Diese Arbeit wurde vorgelegt am
Lehrstuhl für Hochleistungsrechnen (Informatik 12), IT Center.

Evaluierung einer alternativen Kostenfortpflanzung im Delay-Kostenmodell von Scalasca

Evaluating an alternative cost-propagation in the delay-cost model of Scalasca

Masterarbeit

Dominik Viehhauser
Matrikelnummer: 354322

Aachen, den 21. März 2023

Communicated by Prof. Matthias S. Müller

Erstgutachter:	Prof. Dr. rer. nat. Matthias Müller (’)
Zweitgutachter:	Prof. Dr. rer. nat. Felix Wolf (*)
Betreuer:	Marc-André Hermanns, Dr. rer. nat. (’)

(’) Lehrstuhl für Hochleistungsrechnen, RWTH Aachen University
IT Center, RWTH Aachen University
(*) Lehrstuhl für Parallele Programmierung, TU Darmstadt

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden.

Aachen, den 21. März 2023

Kurzfassung

Die fortschreitende Entwicklung moderner HPC-Systeme erhöht den Bedarf an optimierten Anwendungen, um den wissenschaftlichen Erkenntnisgewinn pro verbrauchter Core-Stunde zu maximieren. Werkzeuge zur Leistungsanalyse ermöglichen es Anwendungsprogrammierern, die Leistung und Ressourcennutzung zu untersuchen, um Optimierungspotenziale aufzudecken. Es gibt eine Vielzahl von Werkzeugen, die verschiedene Programmiermodelle unterstützen und unterschiedliche Ansätze für die Analyse verfolgen. Die Scalasca Trace Tools sind eine Toolchain zur Leistungsanalyse, die Wartezustände identifiziert, die durch Kommunikations- und Synchronisationsungleichgewichte in parallelen Anwendungen verursacht werden. Sie analysieren Trace-Daten mit Hilfe eines Replay-basierten Post-Mortem-Trace-Analysators und sind in der Lage, die Ursachen von Delays zu identifizieren. Diese Delay-Analyse stützt sich auf ein Delay-Kostenmodell, dass die Verteilung der gemessenen Wartezeit vornimmt. In dieser Arbeit wurde ein alternatives Delay-Kostenmodell untersucht, dass sich auf die Kostenfortpflanzung der Wartezeit fokussiert, und wurde mit dem bisherigen Delay-Kostenmodell anhand mehrerer Anwendungen verglichen. Das vorgeschlagene Modell wurde definiert, implementiert und die theoretischen Unterschiede zum derzeitigen Modell wurden untersucht. Die Auswertung ergab, dass das Modell zwar keine neuen Ursachen aufdeckt, aber die Mehrdeutigkeit der Analyseergebnisse durch eine stärkere Differenzierung zwischen verschiedenen Wartezuständen potenziell verringert.

Stichwörter: HPC, MPI, OpenMP, Trace-basierte Performance Analyse, Delay Analyse, Scalasca

Abstract

The ongoing development of modern HPC systems increases the need for optimized applications in order to maximize the gained science per consumed core-hour. Performance analysis tools enable application programmers to investigate performance and resource usage helping to expose optimization potential. A variety of tools is available, supporting different programming models and taking different approaches on their analysis. The Scalasca Trace Tools are a performance analysis toolchain that targets wait states caused by communication and synchronization imbalances in parallel applications. They analyze event traces by using a replay-based post-mortem trace analyzer and are capable of identifying root-causes of delays. This delay analysis relies on a delay-cost model that performs the distribution of measured waiting time. This thesis examined an alternative delay-cost model focusing on waiting time propagation and compared it to the previous delay-cost model using multiple applications. The proposed model was defined, implemented and the theoretical capabilities were explored. The evaluation showed that while the model did not uncover new root-causes, it potentially reduces the ambiguity of the analysis results by providing a stronger differentiation between different wait states.

Keywords: HPC, MPI, OpenMP, Trace-based performance analysis, delay analysis, Scalasca

Contents

List of Figures

List of Tables

1. Introduction

With the first exascale system being on the Top500 list of supercomputers in June 2022 [4], a new era of computation has begun for High Performance Computing (HPC). While exascale level hardware enables new kinds of scientific computations, like the simulation of neutrinos in a higher dimensional space [38], it adds an additional layer of complexity to HPC. Even before the existence of exascale systems, HPC platforms grew more diverse through the usage of different architectures. By using both distributed and shared memory, applications can be parallelized more efficiently, while adding a complexity layer through the necessary communication and synchronization. The Message Passing Interface (MPI) [25] represents the standard for distributed memory parallelism by enabling inter-process communication and synchronization. Open Multi-Processing (OpenMP) [12] on the other hand offers the usage of shared memory by making use of multi-threading. It is also possible to use both models together in a hybrid approach to enable even more parallelism for suitable applications. In the last years, supercomputers used to integrate more and more accelerator devices into the systems to provide the necessary computing power for specialized task. These devices are accessed by programming models like the Compute Unified Device Architecture (CUDA) [23] or the Open Computing Language (OpenCL) [28]. While CUDA targets more towards general purpose GPUs, OpenCL enables the use of FPGAs in the context of HPC [28]. This heterogeneity in platform architectures makes it difficult to engineer optimized applications for those systems. As the used hardware and software depends on the manufactures and use-cases of HPC clusters and a variety of parallel programming models is available, it is a challenge to provide applications that perform well on most systems. The high degree of parallelism enables the computation of many different scientific problems, but also introduces communication and synchronizations calls that scale with the number of cores and nodes available in a system. This results in complex communication and synchronization patterns, that can impact the overall performance by delaying processes and cause wait states that results in unproductive occupation of hardware resources. This kind of performance degrading behavior has to be targeted for optimization to enable higher performance of a HPC application, as a priority target is to achieve more science per used core-hour.

In order to optimize HPC applications, performance analysis tools are used to measure, analyze and present performance data in text form or through visual representations. They enable the detection of performance impacting factors like e.g., communication imbalances, wait states, or inefficient resource usage. Measurement data is commonly either collected through sampling or instrumentation. Sampling uses recurring triggers like a timer, to gather measurement data at a certain point of execution and is used in tools like HPCToolkit [5]. Which metrics can be measured depends on the used tools and libraries. PAPI [26] is a library that provides access to hardware performance counters that enable to record performance metrics like e.g., energy consumption. Instrumentation uses additional tools like Extrae [27] or Score-P [7] to inject special triggers into an applications source code to enable measurements tied to the execution of events. In contrast to sampling this ensures that all occurrences of an event type are captured and recordable. The collected data then needs to be processed to provide insights about the performance behavior of an application. Profiling uses the recorded data to provide a statistical overview about a programs performance by summarizing the measurement data for certain metrics. Tools that support profiling are e.g., IntelVTune [31], PerfExpert [11], TAU [33], HPCToolkit, and Scalasca [17]. Another way for data processing is trace-based analysis. Tracing accepts the overhead of capturing complete events to provide a different perspective on the application's behavior by recording the events occurring during runtime and writing them into a trace file. Tracing retains the order of events provided by timestamps and allows to examine the program behavior by reconstructing the execution based on the recorded events. Well-known tools using tracing are e.g., VAMPIR [29], PARAVER [30], TAU, HPCToolkit and Scalasca. These tools can be classified as on-line or post-mortem depending on when the data-processing is performed. On-line tools provide live insights during an application's runtime. Periscope [18] is an on-line performance analysis tool, but has already been discontinued for several years at the point of writing. Post-mortem analysis is performed after the application terminated and is the base of most trace-based tools, as they depend on the full event-trace before they can perform their analysis. Other noteworthy tools are e.g., Gauge [13], which is used for the analysis of I/O performance of parallel applications, and Patha [37] which targets big-data applications on HPC systems and provides execution time and data dependency analysis, that are not relevant for this thesis.

The Scalasca analysis toolset provides tools for performance analysis and diagnosis in HPC applications [17]. It is able to identify communication and synchronization imbalances and detects wait states as well as delays that impact performance and is also able to calculate the critical path of an application. Scalasca aims for high scalability by using a parallel replay of recorded trace-files. It currently supports serial, MPI, OpenMP and hybrid executions, and the support for additional locations like accelerators is planned for the future. Scalasca's analyzer features a delay detection algorithm that assigns waiting-time to wait states to determine candidates for root-causes. These can then be targeted by a user to optimize the programs behavior. In order to perform this root-cause analysis, a delay-cost model is used that was first introduced by Boehme in [8]. The model takes direct and propagated waiting time and distributes it onto the identified wait states to calculate the root-causes. This model was later extended by Hermanns [20] to also be applicable to one-sided MPI communication by providing a cost model that covers this type of wait states.

Scalasca's delay analysis was successfully used in the past to optimize various HPC applications [19]. As delays can also cause wait states indirectly through propagation effects, waiting time is transported backwards in the timeline and assigned to the causes. The current delay-cost model uses a proportional assignment of the waiting time that depends on the length of process-local wait states and computations. With the currently used model, it occurs that wait states are not always fully covered by assigned waiting time due to the proportional distribution that is applied if excess computation is measured in the local profiles. This may lead to an under representation of the propagation effects and can potentially provide misleading results by highlighting intermediary operations that are only considered to be delays, because they were caused by other wait states through propagation. In order to examine whether a focus on propagated waiting time provides a more unambiguous view upon an applications performance, this thesis proposes and evaluates an alternative delay-cost model. The proposed model uses a different approach to distribute the waiting time and prefers wait states during the distribution steps. Moving the cost from intermediary communication or synchronization wait states more towards the end of a delay chain, could result in a more precise determination of the root-causes and therefore the possible impact on optimization potential. It will be evaluated if this model provides new insights in contrast to the current model and if it could be an adequate replacement for the current model, a meaningful extension or if it provides no additional information about the root-causes or even fails to detect previously detected root causes.

This thesis will be structured as follows: Section 2 covers the required background information about performance analysis and the basic functionality and workflow of Scalasca. Section 3 will start by examining the potential weaknesses of the current model and presents the idea of the alternative model. This section also covers the formal definition of the proposed model, the theoretically expected changes during calculation and the implementation into Scalasca. The previously made assumptions are then tested and evaluated by using multiple benchmarks on the CLAIX18 HPC cluster of the RWTH Aachen University in comparison to the current model in Section 4. This thesis closes with a conclusion about the tested model and if it should be integrated into the Scalasca toolset in the future in Section 5.

2. Background

2.1. Parallel paradigms

To be able to understand the wait-state patterns that can be detected during wait state and delay analysis, it is necessary to be familiar with the standard concepts of parallel programming that can cause these patterns.

2.1.1. Message Passing Interface (MPI)

The Message Passing Interface (MPI) [25], is a message passing standard used for inter-process communication in HPC and frequently used for inter-node parallelism in large-scale HPC systems. In the context of MPI, processes are ordered into groups in which they are addressed by a unique identifier called a rank. These groups can be part of one or more communicator objects which provide a distinct communication context for the group members. On initialization, MPI creates one default communicator which covers the group of all processes that were started by a single job. This communicator can then be used to derive communicator objects for subgroups. An MPI operation consists of four stages, *Initialization*, *Starting*, *Completion* and *Freeing*. A blocking operation combines all four stages in a single procedure call, while a non-blocking operation separates the first two stages from the rest. During blocking procedures, a process may have to wait for a specific remote action, such as a receive operation waiting for the corresponding send operation to start. If these actions are not started simultaneously on the involved processes, waiting time can occur. MPI supports three communication paradigms: (1) point-to-point communication, (2) collective communication, and (3) one-sided communication. The latter will be mentioned briefly but is not relevant for this thesis, as the integration of their support in Scalasca is not yet complete.

