, like Hello, required around 1 second to build the execution graph, and 0.218 ms to build the critical path graph. However, a complicated program, like Wave50, required around nine times the time needed for the Hello application, to build the execution graph, and around 3000 times to build the critical path graph. This is not surprising, given the complex communication nature of this program.



	Execution Graph Analysis (ms)	Critical Path Computation (ms)	Total (ms)
Hello	1027.555	0.218	1027.773
Multitask	1048.544	0.877	1049.421
Ring	1054.618	2.32	1056.938
Communicator	1099.721	0.542	1100.263
MonteCarlo8	1210.824	8.894	1219.718
MonteCarlo50	1967.523	14.248	1981.771
Wave10	2646.671	255.663	2902.334
Wave50	9381.653	684.644	10066.297

Figure 4.12 MPI execution graph and critical path analysis duration

The overhead of using the Score-P instrumentation can be seen in Table 4.2. As the application tends toward complexity, there is an increase in the executable file size after compiling it with Score-P instrumentation. The overhead represents the additional commands inserted by Score-P to collect the required data for tracing. The overhead was measured for both MPICH and OpenMPI libraries. The difference in file sizes between both implementations is due to the different compilation libraries used by each implementation.

Table 4.2 Score-P instrumentation file size overhead (MPICH vs. OpenMPI)

	MPICH File Size (KB)			**OpenMPI File Size (KB)**		
Test	**Executable**	**With Score-P**	**Overhead (%)**	**Executable**	**With Score-P**	**Overhead (%)**
Hello	17	22	22.727	17	25	32.000
Ring	17	27	37.037	17	26	34.615
Communicator	18	28	35.714	18	27	33.333
MonteCarlo50	18	30	40.000	18	28	35.714
MonteCarlo8	18	30	40.000	18	28	35.714
Multitask	18	29	37.931	18	28	35.714
Wave10	22	35	37.143	22	40	45.000
Wave50	22	35	37.143	22	40	45.000

Table 4.3 Execution time overhead (MPICH vs. OpenMPI)

	MPICH Execution time (s)		OpenMPI Execution time (s)		
Test	**Executable**	**Score-P Overhead (%)**	**Executable**	**Score-P Overhead (%)**	**LTTng Overhead (%)**
Hello	1.001	10.545	1.080	41.935	51.131
Ring	8.250	9.371	2.810	33.412	31.296
Communicator	0.993	16.203	1.140	40.625	45.972
MonteCarlo50	3.148	40.072	2.950	50.915	55.303
MonteCarlo8	1.120	22.545	1.180	45.37	53.906
Multitask	6.987	6.591	8.410	6.452	6.659
Wave10	1.432	28.292	2.130	33.438	37.353
Wave50	4.464	40.488	3.610	47.908	50.138

The execution time overhead was measured for both MPICH and OpenMPI compilations, as seen in Table 4.3. With applications running with a higher number of ranks, there is a noticeable increase in the execution time of the application. This is due to the numerous MPI communication events between ranks, which cause more data events to be recorded by the trace tool. The execution libraries for each implementation are different, and hence a difference in the execution time for the executable files is expected. OpenMPI was used for the LTTng UST runtime instrumentation library, since it only supports OpenMPI. The purpose of this execution was to evaluate Score-P and LTTng UST instrumentation overhead. As a result, the overhead was increased when running with both Score-P and LTTng instrumentations, with the OpenMPI implementation. This experiment was required to demonstrate the effectiveness of the proposed framework in showing kernel-level and MPI user-level events. It is important to visualize the system behavior on both levels, to provide software engineers with the full information about the system, during MPI execution. The kernel and user events views can be seen in Figure 4.13.

Figure 4.13 The proposed framework kernel and user events view (not Synchronized)

Figure 4.13 demonstrates the top-down information view. The upper portion represents the control flow of the operating system. This is the level where kernel events can be viewed, with scheduling events showing when each thread is running, blocked waiting, or preempted. The second portion represents the aggregated critical path analysis for executing the mpirun command. The third view illustrates the critical path analysis of the operating system. Thereafter, the MPI CallStack analysis can be seen, and finally, the critical path analysis for the MPI events.