Point-to-point communication

Point-to-point operations involve a pair of processes. These pairs consist of one sender and one receiver and participate together in a message exchange. Depending whether this exchange is performed in synchronous or asynchronous manor, waiting time can occur if one process arrives earlier at the communication point than the other.

Collective communication

A collective operation is performed over a communicator object and involves all processes in the associated group. The barrier is a synchronization construct that ensures that all processes of the targeted group have reached a common point in the execution before continuing. MPI offers different types of data exchange patterns that fall into this category. These patterns can be classified according to the number of sender and receiver processes: (1) 1×N, (2) N×1, and (3) N×N. In an 1×N operation, one process acts as the sender, while all other processes take the role of the receiver. An example for such an operation is the broadcast. It involves one process that sends its data to all other processes of the used communicator. A similar operation is the scatter, where one process divides its data among all processes. An N×1 operation is defined analogously, but all process send a message to the same process. The N×1 operations can be considered as the opposing operations to the N×1 patterns. Examples are the reduce, which aggregates all data onto a process, and the gather, which collects the data from all processes on a single process. An N×N operation involves a complete message exchange between all participating processes. The N×N patterns are mostly variants of the already mentioned patterns, but involve all processes as sender and receiver. This includes operations like e.g., all-reduce, all-gather and all-scatter.

One-sided communication

A one-sided operation does not require an active communication partner and allows operations like e.g., pushing a message to another process. The origin rank defines all parameters of the communication and uses a remote memory buffer provided by the target rank.

2.1.2. OpenMP

Open Multi-Processing (OpenMP) [12] targets parallelism on node-level by making use of threading and shared-memory. It is an API that allows multi-threading of applications by introducing compiler directives into the applications source code to enable code generation for parallel code. It features parallel structures and communication over shared-memory. Therefore a process is divided into multiple threads that share the resources allocated to the parent process. On large-scale HPC clusters it is often used in combination with MPI for a hybrid model execution. In this case MPI spawns multiple processes that in turn spawn multiple threads themselves, allowing a more fine granular distribution of the workload.

2.2. Performance analysis

The process of performance analysis can be separated into different phases: (1) data collection, (2) processing, and (3) analysis. Each phase can be realized via different techniques that differ in their capabilities and approaches.

2.2.1. Data collection

Before performance analysis can be conducted, measurement data has to be collected. Depending on the amount of detail that is required and the available resources, different approaches can be used. Sampling uses recurring triggers, e.g., probing every X milliseconds, to measure performance metrics in the application, which provides a statistical overview of the events [5]. More frequent sampling provides more data points while potentially increasing the runtime or resource usage of the analysis. In contrast to sampling, instrumentation uses triggers on the events itself instead of time-based samples [5] and adds the possibility to record more detailed traces. While the systematical error caused by sampling is influenced via the sampling rate, tracing introduces overhead on each instrumented call. When using instrumentation it is necessary to instrument the analyzed application beforehand using specialized tools, shortly known as instrumentors. Examples are the integrated instrumentor in the analysis toolkit TAU [33] and the stand-alone instrumentor Score-P [7]. The latter is used to generate traces for tools like Scalasca and VAMPIR.

Score-P

Biersdorf et al. [7] developed Score-P as an instrumentor that that provides measurement data compatible with a variety of tools, therefore reducing the number of measurement runs needed when working with multiple tools. Score-P was developed to provide data compatible with Periscope [18], VAMPIR [29], TAU [33] and Scalasca [17]. The instrumentor can be used for C/C++ and Fortran code and provides performance metrics like execution time, communication metrics and hardware performance counters [7]. It supports serial execution and parallel execution using MPI, OpenMP or a hybrid approach. The kind of instrumentation performed depends on the used paradigms, but gets automatically detected when the instrumentation process is invoked by prefixing the compiler and linker calls with the Score-P instrumentor command. Instrumentation is possible on compiler level, through special libraries or manually using predefined macros. To allow the instrumentation of OpenMP code an integration of the Opari2 [7] library is included. Score-P supports profiling and tracing and the output format of the data depends on the targeted analysis tool. When used with Periscope, no data is explicitly stored, as it is provided through a runtime interface which enables on-line analysis. For post-mortem analysis, the file format depends on whether profiling or tracing mode is enabled. The profiling mode will produce a profile in the CUBE4 format [32] that can be examined with its report explorer. In tracing mode, Score-P will write the recorded event data into an OTF2 trace file which is compatible with VAMPIR and Scalasca.

2.2.2. Processing of measurement data

Once data has been collected, it needs to be processed to enable analysis. The data processing for performance analysis can be generally divided into profiling and tracing. Profiling provides summarized information about an analyzed application and is able to include various performance metrics. Libraries like PAPI [26] provide access to hardware performance counters to record those metrics during application runtime. In contrast to that, during tracing each individual measurement record is saved to a so-called trace file that eventually provides a detailed chronological record of the application behavior. As trace files are used to record temporal information about occurred events, it is able to provide a more in-depth analysis of the applications performance by being able to recreate the programs event history [14].

OTF 2

How the event data is represented in a trace file and later accessed during analysis depends on the used trace format. The Open Trace Format 2 [14] was developed by Eschweiler et al. to join the existing Open Trace Format [22] used by VAMPIR, with the EPILOG [35] trace format used by Scalasca, to provide a trace format that is usable by both tools. This allows a single measurement run to be used for multiple analysis tools. Highly parallel applications use many processes and threads, that all trigger events that will be recorded in the trace file, resulting in large traces. To achieve scalability, the size of the trace file has to be manageable to not heavily impact the analysis performance. Therefore the location local trace data from locations like processes or threads is stored separately for each location. To avoid storing redundant information that is identical for all locations, the file structure aims to provide such information on a global level, resulting in less memory usage of the local trace files. OTF2 employs an anchor file that is used to manage the whole trace with stored meta data. The aforementioned information that is identical among locations is stored in the global definition file. Definitions that only apply to a single location are stored in the local definition file, which also contains mapping data between local and global definitions. The trace data itself is stored location-locally in the local trace file and contains all events recorded for that location.

2.2.3. Analyzing collected data

Once performance data is collected, it can be analyzed to provide insights into the measured application's performance. The analysis can be classified by the time it is performed. On-line analysis provides results during the application's runtime and uses data that has already been recorded and processed. This is employed in tools like Periscope [18]. The second approach is called post-mortem analysis and is performed after the measured application has completed its execution. This assures that all data has been collected and enables more complex analysis types that require certain dependencies in the measured data. An example for this is the detailed analysis of event traces for the effects of imbalances that can be propagated through the execution, influencing operations at a later points in time. Examples for performance analysis tools that employ post-mortem analysis are VAMPIR [29] and Scalasca [17].

2.3. The Scalasca Trace Tools

The Scalasca Trace Tools form a scalable parallel performance analysis toolset for wait state and delay analysis of parallel applications written in C++ [17]. It is able to analyze C/C++ and Fortran applications supporting the MPI, OpenMP or hybrid MPI+OpenMP paradigms besides serial execution. Scalasca performs trace-profiling—a trace-based post-mortem analysis aggregating analysis results in a performance profile—and supports various analysis targets, such as wait states, delays or the critical path. Scalasca identifies wait states and writes them into a CUBE profile that can be examined using the CubeGUI [24]. This thesis will focus on the delay detection of Scalasca's trace analyzer after a brief introduction of Scalasca's functionalities. Scalasca employs a replay-based post-mortem trace analyzer which processes OTF2 trace files recorded from a previously instrumented application run. As a post-mortem analyzer, the analysis is performed after the runtime of the examined application, which allows the use of event trace records to be used for in-depth analysis. The replay-based approach employs a parallel analysis over the entire trace file replaying the synchronization and communication operations that occurred during the application runtime. These operations are re-executed using similar constructs to allow the wait-state analysis on the information stored in the trace file.

2.3.1. Definitions

Before explaining the parallel analyzer itself, certain definitions need to be made. In the scope of Scalasca, the analyzed target patterns are defined and categorized.

Wait states

In Scalasca, wait states are defined as the intervals in which a process experiences waiting time [8]. The impact of a wait state is described by a numerical value according to the waiting time. Furthermore, they can be classified according to their cause by other processes or their position in a series of influencing operations. The impact of a wait state is either direct or indirect. A direct wait state is directly caused from excess computation on the direct communication partner for the waiting communication or synchronization operation. A wait state is called indirect if it was caused by another wait state and is therefore caused by propagation. This leads to the classification of wait states according to their position in the execution. The last process in a series of influencing operations is defined as a terminating wait state as it caused no additional wait states. All other wait states that caused a wait state are defined as propagating wait states, causing further waiting time on another processes.

Delay

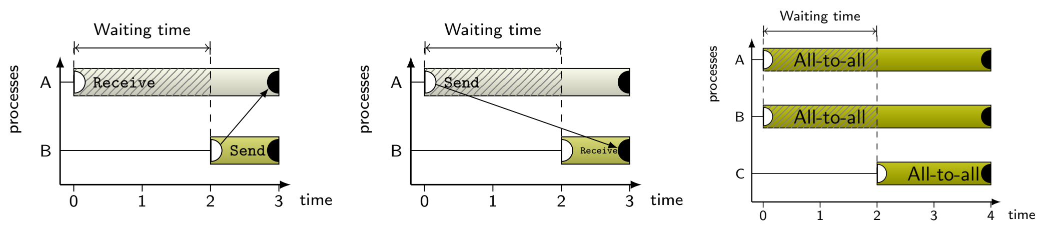
In Scalasca, an activity which causes a late arrival of the process at a point of communication or synchronization, is called a delay [8]. As a delay can cause indirect wait states, all operations that are influenced in some way, are from then on considered to be on its so called delay-chain. To determine which delay is the root cause of a delay-chain, Scalasca uses delay costs. The delay costs are used to assign the impact on the caused wait states among the delays. It is divided into short-term costs, which describe the proportion of caused direct wait states, and long-term costs, which describe the proportion of caused indirect wait states. A detailed explanation of the current delay-cost model follows later on.

Syncpoints and Synchronization Intervals

In order to identify delays, wait states and waiting time, Scalasca uses timestamps to calculate whether an operation acts as delay, and therefore causes waiting time. Operations where one process waits for another are considered as syncpoints [8]. During the execution of highly parallel applications, many communication and synchronization operations are executed across different sets of processes. This includes both MPI point-to-point and collective operations as well as OpenMP synchronizations like barriers. Each operation can introduce new waiting time for other processes or threads that are involved in future synchronization with the affected processes. As the event happening at a syncpoint involves more than one process, the late arrival of one or more processes can cause waiting time for other involved processes or threads, creating complex dependencies between different syncpoints. To quantize the intervals where waiting time is caused, that can possibly interfere with future computation, Scalasca operates on synchronization intervals. These intervals are defined to cover all computation and communication that occurs between two directly succeeding syncpoints that involve the same group of processes [8].

Wait-state patterns

In order to classify the occurred waiting time, several basic wait-state patterns are defined. These patterns represent general patterns of synchronization that can occur during different communication operations. As Scalasca supports both MPI and OpenMP, different patterns for both paradigms are defined [17]. Figure 2.1 shows timeline diagrams of some common wait-state patterns. A timeline diagram consists of multiple processes along the y-axis with their computation over time-units along the x-axis. A white semicircle denotes an enter event on that process, while a black semicircle denotes an exit event. These represent events recorded in



(a) Late-Sender (b) Late-Receiver (c) Wait-at-NxN

Figure 2.1.: Examples for different types of point-to-point and collective wait states supported by Scalasca. Inspired by [17].

a trace, while the arrows denote the direction of communication that was recorded. For point-to-point communication in MPI, the supported wait-state patterns are *Late-Sender* Figure 2.1a and *Late-Receiver* Figure 2.1b. In case of the *Late-Sender* pattern, process **A** enters its MPI-Receive operation on timestep 0 before the corresponding message is sent. The corresponding send operation from Process **B** is sens on timestep 2. Therefore **A** has to wait two time units for the message, experiencing two units of waiting time. Because the waiting time occurred due to the sender of the message arriving late, this type of wait-state pattern is called *Late-Sender*. The *Late-Receiver* depicted in Figure 2.1b is defined accordingly, but with switched operations. For collective communication, *Wait-at-Barrier*, *Early-Reduce*, *Late-Broadcast* and *Wait-at-N×N* are defined. The *Wait-at-N×N* pattern depicted in Figure 2.1c is an example for a collective wait state and could be caused by operations like a MPI-All-to-all, where every process sends data to every other process. For collective communication every process participating in the communication has to perform the operation before it is complete. Therefore the process that arrives last delays the completion. Depending on the specific operation executed, it either impacts the root rank in 1×N or N×1 communication or all participating ranks in N×N operations. In the latter case, the overall waiting time associated with the pattern is the sum of waiting times occurring on all rank during this pattern. In the example of Figure 2.1c this would result in **A** and **B** contributing two units of waiting time each, resulting in a total of four time units being attributed to process **C** that arrived last at the operation. Note, that patterns like *Wait-at-N×N* can be caused by different collective operations like e.g., an MPI-All-reduce or an MPI-All-gather. The pattern for the OpenMP barrier is similar to the pattern of the MPI barrier where processes are substituted by threads accordingly. The second supported OpenMP pattern are the thread-idleness costs which is classified as a collective communication pattern.

2.3.2. The parallel trace analyzer

The “scalable optimization utility” or short SCOUT, is the parallel trace analyzer that is part of Scalasca [39]. In its current state it is able to perform both the wait state and delay detection and is also capable of calculating the critical path. As the core of this thesis is related to the delay detection, the critical path detection is not covered in detail in this thesis but was also implemented by Boehme in [9] alongside Scalasca's current delay detection. The critical path of an application is the longest path of execution without wait states [8]. Therefore optimizing the critical path results in reducing the length of that path and therefore reducing the overall runtime of the application. Scalasca's critical path detection is performed alongside with the delay detection and uses the previously identified syncpoints to backtrack the critical path through the trace file.

PEARL

Besides a scalable trace format, it also requires a scalable interface for accessing the trace data to be able to analyze large scale applications. The Event Analysis and Recognition Library (EARL) is a C++ library that provides a serial interface to a single trace file [34]. The PEARL library is a parallelized replacement by Geimer et al. [15] of EARL. PEARL was developed to add scalability to the interface by working on multiple local trace files instead of a single file. Divided in global definitions and local traces, PEARL uses a similar structure to OTF2, which makes them a suitable combination for the use in Scalasca.

The global definitions are accessed and processed by each process. Each local trace is processed on a separate process, providing a similar execution structure to the actual runtime of the analyzed application. PEARL also provides a certain degree of timestamp correction for incoherences in event timestamps caused by unsynchronized clocks across nodes, using linear interpolation when reading a local trace file. For each executed trace, PEARL provides access to individual events and their respective attributes. PEARL provides a 1:1 mapping of its classes to OTF2 records. That is, for each supported OTF2 record it defines its own C++ class to represent it. These events can be coupled with callback functions that will be executed when the respective event is processed during the analysis. These callback functions allow to perform specific tasks for different events and are used to enable the wait state and delay detection.

Parallel replay

When loading a trace file, each process of the analyzer is assigned to the trace file that was recorded on a mapped process during the original applications runtime. To be able to detect wait states and calculate delay-costs, data exchange between the processes is required. SCOUT gathers the information it needs for the analysis by replaying the communication operations using similar constructs. At any point where communication is recorded during application measurement, the communication event is leveraged to exchange data during the analysis instead of the original message sent by the application. This allows SCOUT to transfer any data and information that is needed during the analysis between the processes. The most basic usage of these transfers are the exchanges of enter and exit timestamps of opera-

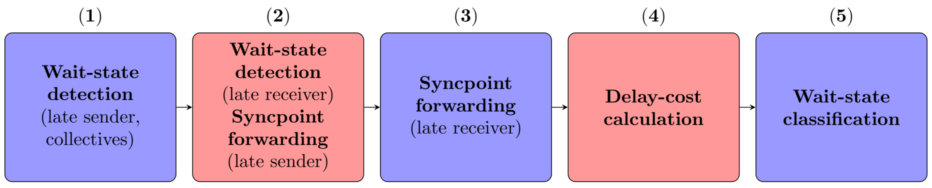

(1)
Wait-state detection (late sender, collectives)
(2)
Wait-state detection (late receiver) Syncpoint forwarding (late sender)
(3)
Syncpoint forwarding (late receiver)
(4)
Delay-cost calculation
(5)
Wait-state classification

Figure 2.2.: The five replay steps for wait state and delay detection, including multiple forward (blue boxes) and backward (red boxes) replays. Modified from [8].

tions. This allows to determine whether a process had to wait in the execution and which type of wait state occurred. If the enter timestamp of a send operation lies behind the enter timestamp of the corresponding receive operation, a *Late-Sender* wait state occurred and the difference between both timestamps can be defined as the time the receiving process spent waiting. In order to detect all of the mentioned wait states and for the delay detection later on, SCOUT has to perform both forward and backward replays of the trace file. In a backward replay the trace file is analyzed from the end to the beginning while the roles of sender and receiver are reversed for each syncpoint.

Figure Figure 2.2 shows those five steps. Blue boxes denote a forward replay step, while red boxes denote backward replays. For each wait state it is necessary to first detect the wait state itself as described before and then annotate all ranks that participated in the operation with the ranks of the other participants. These two steps have to be performed in opposite directions to prepare all processes to compute or receive the necessary information. That is why the wait state detection for *Late-Sender* and the collective wait states are performed in the first forward replay in phase 1, while the corresponding syncpoint forwarding, which performs the preparation, is performed in the next phase where the *Late-Sender* wait states are processed during the backward replay in phase 2. Some wait-state patterns like the *Late-Receiver* are only detectable in a backward replay and are therefore performed in replay phase 2. In the case of a *Late-Receiver* wait state, the rank that caused the waiting time performs the send operation. As data can only be exchanged in the direction of the communication and the data of the victim has to be transferred towards the cause, it can only be performed in a backward phase where the send directions are reversed and allow that transfer. The third replay phase is then again a forward replay for the corresponding syncpoint forwarding. The last two phases of the analysis consist of a second backward and a third forward replay. These steps are needed for the actual delay analysis as the caused waiting time is only measurable after the delay occurred. To assign this costs to the causing delay, the information about the wait state has to flow backward in time, making a backward replay most suitable for this task. After the completed analysis and wait-state classification, SCOUT writes a report in CUBE format that can then be examined in the report explorer provided by the CubeGUI [32].

2.3.3. The current delay-cost model

In order to find the root-causes of wait states, which are delays, the caused waiting time has to be distributed among the processes and computational routines that caused that waiting time. Scalasca's delay analyzer uses a delay-cost model to determine how the waiting time is distributed. The current model was introduced by Boehme in [8]. The formulas used in this section use the notation that was later introduced in the algorithmic description of the model in [9]. All calculations are executed in the fourth replay phase of the analyzer during a backward replay. Due to the backward direction, all sender and receiver relations are reversed while the trace is processed from the latest timestep to the earliest. Therefore the calculation starts at the latest terminal wait state and moves towards the root-cause of a callpath. The calculations are performed on each local trace resulting in local results for each syncpoint that are then exchanged through the corresponding communication operations. The delay-cost calculation and distribution is described in the following as published in [9]. At this point it will not be differentiated between MPI or OpenMP syncpoints as the used formulas are equivalent and only differ in how they are technically implemented inside the source code.

The model

For all syncpoints S the corresponding synchronization interval I_S is determined using the preceding syncpoint with the same ranks. A so called mini-profile p^{local} is calculated for each callpath c in the synchronization interval. It includes the runtime of each computation that was performed. The formal definition of the profile computation is provided in Equation (2.1).

$$p^{\text{local}}[c] = \text{runtime}^{\text{local}}(c) - \text{wait}^{\text{local}}(c), \forall c \in I_S \tag{2.1}$$

The calculation of the delay costs is performed on the process that caused a wait state. Therefore each rank that participated at a certain syncpoint sends its previously calculated mini-profile to the causing process alongside with short and long-term delay-costs. Short-term costs correspond to direct waiting time and exist for every interval that experienced some waiting time. They are denoted as γ_{short}. The long-term delay costs γ_{long} represent propagated waiting time caused by indirect wait states. The causing rank receives the profile and the delay costs and stores them in the form of X^{rem}, where *rem* corresponds to the remote rank that sent the data. E.g., $\gamma^{A}_{\text{short}}$ corresponds to the short-term costs received from rank **A**. Waiting time can be either distributed onto computational routines or other wait states. Before the costs can be calculated, it has to be determined how much excess computation occurred for each routine. Therefore a difference profile Δ_p is calculated as defined in Equation (2.2), using the previously received mini-profile from the remote ranks and the local profile of the current rank.

$$\Delta_p[c] = \max(p^{\text{local}}[c] - p^{\text{remote}}[c], 0), \forall c \in I_S \tag{2.2}$$

A routine only appears in the difference profile, if the runtime on the causing rank is higher than on the delayed rank. It is assumed that if the runtime is higher on a delayed rank, the corresponding calculation on the delaying rank could not have caused the waiting time and is therefore set to zero in the difference profile. The current rank is now able to calculate and distribute the delay costs. The cost model by Boehme uses a proportional assignment of the delay costs represented by a scaling factor s, whose calculation is defined by Equation (2.3). The sum operations on the profiles and wait states add up the entries for all callpaths and synchronzation intervals.

$$s = \frac{1}{Sum(\Delta_p) + Sum(\text{wait}(I_S))} \tag{2.3}$$

This scaling factor distributes the delay costs onto computation and waiting time proportionally to the amount of total computation and waiting time on the current rank. The delay costs are divided into local short-term and local long-term costs and the remaining delay costs are propagated further as propagated delay costs. The local short-term costs are calculated according to Equation (2.4). The received remote short-term costs are scaled using the scaling factor s and assigned to each computational routine in the difference profile according to their contribution.

$$\text{Del.}_{\text{short}}[c] = s \cdot \gamma^{\text{rem}}_{\text{short}} \cdot \Delta_p[c] \tag{2.4}$$

The local long-term costs are calculated analogously(Equation (2.5)), but instead distributes the costs that were propagated from another rank.

$$\text{Del.}_{\text{long}}[c] = s \cdot \gamma^{\text{rem}}_{\text{long}} \cdot \Delta_p[c] \tag{2.5}$$

Waiting time that was not assigned to a computation is assumed to be caused by a wait state on another rank and is therefore propagated further and will be distributed on another rank. Equation (2.6) covers this calculation as it distributes the remaining short and long-term costs to the local wait state. This propagated costs will be send to the next rank in the chain as long-term costs $\gamma^{\text{rem}}_{\text{long}}$.

$$\gamma_{\text{long}}[w] = s \cdot (\gamma^{\text{rem}}_{\text{short}} + \gamma^{\text{rem}}_{\text{long}}) \cdot \text{wait}(w) \tag{2.6}$$

This calculation will repeat on each propagating wait state until the beginning of the trace file is reached. At the end of the analysis, every unit of waiting time is assigned to some computation. As it is shown in [8] that the distributed costs equal the overall waiting time, the proof is omitted in this thesis. The analysis provides the information how much delay costs were assigned to each computation. The results of the delay analysis have to be interpreted by the user as the resulting report only shows how the costs were distributed and does not provide direct information what behavior caused the wait states. The computational routine with the highest delay cost, caused the largest amount of waiting time, either directly or indirectly, and should be examined for optimization potential.