4.7 Conclusion

The performance improvement of parallel programs is important for efficient multi-core system usage. As a result, tracing and visualizing events are crucial in the parallel programming development paradigm. With the right toolset, the developers can detect program performance issues and optimize their application. This study presented the development of the MPI critical path analysis framework, based on Trace Compass. The proposed framework was derived by developing a new algorithm to build an MPI execution graph, followed by the integration with the generic kernel-level critical path bounded algorithm used by Trace Compass. Several testing scenarios were used to validate the MPI execution graphs and the critical path computations, based on the

observations from the events table and the CallStack flame graph analysis. Due to its scalability, the proposed MPI execution graph algorithm can easily be extended to analyze any MPI group communication not currently covered.

CHAPTER 5 GENERAL DISCUSSION

There is a growing need to optimize the performance of parallel programs. Furthermore, it is challenging to debug performance issues without having the proper tools for that task. As a result, this study conducted the development of a critical path computation framework for MPI applications. In this chapter, we review the research objectives and discuss how the outcome of this study helped in fulfilling them.

5.1 Summary of Research

This study proposed a framework for MPI critical path computation, based on Trace Compass. The framework was developed in three steps. First, we developed an algorithm to build the execution graph of MPI applications, with the needed nodes and edges to express the program behavior. Three contexts were proposed for MPI calls, running, communicating, and networking, as illustrated in Chapter 3. The second step is the implementation and integration of the MPI execution graph algorithm with Trace Compass. Finally, the kernel-level critical path algorithm of Trace Compass was extended to account for MPI events.

The proposed framework was validated through the use of Score-P to record events trace files. Furthermore, some statistical analyses of overhead were conducted to compute the overhead caused by the instrumentation in terms of file size, and execution time. Using Score-P as an instrument had a slight overhead in terms of executable file size and execution time of the application. It is worth mentioning that this overhead varies based on the nature of the application in terms of the number of ranks, and the MPI communication approaches used. The overhead was calculated based on 120 samples of execution time data, with and without instrumentation. This test was automated by a script implemented to run on the SLURM server nodes. This script starts by compiling the available applications for the test. After that, it records the size of each application, instrumented, and not instrumented. Then, the script triggers the execution of each application and records the execution time. The tester can set the number of executions based on the preferred test scenario. Finally, the collected data for the execution time was averaged and presented in Chapter 4.

5.2 Research Milestones

This study was motivated by the need to close the gap found in literature in providing proper performance analysis tools for MPI libraries. Most of the developed algorithms in the literature were based on a standalone implementation. This means, when using any of these approaches, that the critical path of MPI applications is computed solely based on the MPI events. However, this does not provide a detailed view and analysis, that takes into account operating system kernel events, for the developers. It is crucial to provide such capability to fully understand the execution behavior of such complicated applications. For instance, viewing the CallStack of MPI events, along with the critical path view, provides a better understanding of what is happening in the system, during a specific event or time.

The work presented in Chapter 4 showed the proposed framework effectiveness in computing and visualizing the critical path of several MPI applications. Multiple views were shown to demonstrate the value of having such a framework, integrated with Trace Compass. This opens significant opportunities in enhancing the performance of parallel programs based on MPI libraries. The trace files for each MPI test were collected by the Score-P instrumentation libraries. After that, the generated traces were converted to a proper format for Trace Compass compatibility.

The proposed execution graph algorithm was validated based on the events table and CallStack analysis flame graph in Trace Compass. This was a critical step to ensure the correctness of this graph, since it is used as an input to the extended critical path algorithm. The execution graph implementation was integrated with Trace Compass data providers, to make it available as a java analysis module plug-in, in the Trace Compass interface. Further work was implemented to create the critical path graph interface, by developing the time graph providers and listener modules.