Example

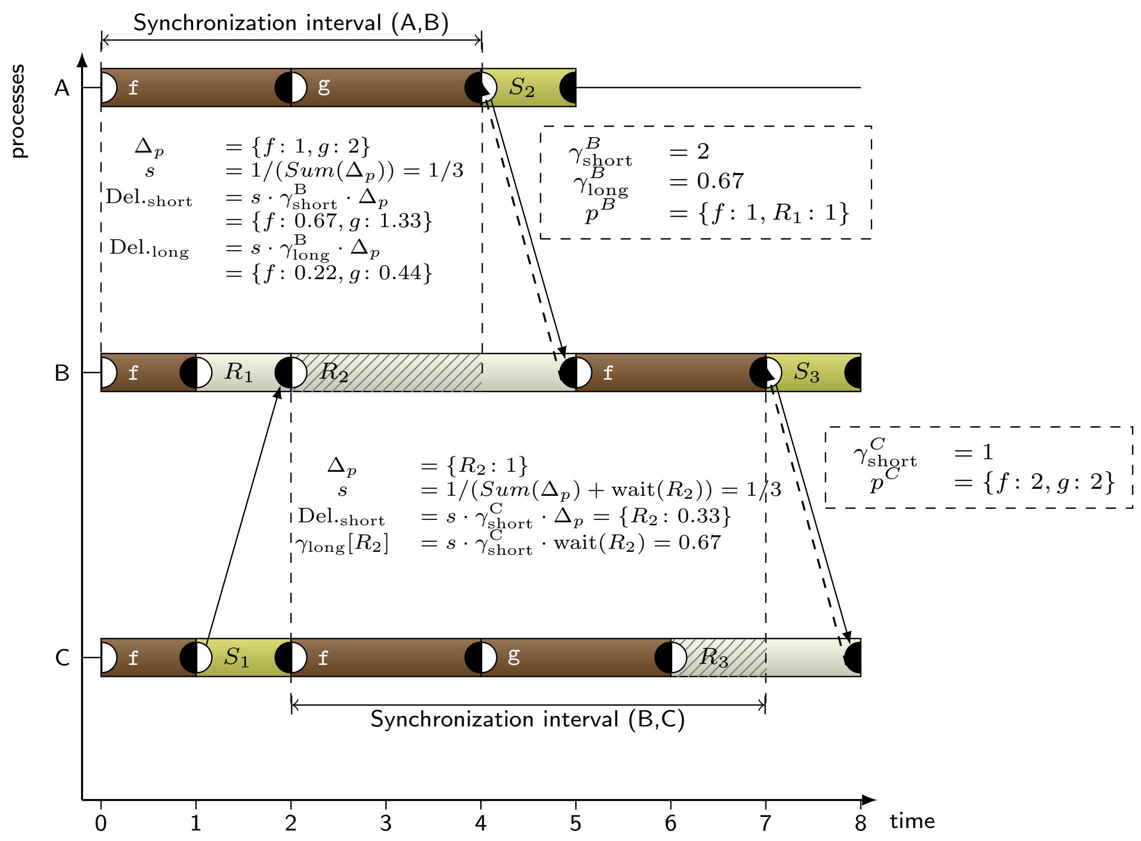

Synchronization interval (A,B)
processes
A
B
C
f
g
S_2
$\Delta_p = \{f\colon 1, g\colon 2\}$
$s = 1/(Sum(\Delta_p)) = 1/3$
$\text{Del.}_{\text{short}} = s \cdot \gamma^{B}_{\text{short}} \cdot \Delta_p = \{f\colon 0.67, g\colon 1.33\}$
$\text{Del.}_{\text{long}} = s \cdot \gamma^{B}_{\text{long}} \cdot \Delta_p = \{f\colon 0.22, g\colon 0.44\}$
$\gamma^{B}_{\text{short}} = 2$
$\gamma^{B}_{\text{long}} = 0.67$
$p^{B} = \{f\colon 1, R_1\colon 1\}$
R_1
R_2
S_3
$\gamma^{C}_{\text{short}} = 1$
$p^{C} = \{f\colon 2, g\colon 2\}$
$\Delta_p = \{R_2\colon 1\}$
$s = 1/(Sum(\Delta_p) + \text{wait}(R_2)) = 1/3$
$\text{Del.}_{\text{short}} = s \cdot \gamma^{C}_{\text{short}} \cdot \Delta_p = \{R_2\colon 0.33\}$
$\gamma_{\text{long}}[R_2] = s \cdot \gamma^{C}_{\text{short}} \cdot \text{wait}(R_2) = 0.67$
S_1
R_3
Synchronization interval (B,C)
0
1
2
3
4
5
6
7
8
time

Figure 2.3.: Timeline diagram with 3 processes. Process **B** and **C** both experience one late sender wait state that was caused by process **A**.

Figure 2.3 shows a timeline diagram with an example calculation on three processes **A**, **B** and **C** using the current delay-cost model. Exchanged data is displayed in dashed boxed, while the calculations are written between the two communication partners. As the delay analysis follows the backward direction, the analysis starts on the last receive operation R_3 on **C**. The local profile of **C** is calculated according to Equation (2.1) and covers two time units of execution time for routine f and two time units for routine g. Together with the short-term costs $\gamma^{\mathbf{C}}_{\text{short}} = 1$, as **C** waited for one time unit, it is send to **B** along the reversed communication direction. The dashed arrows indicate the direction of the data exchange during analysis, whereas the solid arrows indicate the direction in which the communication was recorded. **B** performs the delay-cost calculation for the synchronization interval (**B**,**C**) by first calculating the difference profile from Equation (2.2). **B** and **C** both executed f for two time units, so f will not be included in the difference profile. Therefore the only operation appearing in the profile is R_2. The scaling factor is calculated as $\frac{1}{3}$(see Equation (2.3)) and is then used to calculate the local short-term delay cost(see Equation (2.4)) of $\text{Del.}_{\text{short}} = \{R_2 : 0.33\}$. The remaining short-term cost are assigned to the wait state and therefore $\gamma_{\text{long}}[R_2] = 0.67$(Equation (2.6)) is propagated to process **A** as the long-term cost of **B**, $\gamma_{\text{long}}^{\mathbf{B}}$. The profile p^B, and the costs $\gamma_{\text{short}}^{\mathbf{B}}$ and $\gamma_{\text{long}}^{\mathbf{B}}$ are send to **A**, which calculates the delay costs for the interval (**A**,**B**). As the routine *f* was executed longer on **A** than on **B**, the difference profile includes one remaining unit of execution time for *f*. As *g* was only executed on **A** is entire computation time is included in the profile. **A** did not experience any waiting time, which leads to the scaling factor being calculated just from the total computation time of Δ_p. As no wait state exists on **A**, nothing will be propagated any further. This leads to a final local short-term delay cost of $\gamma_{\text{short}}^{\text{A}} = 2$ and a local long-term delay cost of $\gamma_{\text{short}}^{\mathbf{A}} = 0.66$. While the total costs are preserved, rounding errors in the floating point calculations lead to small deviations as seen for $\gamma_{\text{short}}^{\mathbf{A}}$. At this point the total waiting time of 3 units was distributed among the different delays. Interpreting the results, computation *f* can be identified as the root-cause for the wait states that occurred in this example.

2.4. Summary

Most HPC applications rely on certain programming models to enable efficient parallelism. Therefore, the two standards MPI and OpenMP were introduced to enable inter-process and thread-based parallelism. This chapter provided an introduction to performance analysis and explored the different techniques like, e.g., sampling/instrumentation, profiling/tracing and on-line/post-mortem analysis. It later focused on the Scalasca Trace Tools and provided the applying definitions for wait states, delays, wait-state patterns and points of synchronization. The parallel trace analyzer uses a replay-based approach and provides a scalable analysis of trace files for wait-state patterns and their root-causes. The underlying and currently used delay-cost model was explained to provide a basis for the following proposal of an alternative delay-cost model.

3. An alternative delay-cost model for Scalasca

This chapter covers the alternative delay-cost model that is proposed in this thesis. It will be explained why an alternative model could be beneficial for Scalasca's delay analysis by exposing the weakness of the current delay-cost model. Once it is clear what can be improved, the idea of the alternative is explained, followed by a proof of correctness and a theoretical overview about the calculation capabilities. The last section of the chapter covers the implementation and the tests that were performed to ensure a correct implementation of the model.

3.1. Weaknesses of the current delay-cost model

The first problem arises not from the model itself, but is created when using a timeline diagram to illustrate the classification into direct and indirect wait states. Figure 3.1 shows a slightly modified version of the timeline diagram from Figure 2.3, including small changes of the computations on **B** and **C**. Instead of the delay-cost calculation it is annotated with the delays and occurring wait-state types. According to the definition of delays in Scalasca, delays equal intentional excess computation that causes waiting time on another process. That causes the two units of computation of routine g on process **A** to be considered a delay. This delay causes a direct wait state in the receive operation R_2 on process **B**, where a late sender wait state occurred. As a propagation effect the receive operation becomes a delay itself, delaying the receive operation on process **C**. The receive operation gets further delayed by the execution of routine g. Relying purely on the visual representation suggests that the wait state on **C** is only comprised of a direct wait state. When applying the current delay-cost model on this example, the distribution on **B** results in locally distributed costs of $\text{Del}_{\text{short}} = \{R_2 : 0.6, g\colon 1.2\}$ and propagated costs of $\gamma_{\text{long}} = 1.2$. Therefore, 40% of the waiting time was distributed onto wait states, indicating that the wait state on **C** was caused, at least to some extend, by propagation effects of other wait states. While visual representations of mathematical models are often susceptible to some deviation, the visualization used so far does completely disguise the indirect wait state in situations similar to the chosen example. This creates the necessity of adjusting the timeline-diagrams at some point. In the scope of this thesis, only a proposition for an alternative visualization will be made, when introducing the idea of the alternative model, as it is not affected by the delay-cost model itself.

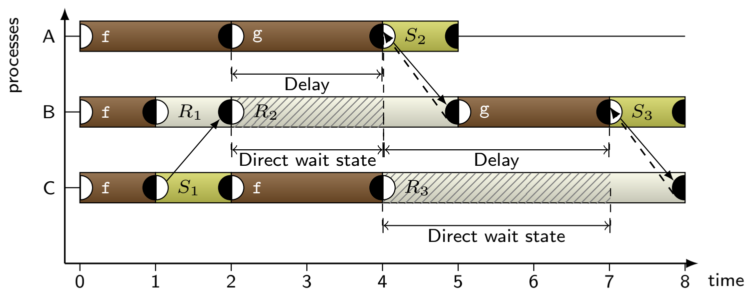


Figure 3.1.: Timeline diagram from Figure 2.3 with annotated delays and wait-state types.