5.3 Limitations

Even though the proposed framework helped in analyzing the critical path of MPI applications, a few limitations remain. First, the developed execution graph was created for the OTF2 trace format. If another trace format is used, minor modifications might be needed to recognize the possibly different MPI events in the new trace format. In addition, the proposed framework does not yet account for the all-to-all MPI communications. Further implementation work is needed on the MPI CallStack analysis to compute the state system for this. Without this analysis, it might be

challenging to validate the results of the execution graph. The proposed framework used Score-P for instrumenting MPI executions. However, this is not the only instrumentation tool available for MPI applications. ROCm and LTTng can now provide some MPI event traces. As a result, further work will be required to adapt the MPI execution graph algorithm for new trace formats.

CHAPTER 6 CONCLUSION AND RECOMMENDATIONS

6.1 Summary

The complexity of modern HPC applications makes the debugging and performance optimization of parallel programs to be a challenging task. As a result, tracing and visualizing events are crucial in the parallel programming development paradigm. With the right tools, the developers can detect program performance issues and come up with optimizations. This research aims to develop a framework that can be used to compute the critical path of MPI programs at a detailed level, based on the Trace Compass analysis and visualization tool.

The contribution of this study was achieved through developing an MPI execution graph builder algorithm, followed by the integration of this algorithm within the Trace Compass incubator environment for testing. Furthermore, an extension of the generic kernel-level critical path algorithm from Trace Compass was derived, to compute the critical path of MPI applications. This extension included adding the context states of the MPI system when running, communicating, and networking. A few testing scenarios were used to validate the MPI execution graphs and the critical path computations, based on the observation of events and the CallStack flame graph analysis.

6.2 Objective Achievement

This study demonstrated the possibility of developing efficient multi-level trace analysis modules for MPI in Trace Compass. It provides MPI critical path computation and analysis features to Trace Compass. With such analysis, HPC developers can leverage a wide range of MPI performance analyses and visualizations to enhance their user experience with Trace Compass. Several test scenarios were produced to validate the proposed framework, which included several MPI events in terms of point-to-point, root-to-all, and all-to-root communications. Finally, further case studies can be examined to demonstrate the effectiveness of the proposed solution.

6.3 Recommendations and Future Work

Due to its scalability, the proposed MPI execution graph algorithm can be extended to analyze the MPI all-to-all communications in Trace Compass. This analysis is not yet available for the OTF2 MPI CallStack analysis. Once available, the proposed MPI execution graph algorithm can be

extended to cover the all-to-all communications. An integration of the new proposed MPI events handlers, with the available Linux operating system events handlers, will be a good step forward to compute and visualize lower-level kernel events, for better performance optimization and bottlenecks detection.

REFERENCES

[1] A. Spear, M. Levy, and M. Desnoyers, “Using tracing to solve the multicore system debug problem,” Computer, vol. 45, no. 12, pp. 60–64, 2012.

[2] S. M. Ghazimirsaeed, “High-performance Communication in MPI through Message Matching and Neighborhood Collective Design,” PhD Thesis, Queen’s University (Canada), 2019.

[3] H.-V. Dang et al., “A lightweight communication runtime for distributed graph analytics,” in 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2018, pp. 980–989.

[4] A. Bahmani and F. Mueller, “ACURDION: An adaptive clustering-based algorithm for tracing large-scale MPI applications,” in 2015 IEEE International Conference on Big Data (Big Data), IEEE, 2015, pp. 785–792.

[5] M. Schulz, “Extracting critical path graphs from MPI applications,” in 2005 IEEE International Conference on Cluster Computing, IEEE, 2005, pp. 1–10.

[6] F. Nielsen and F. Nielsen, “Introduction to MPI: the message passing interface,” Introd. HPC MPI Data Sci., pp. 21–62, 2016.

[7] L. Clarke, I. Glendinning, and R. Hempel, “The MPI message passing interface standard,” in Programming Environments for Massively Parallel Distributed Systems: Working Conference of the IFIP WG 10.3, April 25–29, 1994, Springer, 1994, pp. 213–218.

[8] S. Vanecek, “Design, Implementation and Test of Efficient GPU to GPU Communication Methods,” Department of Informatics, Technical University of Munich, 2020.

[9] F. Li, F. Zou, and J. Rao, “A multi-GPU and CUDA-aware MPI-based spectral element formulation for ultrasonic wave propagation in solid media,” Ultrasonics, p. 107049, 2023.