The current model contains two possible weaknesses. The first weakness refers to the assumption of proportionally distributing the waiting time purely based on the amount of local waiting time and excess computation. By always taking the excess amount of computation into account while a local wait state is present, it is implicitly assumed that the excess computation would occur even if the wait state would not be present. Predicting the effects of propagation is not feasible in large-scale applications due to the frequent use of synchronization constructs. Thus it should be assumed that while a local wait state is present, the possibility exists that the excess computation only acts as a delay, because the process experienced waiting time on itself. It is still possible that the delay would occur anyway even if the process did not wait in the same synchronization interval. As it depends solely on the application, it is hard to judge if a delay is caused by propagation, or is always present due to missing load-balancing. The current model made one possible assumption about this behavior and settled with a proportional distribution to achieve a trade-off between those cases. But the possibility exists, that preferring wait states in delay-cost distribution provides a better highlighting for root-causes of propagation chains. This is the core behavior that will be addressed by the alternative model.

The second weakness regards the calculated difference profile. Assume that the application depicted in Figure 3.1 involves operations on matrices. In order to parallelize the calculations, the computation is split into an operation f for the lower triangular matrix and another operation g performing the calculations on the upper triangular matrix. Both functions perform their intended computations in the same synchronization interval. While they are algorithmically executed at the same time, they are recorded as different events, because they call different functions. The difference profile used in the delay model is used to prevent computations to be considered delays, if a routine with the same name was executed on the communication partner. In the mentioned scenario f and g do not fulfill this requirement and are considered as two different tasks. This results in the distribution of local delay-costs to g, whereas it should be treated equally to f from the algorithmic perspective. If the delay-cost model had considered them to be the same computation, the only delay on **B** would have been the computational fraction of R_2. In this case a higher value of $\gamma_{\text{long}} = 2$ would be assigned to the wait state indicating wait state propagation. It has to be mentioned, that the analysis itself is not able to determine whether multiple routines should be considered as equal when they use different function signatures. Therefore, this problem is more related to the implementation of the analyzed application, than Scalasca's delay-cost model, but may be partly mitigated by the lower amount of distributed delay costs on intermediary computations.

3.2. The idea behind the alternative model

The previous section covered the possible weaknesses of the current delay-cost model using Figure 3.1. The proportional assignment of delay-costs only considers the difference profile of the computation and the local waiting time. This may not cover the importance of propagated costs enough. In large-scale parallel applications communication between threads and processes plays an important role for the calculations. Therefore, it is hard to provide optimal balancing of the workload per process and thread which leads to wait states. It only takes a few dependent calculations for a wait state to interfere heavily through propagation effects. Even with the use of delay analysis approaches like the one used by Scalasca, it is not trivial to detect which computations have the largest effect on the experienced waiting time. Highlighting the root-cause of a propagation chain provides a promising approach to eliminate most of the wait states caused by propagation. With this in mind, the alternative model proposed in this thesis tries to shift the focus even more on this root-causes to reduce the costs distributed onto intermediate delays caused by propagation. Distributing the waiting time on fewer routines could provide a better overview about the delays that caused the most wait states.

To achieve a stronger highlighting on delays that propagate a lot, wait states have to be preferred during delay cost assignment, allowing differences in the calculated profiles, as long as the waiting time was not directly caused by the computation.

Figure 3.2 revisits the example from Figure 3.1, but with some changes. It is changed how indirect wait states can be detected visually through the visual representation. Instead of annotating the wait states underneath a direct wait state to be indirect, the effect of the direct wait state on **B** gets shifted to the start of the wait state on **C**. This is visualized by the red arrows and results in marking the first two time units of the wait state on **C** as indirect. The remaining waiting time is considered to be caused by the routines R_3 and g and is therefore marked as direct. This way, the waiting time is first assigned to the wait state on **B**. Only waiting time that exceeded the length of the local wait state is distributed among the computations proportionally to their length in comparison to the sum of the local computing time. If waiting time gets assigned to local routines, therefore depends on the amount of waiting time that a process receives. As long as the wait states are not completely satisfied, all waiting time is propagated, as the model assumes that

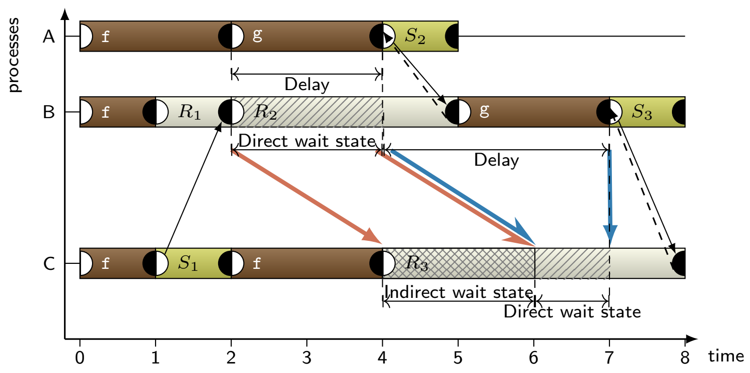


Figure 3.2.: Timeline diagram from Figure 3.1 adopted to the new model of distribution, classifying two thirds of the wait state on **C** as an indirect wait state.

the local wait state was entirely caused by propagation effects. Only if the received waiting time is larger than the local wait state, it can safely be assumed that some fraction was indeed caused by direct effects.

3.3. Deriving the model

The idea from the previous section will now be transformed into mathematical formulas that will cover the exact behavior of the proposed model. It will then be checked for certain properties and will be compared with the current delay-cost model by using examples that cover the possible occurring scenarios.

3.3.1. Formal definition of the alternative model

The definition starts with the introduction of two abbreviations covering the sums of the difference profile and the total waiting time of a callpath. The sum of execution times of all callpaths contained in the difference profile Δ_p is written as computation time comp_t, as defined in Equation (3.1).

$$\text{comp}_t = \sum_{c \in I_S} \Delta_p[c] \tag{3.1}$$

Analogously, the total waiting time is written as wait_t to provide better readability of the following formulas, as defined in Equation (3.2).

$$\text{wait}_t = \sum_{w \in I_S} \text{wait}(w) \tag{3.2}$$

In Equation (3.3) the amount of waiting time that is assigned to the wait states of a remote process is defined. If the received short-term waiting time $\gamma^{\text{rem}}_{\text{short}}$ is less or equal to the length of the local wait states wait_t, the complete short-term waiting time is assigned to the wait states. In the case that $\gamma^{\text{rem}}_{\text{short}}$ is larger than wait_t, the amount of assigned time equals the length of the wait states. This ensures that the wait states are preferably covered before computation is considered.

$$\gamma^{\text{rem}}_{\text{wait}} = \min(\text{wait}_t, \gamma^{\text{rem}}_{\text{short}}) \tag{3.3}$$

The amount of waiting time assigned to computation is defined in Equation (3.4). It depends on whether or not the difference between the incoming short-term waiting time $\gamma^{\text{rem}}_{\text{short}}$ and the total length of all local wait states wait_t is a positive value. In this case the incoming waiting time exceeds the local wait states and waiting time has to be assigned to computational routines. Otherwise the waiting time was completely used for the local wait states.

$$\gamma^{\text{rem}}_{\text{comp}} = \max(0, \gamma^{\text{rem}}_{\text{short}} - \text{wait}_t) \tag{3.4}$$

The current model uses a single scaling factor to perform the waiting time distribution. In order to cover the special behavior in the alternative model, two separate scaling factors have to be used to allow correct distribution. s_{comp} is defined as the inverse of comp_t and is used to scale the waiting time assigned to the computation, as defined in Equation (3.5).

$$s_{\text{comp}} = \frac{1}{\text{comp}_t} \tag{3.5}$$

Analogously, s_{wait} is defined to be used to scale the waiting time assigned to the local wait states.

$$s_{\text{wait}} = \frac{1}{\text{wait}_t} \tag{3.6}$$

The formula used to calculate the local short-term delay costs $\text{Del}_{\text{short}}$, Equation (3.7), differs to the current formula in the used scaling factor and receives a different input of waiting time by using $\gamma^{\text{rem}}_{\text{comp}}$ instead of $\gamma^{\text{rem}}_{\text{short}}$. The incoming short-term waiting time $\gamma^{\text{rem}}_{\text{comp}}$ is scaled according to the amount of waiting time that exceeded the local wait states by s_{comp}. $\text{Del}_{\text{short}}$ is greater zero iff comp_t is greater zero, resulting in a distribution onto computation if there was enough waiting time to be distributed on the wait states first.

$$\text{Del}_{\text{short}}[c] = \gamma^{\text{rem}}_{\text{comp}} \cdot s_{\text{comp}} \cdot \Delta_p[c] \tag{3.7}$$

Lemma 1. *For all callpaths c in a synchronization interval the following holds:* $\sum_{c \in I_S} Del_{short}[c] = \gamma^{rem}_{comp}$

Proof.

$$\begin{aligned}
\sum_{c\in I_S} \mathrm{Del}_{\mathrm{short}}[c] &= \sum_{c\in I_S} (\gamma^{\mathrm{rem}}_{\mathrm{comp}} \cdot s_{\mathrm{comp}} \cdot \Delta_p[c]) \\
&= \gamma^{\mathrm{rem}}_{\mathrm{comp}} \cdot s_{\mathrm{comp}} \cdot \sum_{c\in I_S} \Delta_p[c] \\
&\overset{\text{Equation (3.1)}}{=} \gamma^{\mathrm{rem}}_{\mathrm{comp}} \cdot s_{\mathrm{comp}} \cdot \mathrm{comp}_t \\
&\overset{\text{Equation (3.5)}}{=} \gamma^{\mathrm{rem}}_{\mathrm{comp}} \cdot \frac{1}{\mathrm{comp}_t} \cdot \mathrm{comp}_t \\
&= \gamma^{\mathrm{rem}}_{\mathrm{comp}} \qquad \square
\end{aligned}$$

The calculation of the local long-term delay cost requires a different modification (Equation (3.8)). While it uses the same scaling factor as the short-term costs, the incoming value of $\gamma^{\mathrm{rem}}_{\mathrm{long}}$ requires additional scaling. It is necessary to determine the fraction of long-term costs that is assigned to computation and the fraction that will be propagated. This is achieved by dividing the time assigned to computation $\gamma^{\mathrm{rem}}_{\mathrm{comp}}$ by the total incoming short-term waiting time $\gamma^{\mathrm{rem}}_{\mathrm{short}}$.

$$\mathrm{Del}_{\mathrm{long}}[c] = \gamma^{\mathrm{rem}}_{\mathrm{long}} \cdot \frac{\gamma^{\mathrm{rem}}_{\mathrm{comp}}}{\gamma^{\mathrm{rem}}_{\mathrm{short}}} \cdot s_{\mathrm{comp}} \cdot \Delta_p[c] \tag{3.8}$$

Lemma 2. *For all callpaths c in a synchronization interval the following holds:* $\sum_{c\in I_S} Del_{long}[c] = \gamma^{rem}_{long} \cdot \frac{\gamma^{rem}_{comp}}{\gamma^{rem}_{short}}$