[10] K. V. Manian, A. A. Ammar, A. Ruhela, C.-H. Chu, H. Subramoni, and D. K. Panda, “Characterizing CUDA Unified Memory (UM)-Aware MPI Designs on Modern GPU Architectures,” in Proceedings of the 12th Workshop on General Purpose Processing Using GPUs, Providence RI USA: ACM, Apr. 2019, pp. 43–52. doi: 10.1145/3300053.3319419.

[11] T. Hoefler, A. Lumsdaine, and W. Rehm, “Implementation and performance analysis of non-blocking collective operations for MPI,” in Proceedings of the 2007 ACM/IEEE conference on Supercomputing, 2007, pp. 1–10.

[12] M. Si and P. Balaji, “Process-based asynchronous progress model for MPI point-to-point communication,” in 2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), IEEE, 2017, pp. 206–214. Accessed: Oct. 12, 2023. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8291930/

[13] A. Thune, S.-A. Reinemo, T. Skeie, and X. Cai, “Detailed modeling of heterogeneous and contention-constrained point-to-point MPI communication,” IEEE Trans. Parallel Distrib. Syst., vol. 34, no. 5, pp. 1580–1593, 2023.

[14] R. Thakur, R. Rabenseifner, and W. Gropp, “Optimization of Collective Communication Operations in MPICH,” Int. J. High Perform. Comput. Appl., vol. 19, no. 1, pp. 49–66, Feb. 2005, doi: 10.1177/1094342005051521.

[15] J. L. Träff, S. Hunold, G. Mercier, and D. J. Holmes, “MPI collective communication through a single set of interfaces: A case for orthogonality,” Parallel Comput., vol. 107, p. 102826, 2021.

[16] D. J. Holmes et al., “Partitioned collective communication,” in 2021 Workshop on Exascale MPI (ExaMPI), IEEE, 2021, pp. 9–17. Accessed: Oct. 12, 2023. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9652828/

[17] S. D. Sharma and M. Dagenais, “Hardware-assisted instruction profiling and latency detection,” J. Eng., vol. 2016, no. 10, pp. 367–376, 2016.

[18] T. Dontje et al., “Sun HPC ClusterToolsTM 7+: A Binary Distribution of Open MPI,” in Tools for High Performance Computing, M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 3–18. doi: 10.1007/978-3-540-68564-7_1.

[19] J. Daniel and D. Ladd, “Semantic Diff: a tool for summarizing the effects of modifications,” Proc. 1994 Int. Conf. Softw. Maint., pp. 243–252, 1994, doi: 10.1109/ICSM.1994.336770.

[20] M. A. Feliú Gabaldón, “Logic-based techniques for program analysis and specification synthesis,” PhD Thesis, Universitat Politècnica de València, 2013.

[21] T. Ravitch, E. R. Creswick, A. Tomb, A. Foltzer, T. Elliott, and L. Casburn, “Multi-app security analysis with fuse: Statically detecting android app collusion,” in Proceedings of the 4th Program Protection and Reverse Engineering Workshop, 2014, pp. 1–10.

[22] “Owasp. Source code analysis tools.” [Online]. Available: https://owasp.org/www-community/Source_Code_Analysis_Tools

[23] I. F. De Melo, Software Tracing Comparison Using Data Mining Techniques. in Mémoire de maîtrise. École polytechnique de Montréal, 2017. [Online]. Available: https://books.google.ca/books?id=QtaWzgEACAAJ

[24] S. Shende, “Profiling and tracing in linux,” in Proceedings of the Extreme Linux Workshop, 1999.

[25] S. Fan, R. Keller, and M. Resch, “Enhanced Memory debugging of MPI-parallel Applications in Open MPI,” in Tools for High Performance Computing, M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 49–60. doi: 10.1007/978-3-540-68564-7_4.

[26] J.-B. Besnard, “Profiling and debugging by efficient tracing of hybrid multi-threaded HPC applications.,” PhD Thesis, Université de Versailles Saint Quentin en Yvelines, 2014. Accessed: Oct. 18, 2023. [Online]. Available: https://theses.hal.science/tel-01102639/

[27] G. Bronevetsky, “Communication-sensitive static dataflow for parallel message passing applications,” in 2009 International Symposium on Code Generation and Optimization, IEEE, 2009, pp. 1–12.

[28] L. Pollock and S. Sprenkle, “Program Flow Graph Construction for Static Analysis of MPI Programs”.

[29] M. M. Strout, B. Kreaseck, and P. D. Hovland, “Data-flow analysis for MPI programs,” in 2006 International Conference on Parallel Processing (ICPP’06), IEEE, 2006, pp. 175–184.

[30] M. Perscheid, B. Siegmund, M. Taeumel, and R. Hirschfeld, “Studying the advancement in debugging practice of professional software developers,” Softw. Qual. J., vol. 25, pp. 83–110, 2017.

[31] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google file system,” in Proceedings of the nineteenth ACM symposium on Operating systems principles, Bolton Landing NY USA: ACM, Oct. 2003, pp. 29–43. doi: 10.1145/945445.945450.

[32] E. Knapp, “Deadlock detection in distributed databases,” ACM Comput. Surv., vol. 19, no. 4, pp. 303–328, Dec. 1987, doi: 10.1145/45075.46163.

[33] T. Halpin and T. Morgan, “Information Modeling and Relational Databases,” DATABASE Netw. J., vol. 38, no. 3, p. 8, 2008.

[34] GNU Project, “Debugging with GDB.” [Online]. Available: https://sourceware.org/gdb/current/onlinedocs/gdb.html/

[35] A. Hamou-Lhadj, T. Lethbridge, and L. Fu, “Challenges and requirements for an effective trace exploration tool,” 12th IEEE Int. Workshop Program Comprehension, vol. Proceedings, pp. 70–78, 2004, doi: 10.1109/WPC.2004.1311049.

[36] W. Williams and H. Brunst, “Parallel Performance Engineering using Score-P and Vampir,” in Companion of the 2023 ACM/SPEC International Conference on Performance Engineering, Coimbra Portugal: ACM, Apr. 2023, pp. 121–125. doi: 10.1145/3578245.3583715.

[37] A. Knüpfer et al., “Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir,” in Tools for High Performance Computing 2011, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91.

[38] R. Beamonte and M. R. Dagenais, “Linux low-latency tracing for multicore hard real-time systems,” Adv. Comput. Eng., vol. 2015, 2015, Accessed: Nov. 21, 2023. [Online]. Available: https://downloads.hindawi.com/archive/2015/261094.pdf

[39] M. Rezazadeh, “Performance Analysis of Complex Multi-Thread Applications Through Critical Path Analysis,” PhD Thesis, Ecole Polytechnique, Montreal (Canada), 2019. Accessed: Nov. 23, 2023. [Online]. Available: https://search.proquest.com/openview/ed8de8b6c55f94fe5d1cf1094106d1fb/1?pq-origsite=gscholar&cbl=18750&diss=y

[40] T. Compass, “Trace compass,” En Ligne Httpsprojects Eclipse Org, 2015.

[41] A. Fiorini and M. R. Dagenais, “Visualization of profiling and tracing in CPU-GPU programs,” Concurr. Comput. Pract. Exp., vol. 34, no. 23, p. e7188, Oct. 2022, doi: 10.1002/cpe.7188.

[42] “TAU - Tuning and Analysis Utilities -.” Accessed: Nov. 23, 2023. [Online]. Available: https://www.cs.uoregon.edu/research/tau/home.php

[43] A. Leko, H. Sherburne, H. Su, B. Golden, and A. D. George, “Practical experiences with modern parallel performance analysis tools: an evaluation,” in Parallel and Distributed Processing, IPDPS 2008 IEEE Symposium, 2008, pp. 14–18. Accessed: Nov. 23, 2023. [Online]. Available: https://upc.lbl.gov/ppw-archive/publications/WhitepaperLeko.pdf

[44] L. Adhianto et al., “HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs Http://Hpctoolkit.Org,” Concurr. Comput. Pract. Exp., vol. 22, no. 6, p. 685–701, Apr. 2010.