Proof.

$$\begin{aligned}
\sum_{c\in I_S} \mathrm{Del}_{\mathrm{long}}[c] &= \sum_{c\in I_S} (\gamma^{\mathrm{rem}}_{\mathrm{long}} \cdot \frac{\gamma^{\mathrm{rem}}_{\mathrm{comp}}}{\gamma^{\mathrm{rem}}_{\mathrm{short}}} \cdot s_{\mathrm{comp}} \cdot \Delta_p[c]) \\
&= \gamma^{\mathrm{rem}}_{\mathrm{long}} \cdot \frac{\gamma^{\mathrm{rem}}_{\mathrm{comp}}}{\gamma^{\mathrm{rem}}_{\mathrm{short}}} \cdot s_{\mathrm{comp}} \cdot \sum_{c\in I_S} \Delta_p[c] \\
&\overset{\text{Equation (3.1)}}{=} \gamma^{\mathrm{rem}}_{\mathrm{long}} \cdot \frac{\gamma^{\mathrm{rem}}_{\mathrm{comp}}}{\gamma^{\mathrm{rem}}_{\mathrm{short}}} \cdot s_{\mathrm{comp}} \cdot \mathrm{comp}_t \\
&\overset{\text{Equation (3.5)}}{=} \gamma^{\mathrm{rem}}_{\mathrm{long}} \cdot \frac{\gamma^{\mathrm{rem}}_{\mathrm{comp}}}{\gamma^{\mathrm{rem}}_{\mathrm{short}}} \cdot \frac{1}{\mathrm{comp}_t} \cdot \mathrm{comp}_t \\
&= \gamma^{\mathrm{rem}}_{\mathrm{long}} \cdot \frac{\gamma^{\mathrm{rem}}_{\mathrm{comp}}}{\gamma^{\mathrm{rem}}_{\mathrm{short}}} \qquad \square
\end{aligned}$$

The calculation of the propagated cost $\gamma_{\mathrm{long}}[w]$ is defined in Equation (3.9) and is performed for each wait state that occurred in the currently processed synchronization interval. It consists of the fraction of short-term waiting time that was already hold back to be assigned to the wait states by calculating $\gamma^{\mathrm{rem}}_{\mathrm{wait}}$ and the remaining long-term waiting time that was not already distributed in Equation (3.8). It is later shown that $\gamma^{\mathrm{rem}}_{\mathrm{comp}} + \gamma^{\mathrm{rem}}_{\mathrm{wait}} = \gamma^{\mathrm{rem}}_{\mathrm{short}}$ holds. This implies that by multiplying $\gamma^{\mathrm{rem}}_{\mathrm{long}}$ with $\frac{\gamma^{\mathrm{rem}}_{\mathrm{wait}}}{\gamma^{\mathrm{rem}}_{\mathrm{short}}}$ the remaining amount of long-term waiting time is considered for propagating cost calculation. The summed fractions need to be scaled using s_{wait} to consider if there was enough waiting time to completely satisfy the wait states. The scaled sum is then multiplied with each wait state to finish the calculation of the propagated delay-cost per wait state.

$$\gamma_{\text{long}}[w] = (\gamma^{\text{rem}}_{\text{wait}} + \gamma^{\text{rem}}_{\text{long}} \cdot \frac{\gamma^{\text{rem}}_{\text{wait}}}{\gamma^{\text{rem}}_{\text{short}}}) \cdot s_{\text{wait}} \cdot \text{wait}(w) \tag{3.9}$$

Lemma 3. *For all callpaths c in a synchronization interval holds:* $\sum_{w \in I_S} \gamma_{long}[w] = \gamma^{rem}_{wait} + \gamma^{rem}_{long} \cdot \frac{\gamma^{rem}_{wait}}{\gamma^{rem}_{short}}$

Proof.

$$\begin{aligned}
\sum_{w \in I_S} \gamma_{\text{long}}[w] &= \sum_{w \in I_S} ((\gamma^{\text{rem}}_{\text{wait}} + \gamma^{\text{rem}}_{\text{long}} \cdot \frac{\gamma^{\text{rem}}_{\text{wait}}}{\gamma^{\text{rem}}_{\text{short}}}) \cdot s_{\text{wait}} \cdot \text{wait}(w)) \\
&= (\gamma^{\text{rem}}_{\text{wait}} + \gamma^{\text{rem}}_{\text{long}} \cdot \frac{\gamma^{\text{rem}}_{\text{wait}}}{\gamma^{\text{rem}}_{\text{short}}}) \cdot s_{\text{wait}} \cdot \sum_{w \in I_S} \text{wait}(w) \\
&\overset{\text{Equation (3.2)}}{=} (\gamma^{\text{rem}}_{\text{wait}} + \gamma^{\text{rem}}_{\text{long}} \cdot \frac{\gamma^{\text{rem}}_{\text{wait}}}{\gamma^{\text{rem}}_{\text{short}}}) \cdot s_{\text{wait}} \cdot \text{wait}_t \\
&\overset{\text{Equation (3.6)}}{=} (\gamma^{\text{rem}}_{\text{wait}} + \gamma^{\text{rem}}_{\text{long}} \cdot \frac{\gamma^{\text{rem}}_{\text{wait}}}{\gamma^{\text{rem}}_{\text{short}}}) \cdot \frac{1}{\text{wait}_t} \cdot \text{wait}_t \\
&= \gamma^{\text{rem}}_{\text{wait}} + \gamma^{\text{rem}}_{\text{long}} \cdot \frac{\gamma^{\text{rem}}_{\text{wait}}}{\gamma^{\text{rem}}_{\text{short}}} \qquad \square
\end{aligned}$$

3.3.2. Proving the amount of distributed waiting time

With the model being defined, it remains to show that the formulas defined above result in the correct total amount of waiting time that gets distributed. Waiting time is only considered distributed if it was assigned as local delay costs to computational routine. Propagated waiting time is used to transfer waiting time to another process to consider indirect causes of wait states and is not counted towards the distributed time until it was used in a local delay-cost calculation. Therefore, it has to been shown that despite the possibility that not the entire waiting time is distributed on a certain rank, the total amount is still correct once all calculations are complete.

Lemma 4. *The short-term costs received by a process is fully divided into a local and propagation fraction, because* $\gamma^{rem}_{comp} + \gamma^{rem}_{wait} = \gamma^{rem}_{short}$ *holds.*

First it will be shown that Lemma 4 holds, by resolving the minimum and maximum operations.

Proof.

$$\begin{aligned}
&\gamma^{\text{rem}}_{\text{comp}} + \gamma^{\text{rem}}_{\text{wait}} = \max(0, \gamma^{\text{rem}}_{\text{short}} - \text{wait}_t) + \min(\text{wait}_t, \gamma^{\text{rem}}_{\text{short}})\\
&if(\gamma^{\text{rem}}_{\text{short}} > \text{wait}_t):\\
&\Leftrightarrow \max(0, \gamma^{\text{rem}}_{\text{short}} - \text{wait}_t) = \gamma^{\text{rem}}_{\text{short}} - \text{wait}_t\\
&\Leftrightarrow \min(\text{wait}_t, \gamma^{\text{rem}}_{\text{short}}) = \text{wait}_t\\
&\Rightarrow \gamma^{\text{rem}}_{\text{short}} - \text{wait}_t + \text{wait}_t = \gamma^{\text{rem}}_{\text{short}}\\
\\
&if(\gamma^{\text{rem}}_{\text{short}} \leq \text{wait}_t):\\
&\Leftrightarrow \max(0, \gamma^{\text{rem}}_{\text{short}} - \text{wait}_t) = 0\\
&\Leftrightarrow \min(\text{wait}_t, \gamma^{\text{rem}}_{\text{short}}) = \gamma^{\text{rem}}_{\text{short}}\\
&\Rightarrow 0 + \gamma^{\text{rem}}_{\text{short}} = \gamma^{\text{rem}}_{\text{short}}
\end{aligned}$$

□

Theorem 5. *All costs received in a step are fully distributed, as* $\gamma^{rem}_{short} + \gamma^{rem}_{long} = \sum_{c \in I_S}(Del_{short}[c] + Del_{long}[c]) + \sum_{w \in I_S} \gamma_{long}[w]$ *holds.*

Lemma 1-4 combined lead to the prove of the theorem above, that the sum of propagated and locally distributed waiting time equals the sum of received short and long-term waiting time.

Proof.

$$\begin{aligned}
&= && \sum_{c \in I_S} \text{Del}_{\text{short}}[c] + \sum_{c \in I_S} \text{Del}_{\text{long}}[c] + \sum_{w \in I_S} \gamma_{\text{long}}[w]\\
&\overset{\text{Lemma 1}}{=} && \gamma^{\text{rem}}_{\text{comp}} + \sum_{c \in I_S} \text{Del}_{\text{long}}[c] + \sum_{w \in I_S} \gamma_{\text{long}}[w]\\
&\overset{\text{Lemma 2}}{=} && \gamma^{\text{rem}}_{\text{comp}} + \gamma^{\text{rem}}_{\text{long}} \cdot \frac{\gamma^{\text{rem}}_{\text{comp}}}{\gamma^{\text{rem}}_{\text{short}}} + \sum_{w \in I_S} \gamma_{\text{long}}[w]\\
&\overset{\text{Lemma 3}}{=} && \gamma^{\text{rem}}_{\text{comp}} + \gamma^{\text{rem}}_{\text{long}} \cdot \frac{\gamma^{\text{rem}}_{\text{comp}}}{\gamma^{\text{rem}}_{\text{short}}} + \gamma^{\text{rem}}_{\text{wait}} + \gamma^{\text{rem}}_{\text{long}} \cdot \frac{\gamma^{\text{rem}}_{\text{wait}}}{\gamma^{\text{rem}}_{\text{short}}}\\
&= && \gamma^{\text{rem}}_{\text{comp}} + \gamma^{\text{rem}}_{\text{wait}} + \gamma^{\text{rem}}_{\text{long}} \cdot (\frac{\gamma^{\text{rem}}_{\text{comp}}}{\gamma^{\text{rem}}_{\text{short}}} + \frac{\gamma^{\text{rem}}_{\text{wait}}}{\gamma^{\text{rem}}_{\text{short}}})\\
&= && \gamma^{\text{rem}}_{\text{comp}} + \gamma^{\text{rem}}_{\text{wait}} + \gamma^{\text{rem}}_{\text{long}} \cdot (\frac{\gamma^{\text{rem}}_{\text{comp}} + \gamma^{\text{rem}}_{\text{wait}}}{\gamma^{\text{rem}}_{\text{short}}})\\
&\overset{\text{Lemma 4}}{=} && \gamma^{\text{rem}}_{\text{short}} + \gamma^{\text{rem}}_{\text{long}} \cdot (\frac{\gamma^{\text{rem}}_{\text{short}}}{\gamma^{\text{rem}}_{\text{short}}})\\
&= && \gamma^{\text{rem}}_{\text{short}} + \gamma^{\text{rem}}_{\text{long}}
\end{aligned}$$

□

It was shown that each distribution step preserves the amount of waiting time and no costs are lost or additionally generated. It remains to show that the propagated costs are assigned at some point, resulting in a finished distribution of the delay-costs.

Theorem 6. *All costs are assigned, as* $\gamma^{rem}_{short} + \gamma^{rem}_{long} = \sum_{c \in I_S}(Del_{short}[c] + Del_{long}[c]) + \sum_{w \in I_S} \gamma_{long}[w]$ *holds for* $\sum_{w \in I_S} \gamma_{long_{local}}[w] = 0$.