[45] Y. Zhou and C. Artho, “TC4JPF: Using Trace Compass to Visualize JPF Traces,” ACM SIGSOFT Softw. Eng. Notes, vol. 46, no. 3, pp. 42–46, Jul. 2021, doi: 10.1145/3468744.3468757.

[46] A. Knüpfer et al., “The Vampir Performance Analysis Tool-Set,” in Tools for High Performance Computing, M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 139–155.

[47] O. Zaki, E. Lusk, W. Gropp, and D. Swider, “Toward Scalable Performance Visualization with Jumpshot,” Int. J. High Perform. Comput. Appl., vol. 13, no. 3, pp. 277–288, Aug. 1999, doi: 10.1177/109434209901300310.

[48] “HPCToolkit Home.” Accessed: Nov. 23, 2023. [Online]. Available: http://hpctoolkit.org/

[49] M. Wagner, A. Knupfer, and W. E. Nagel, “Enhanced encoding techniques for the open trace format 2,” Procedia Comput. Sci., vol. 9, pp. 1979–1987, 2012.

[50] Y. Zhou, “Execution trace visualization for Java Pathfinder using Trace Compass.” 2020. Accessed: Dec. 19, 2023. [Online]. Available: https://www.diva-portal.org/smash/record.jsf?pid=diva2:1503549

[51] Linux Foundation Diamon Workgroup, “Common Trace Format.” [Online]. Available: https://diamon.org/ctf/v1.8.3/

[52] “OTF2 to CTF converter.” DORSAL, Jul. 05, 2023. Accessed: Oct. 12, 2023. [Online]. Available: https://github.com/dorsal-lab/OTF2-to-CTF-converter

[53] F. Giraldeau, “Analyse de performance de systèmes distribués et hétérogènes à l’aide de traçage noyau,” PhD Thesis, École Polytechnique de Montréal, 2015.

[54] F. Giraldeau and M. Dagenais, “Wait Analysis of Distributed Systems Using Kernel Tracing,” IEEE Trans. Parallel Distrib. Syst., vol. 27, no. 8, pp. 2450–2461, 2016.

[55] J. K. Hollingsworth, “An online computation of critical path profiling,” in Proceedings of the SIGMETRICS symposium on Parallel and distributed tools - SPDT ’96, Philadelphia, Pennsylvania, United States: ACM Press, 1996, pp. 11–20. doi: 10.1145/238020.238024.

[56] C.-Q. Yang and B. P. Miller, “Critical path analysis for the execution of parallel and distributed programs,” in The 8th International Conference on Distributed, IEEE Computer Society, 1988, pp. 366–367. Accessed: Sep. 23, 2023. [Online]. Available: https://ftp.cs.wisc.edu/paradyn/papers/CritPath-ICDCS1988.pdf

[57] A. G. Saidi, N. L. Binkert, S. K. Reinhardt, and T. Mudge, “Full-system critical path analysis,” in ISPASS 2008-IEEE International Symposium on Performance Analysis of Systems and

software, IEEE, 2008, pp. 63–74. Accessed: Sep. 23, 2023. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4510739/

[58] T. Hilbrich, B. R. de Supinski, M. Schulz, and M. S. Müller, “A graph based approach for MPI deadlock detection,” in Proceedings of the 23rd international conference on Supercomputing, 2009, pp. 296–305.

[59] S. Bak, O. Hernandez, M. Gates, P. Luszczek, and V. Sarkar, “Task-graph scheduling extensions for efficient synchronization and communication,” in Proceedings of the Acm International Conference on Supercomputing, 2021, pp. 88–101.

[60] A. Denis and F. Trahay, “MPI overlap: Benchmark and analysis,” in 2016 45th International Conference on Parallel Processing (ICPP), IEEE, 2016, pp. 258–267.

[61] J. Chen and R. M. Clapp, “Critical-path candidates: scalable performance modeling for MPI workloads,” in 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE, 2015, pp. 1–10.

[62] R. D. Friese, N. R. Tallent, A. Vishnu, D. J. Kerbyson, and A. Hoisie, “Generating performance models for irregular applications,” in 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2017, pp. 317–326.

[63] M. Hendriks, J. Verriet, T. Basten, B. Theelen, M. Brassé, and L. Somers, “Analyzing execution traces: critical-path analysis and distance analysis,” Int. J. Softw. Tools Technol. Transf., vol. 19, pp. 487–510, 2017.

[64] R. Dietrich, F. Schmitt, A. Grund, and J. Stolle, “Critical-blame analysis for OpenMP 4.0 offloading on Intel Xeon Phi,” J. Syst. Softw., vol. 125, pp. 381–388, 2017.

[65] P.-F. Denys, Q. Fournier, and M. R. Dagenais, “Distributed computation of the critical path from execution traces,” Softw. Pract. Exp., 2023.

[66] M. Moraru, A. Roussel, H. Taboada, C. Jaillet, M. Pérache, and M. Krajecki, “Performance improvements of parallel applications thanks to MPI-4.0 Hints,” in 2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), IEEE, 2022, pp. 273–282.

[67] J. Burkardt, “MPI Sample Codes.” [Online]. Available: https://people.math.sc.edu/Burkardt/cpp_src/cpp_src.html

[68] F. Schmitt, R. Dietrich, and G. Juckeland, “Scalable critical-path analysis and optimization guidance for hybrid MPI-CUDA applications,” Int. J. High Perform. Comput. Appl., vol. 31, no. 6, pp. 485–498, 2017, doi: 10.1177/1094342016661865.

[69] D. Böhme, F. Wolf, B. R. de Supinski, M. Schulz, and M. Geimer, “Scalable critical-path based performance analysis,” in 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IEEE, 2012, pp. 1330–1340. Accessed: Nov. 10, 2023. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6267934/

[70] A. Shatalin, V. Slobodskoy, and M. Fatin, “Root Causing MPI Workloads Imbalance Issues via Scalable MPI Critical Path Analysis,” in Supercomputing, vol. 13708, V. Voevodin, S. Sobolev, M. Yakobovskiy, and R. Shagaliev, Eds., in Lecture Notes in Computer Science, vol. 13708. , Cham: Springer International Publishing, 2022, pp. 501–521. doi: 10.1007/978-3-031-22941-1_37.

[71] S. Böhm, M. Běhálek, O. Meca, and M. Šurkovskỳ, “Visual programming of MPI applications: debugging, performance analysis, and performance prediction,” Comput. Sci. Inf. Syst., vol. 11, no. 4, pp. 1315–1336, 2014.

[72] T. Bertauld and M. Dagenais, “Low-level trace correlation on heterogeneous embedded systems,” EURASIP J. Embed. Syst., vol. 2017, 2017, doi: 10.1186/s13639-016-0067-1.

[73] M. Rezazadeh, N. Ezzati-Jivan, S. V. Azhari, and M. R. Dagenais, “Performance evaluation of complex multi-thread applications through execution path analysis,” Perform. Eval., vol. 155, p. 102289, 2022.

[74] J. Doerfert, J. Huber, and M. Cornelius, “Advancing OpenMP Offload Debugging Capabilities in LLVM,” in 50th International Conference on Parallel Processing Workshop, Lemont IL USA: ACM, Aug. 2021, pp. 1–8. doi: 10.1145/3458744.3473358.

[75] D. D. Nguyen, “Workflow Critical Path: a Data-Oriented Path Metric for Holistic HPC Workflows,” 2020, Accessed: Nov. 18, 2023. [Online]. Available: https://pdxscholar.library.pdx.edu/open_access_etds/5495/

[76] N. Chaimov, S. Shende, A. Malony, M. Gorentla Venkata, and N. Imam, “Tracking Memory Usage in OpenSHMEM Runtimes with the TAU Performance System,” in OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity, vol. 11283, S. Pophale, N. Imam, F. Aderholdt, and M. Gorentla Venkata, Eds., in Lecture Notes in Computer Science, vol. 11283. , Cham: Springer International Publishing, 2019, pp. 167–179. doi: 10.1007/978-3-030-04918-8_11.