As there can not be any circulatory dependencies between wait states due to the measured applications runtime being finite and the fact that all events occur in a continuous manor, there exists a synchronization interval that contains a delay, but not a local wait state. So, it has to be proven that this last step distributes all incoming costs and does not declare anything as propagated cost.

Proof. As no local wait state exists, $\text{wait}_t = 0$ holds, and therefore
$\gamma^{\text{rem}}_{\text{wait}} = \min(wait_t, \gamma^{\text{rem}}_{\text{short}}) = \min(0, \gamma^{\text{rem}}_{\text{short}}) = 0$.
It also holds that $\gamma^{\text{rem}}_{\text{short}} = \sum \Delta_p = comp_t$, as all short-term costs are caused by delays. This leads to $\gamma^{\text{rem}}_{\text{comp}} = \max(0, \gamma^{\text{rem}}_{\text{short}} - \text{wait}_t) = \max(0, \text{comp}_t) = \text{comp}_t$. The only relevant scaling factor at this point is $s_{\text{comp}} = \frac{1}{\text{comp}_t}$. This leads to the entire short-term costs being distributed as local delay-costs, as $s_{\text{comp}} \cdot \gamma^{\text{rem}}_{\text{comp}} = 1$.
The long-term waiting time is entirely distributed as local delay-costs, because $\gamma^{\text{rem}}_{\text{comp}} = \gamma^{\text{rem}}_{\text{short}}$ and therefore $\frac{\gamma^{\text{rem}}_{\text{comp}}}{\gamma^{\text{rem}}_{\text{comp}}} = 1$. Also no fraction of the long-term waiting time is propagated, because $\gamma^{\text{rem}}_{\text{wait}} = 0$ and therefore $\frac{\gamma^{\text{rem}}_{\text{wait}}}{\gamma^{\text{rem}}_{\text{comp}}} = 0$.
This results in a complete distribution of the entire received waiting-time on the earliest delaying process. □

Since it is proven that each distribution step contains the total amount of received costs and all propagated costs are distributed at some point, the proposed model performs a complete distribution of the waiting time.

3.4. Comparison with the current model

Before looking into a comparison of real-world applications, it will be shown that a theoretically difference exists between the calculated outcome of both models. Equation (3.4) is critical to the behavior of the model. It provides two possible outcomes. If the length of the local wait states wait_t on a process is larger or equal to the incoming waiting time $\gamma^{\text{rem}}_{\text{short}}$, nothing is attributed towards computation and $\gamma_{\text{comp}} = 0$ holds. In case that the total waiting time is smaller than the incoming waiting time, the amount of waiting time assigned to computation equals the difference between the incoming short-term waiting time and the length of the wait state, $\text{comp}_t - \gamma^{\text{rem}}_{\text{short}}$.

3.4.1. Case 1: $\text{wait}_t \geq \gamma^{\text{rem}}_{\text{short}}$

Figure 3.3 illustrates the calculation for the first case. It depicts the same timeline diagram as Figure 2.3, but uses the alternative model for the calculations. Starting at the latest syncpoint, process **C** sends the direct waiting time from R_3 via $\gamma^{\text{C}}_{\text{short}} = 1$ along with the mini-profile $p^C = \{f\colon 2, g\colon 2\}$ to the delaying process **B**. The calculated difference profile on **B** is unchanged, as no changes were made to its

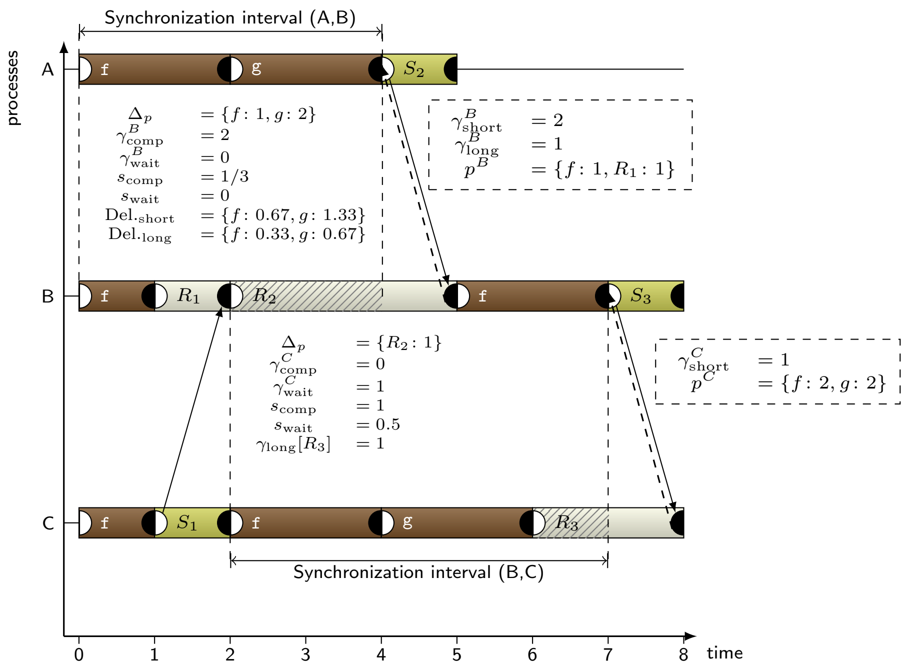

Synchronization interval (A,B)
processes
A
B
C
f
g
S_2
$\Delta_p = \{f: 1, g: 2\}$
$\gamma^B_{\text{comp}} = 2$
$\gamma^B_{\text{wait}} = 0$
$s_{\text{comp}} = 1/3$
$s_{\text{wait}} = 0$
$\text{Del.}_{\text{short}} = \{f: 0.67, g: 1.33\}$
$\text{Del.}_{\text{long}} = \{f: 0.33, g: 0.67\}$
$\gamma^B_{\text{short}} = 2$
$\gamma^B_{\text{long}} = 1$
$p^B = \{f: 1, R_1: 1\}$
R_1
R_2
S_3
$\Delta_p = \{R_2: 1\}$
$\gamma^C_{\text{comp}} = 0$
$\gamma^C_{\text{wait}} = 1$
$s_{\text{comp}} = 1$
$s_{\text{wait}} = 0.5$
$\gamma_{\text{long}}[R_3] = 1$
$\gamma^C_{\text{short}} = 1$
$p^C = \{f: 2, g: 2\}$
S_1
R_3
Synchronization interval (B,C)
0
1
2
3
4
5
6
7
8
time

Figure 3.3.: The short-term waiting time from **C** is less or equal to the length of the wait state on **B**, preventing attribution to the local computation of **B**.

calculation. It then calculates the fractions for computation and wait states. As wait_t is larger than $\gamma^{\text{C}}_{\text{short}}$, Equation (3.4) evaluates to zero, leading to $\gamma^{\text{C}}_{\text{comp}} = 0$. Equation (3.3) is then used to calculate $\gamma^{\text{C}}_{\text{wait}}$ and evaluates to $\gamma^{\text{C}}_{\text{wait}} = 1$. As the waiting time sent by process **A** is less than the length of the wait state on **B**, the model assumes that the whole waiting time was caused by propagation. In the next step, the process calculates the scaling factors. Because s_{comp} is only used for the calculation of local delay cost and $\gamma^{\text{C}}_{\text{comp}} = 0$, it is not required on **B**. The scaling factor for the waiting time s_{wait} is calculated as the inverse of the total waiting time wait_t according to Equation (3.8). R_2 is delayed by two time units, therefore the scaling factor is $s_{\text{wait}} = 0.5$. The last step on **B** involves the calculation of the propagated waiting time as defined in Equation (3.9). With **B** being the first calculating process, no long-term waiting time exists at this point. The propagated waiting time is therefore calculated from $\gamma^{\text{C}}_{\text{wait}} = 1$, the scaling factor $s_{\text{wait}} = 0.5$, and $\text{wait}(R_3) = 2$ which results in $\gamma_{\text{long}}[R_3] = 1$. Therefore, all incoming waiting time is propagated further back to **A**. The communication at the syncpoint from R_2 and S_2 is used to retrieve the waiting time and mini-profile from **B**. **A** first calculates $\gamma^{\text{B}}_{\text{comp}} = 2$ and $\gamma^{\text{B}}_{\text{wait}} = 0$. The scaling factors are calculated to be $s_{\text{comp}} = \frac{1}{3}$ and $s_{\text{wait}} = 0$, because there is no wait state on process **A**. Using Equation (3.7), the local short-term delay cost are calculated to be $\text{Del}_{\text{short}} = \{f: 0.67, g: 1.33\}$ using the previously calculated values. Analogously, Equation (3.8) is used to calculate the local long-term delay cost of $\text{Del}_{\text{long}} = \{f: 0.33, g: 0.67\}$.

For easier comparison of the results for both models, Table 3.1 shows the local and

Process	Current model		Alternative model	
	A	B	A	B
$\text{Del}_{\text{short}}$	$f\colon 0.67, g\colon 1.33$	$R_2 : 0.33$	$f\colon 0.67, g\colon 1.33$	0
Del_{long}	$f\colon 0.22, g\colon 0.44$	0	$f\colon 0.33, g\colon 0.67$	0
γ_{long}	0	$R_2\colon 0.67$	0	$R_2\colon 1$

Table 3.1.: Comparison of the delay-cost calculation for the current and proposed alternative delay-cost model for $\text{wait}_t \geq \gamma^{\text{rem}}_{\text{short}}$.

propagated waiting times on each process. As intended by the alternative model, the propagation on **B** was increased in comparison to the current delay-cost model. This resulted in an increase of the assigned local long-term delay cost on **A** and removed all local delay cost from **B**. Therefore, **B** is no longer considered a cause for wait states as the whole waiting-time was legitimately caused by the delay on **A**.

3.4.2. Case 2: $\text{wait}_t < \gamma^{\text{rem}}_{\text{short}}$

If a process receives waiting time $\gamma^{\text{rem}}_{\text{short}}$ that exceeds the lengths of its own wait states wait_t, the remaining portion will be distributed onto the computations contained in the difference profile of the current synchronization interval. Therefore, two sub-cases are possible. Either the waiting time received from a remote process $\gamma^{\text{rem}}_{\text{short}}$ equals the sum of the length of the local wait states and the local computation or it is less then the sum. It is not possible that the waiting time received from a process can exceed this sum, as the amount of waiting time a process experiences, correlates with the length of the direct and indirect effects of the delaying process. This also holds for collective communication where the assigned delay costs were received from multiple communication partners, as each delayed process is considered separately. The next two cases use the conditions of the example that was used to explain the weaknesses in Section 3.1.

Case 2.1: $\gamma^{\text{rem}}_{\text{short}} = \text{wait}_t + \text{comp}_t$

The first sub-case occurs, if the sum of length of local wait state wait_t and the computation time comp_t is equal to the incoming short-term waiting time. To cover this situation, Figure 3.4 shows the timeline diagram from Figure 3.1, but the operations represented on **B** and **C** now use the same function signature. This causes the computation to not be included in the difference profile as both processes spend the same time in the computation. Process **B** receives $\gamma^{\text{C}}_{\text{short}} = 3$ from **C** at the first syncpoint. It then computes $\gamma^{\text{C}}_{\text{comp}} = 1$ and $\gamma^{\text{C}}_{\text{wait}} = 2$ followed by the scaling factors $s_{\text{comp}} = 1$ and $s_{\text{wait}} = 0.5$. This leads to the distribution of local short-term delay cost of $\text{Del}_{\text{short}} = \{f\colon 1\}$. The remaining waiting time is propagated towards

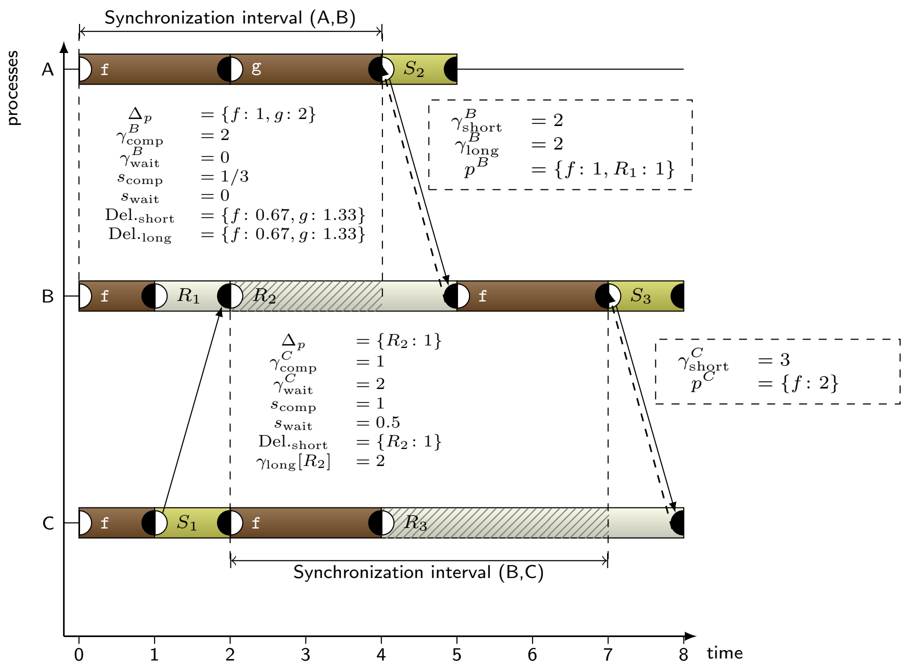

Synchronization interval (A,B)
processes
A
B
C
f
g
S_2
R_1
R_2
S_3
S_1
R_3
$\Delta_p = \{f\colon 1, g\colon 2\}$
$\gamma^B_{\text{comp}} = 2$
$\gamma^B_{\text{wait}} = 0$
$s_{\text{comp}} = 1/3$
$s_{\text{wait}} = 0$
$\text{Del}_{\cdot\text{short}} = \{f\colon 0.67, g\colon 1.33\}$
$\text{Del}_{\cdot\text{long}} = \{f\colon 0.67, g\colon 1.33\}$
$\gamma^B_{\text{short}} = 2$
$\gamma^B_{\text{long}} = 2$
$p^B = \{f\colon 1, R_1\colon 1\}$
$\Delta_p = \{R_2\colon 1\}$
$\gamma^C_{\text{comp}} = 1$
$\gamma^C_{\text{wait}} = 2$
$s_{\text{comp}} = 1$
$s_{\text{wait}} = 0.5$
$\text{Del}_{\cdot\text{short}} = \{R_2\colon 1\}$
$\gamma_{\text{long}}[R_2] = 2$
$\gamma^C_{\text{short}} = 3$
$p^C = \{f\colon 2\}$
Synchronization interval (B,C)
0
1
2
3
4
5
6
7
8
time

Figure 3.4.: The short-term waiting time exceeds the length of the local wait state on **B**, which leads to a propagation equal to the wait state and a distribution of the remaining time unit to the computational routine.

process **A** with $\gamma_{\text{long}}[R_2] = 2$. **A** receives $\gamma^{\text{B}}_{\text{short}} = 2$ and $\gamma^{\text{B}}_{\text{long}} = 2$ as waiting time from **B** through the syncpoint at S_2. Using this values and the difference profile, **A** is able to compute $\text{Del}_{\text{short}} = \{f\colon 0.67, g\colon 1.33\}$ and $\text{Del}_{\text{long}} = \{f\colon 0.67, g\colon 1.33\}$ as the final delay cost distribution.

In order to allow a comparison between both models on this example, Table 3.2 contains the results for both models on this example. The calculations are performed as defined in Section 2.3.3.

Process	Current model		Alternative model	
	A	B	A	B
$\text{Del}_{\text{short}}$	$f\colon 0.67, g\colon 1.33$	$R_2\colon 1$	$f\colon 0.67, g : 1.33$	$R_2\colon 1$
Del_{long}	$f\colon 0.67, g : 1.33$	0	$f\colon 0.67, g\colon 1.33$	0
γ_{long}	0	$R_2\colon 2$	0	$R_2\colon 2$

Table 3.2.: Comparison of the delay-cost calculation for the current and proposed alternative delay-cost model for $\gamma^{\text{rem}}_{\text{short}} = \text{wait}_t + \text{comp}_t$.

Comparing the results of the delay cost calculations shows that in this case both models end up with the same results. This shows that it is theoretically possible to get the same analysis results using any of the two models. Therefore, it could be

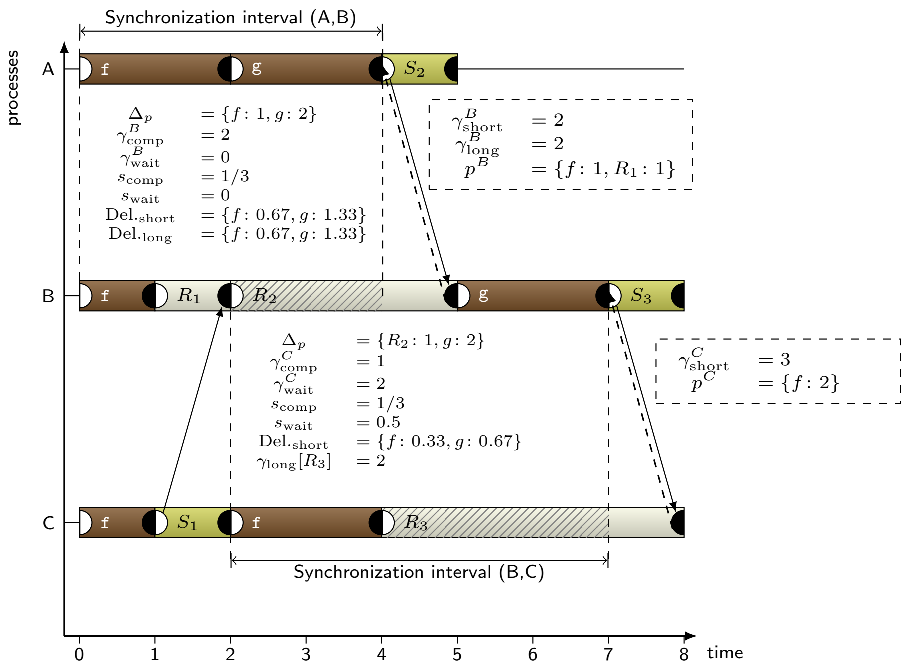

Synchronization interval (A,B)
processes
A
B
C
f
g
S_2
R_1
R_2
S_3
S_1
R_3
$\Delta_p = \{f: 1, g: 2\}$
$\gamma^B_{\text{comp}} = 2$
$\gamma^B_{\text{wait}} = 0$
$s_{\text{comp}} = 1/3$
$s_{\text{wait}} = 0$
$\text{Del.}_{\text{short}} = \{f: 0.67, g: 1.33\}$
$\text{Del.}_{\text{long}} = \{f: 0.67, g: 1.33\}$
$\gamma^B_{\text{short}} = 2$
$\gamma^B_{\text{long}} = 2$
$p^B = \{f: 1, R_1: 1\}$
$\Delta_p = \{R_2: 1, g: 2\}$
$\gamma^C_{\text{comp}} = 1$
$\gamma^C_{\text{wait}} = 2$
$s_{\text{comp}} = 1/3$
$s_{\text{wait}} = 0.5$
$\text{Del.}_{\text{short}} = \{f: 0.33, g: 0.67\}$
$\gamma_{\text{long}}[R_3] = 2$
$\gamma^C_{\text{short}} = 3$
$p^C = \{f: 2\}$
Synchronization interval (B,C)
0
1
2
3
4
5
6
7
8
time

Figure 3.5.: The short-term waiting time exceeds the length of the local wait state on **B**, but is not sufficient to cover all local routines, leading to a proportional assignment for computations.

hard to predict whether the alternative model provides any new insights when used on real-world applications, but as it requires exact equality even minor inaccuracies lead to this case not being triggered making it very unlikely to occur in real-world applications. This will be examined on several using several examples later on in the evaluating section of this thesis.

Case 2.2: $\gamma^{\text{rem}}_{\text{short}} < \text{wait}_t + \text{comp}_t$

If the short-term costs suffices to cover all local wait states, but the remaining waiting time is less than there is computation, the amount of locally assigned delay-cost differs between both models. Again, the example from Section 3.1 is used for demonstration purposes and is depicted in Figure 3.5 in the adjusted form, alongside the calculations using the alternative model.

The operations on **B** and **C** are again considered to be recognized as different computations. The three units of waiting time that are send to **B** are first considered for the local wait state. This leaves one unit to be distributed onto the computations. Therefore, one third of the remaining waiting time is attributed to the computational part of R_2, while two thirds are assigned to g. As this distribution is done only proportionally to the total local computation time and not proportional over the sum of computation time and wait state like the current model, both models differ in their results. Table 3.3 shows the results for this example using both models, omitting the detailed calculation for briefness sake.

Process	Current model		Alternative model	
	A	B	A	B
$\mathrm{Del}_{\mathrm{short}}$	$f\colon 0.67, g\colon 1.33$	$R_2\colon 0.6, g\colon 1.2$	$f\colon 0.67, g\colon 1.33$	$f\colon 0.33, g\colon 0.67$
$\mathrm{Del}_{\mathrm{long}}$	$f\colon 0.4, g\colon 0.8$	0	$f\colon 0.67, g\colon 1.33$	0
γ_{long}	0	$R_2\colon 1.2$	0	$R_2\colon 2$

Table 3.3.: Comparison of the delay-cost calculation for the current and proposed alternative delay-cost model for $\gamma^{\mathrm{rem}}_{\mathrm{short}} < \mathrm{wait}_t + \mathrm{comp}_t$.

3.5. Implementation and correctness checking of the alternative model

The proposed alternative model was implemented in Scalasca in the scope of this thesis. The implementation is based on a development version based on Scalasca 2.6.0 (master@5a35cf70). All previously available functionalities where retained by adjusting every currently supported wait state pattern to follow the new delay-cost model. This also included minor adjustments in the calculations for the classification of propagating/terminal and direct/indirect wait states as it is also based on the delay-cost calculation.

To ensure that the measurement differences of both models used for evaluation are only caused by the changed distribution of the waiting time and not by side effects, several correctness checks were performed. The Scalasca testsuite is designed to be used to check that the calculated results do not change when Scalasca is updated to a newer version. This is done by using a reference solution and comparing it to the results of several benchmark test cases. Because the goal of the proposed model is to change the calculations, the testsuite reports test failures when comparing against the reference version. By removing the test cases for the delay-cost metrics it was ensured, that if an error is reported it is not directly related to the changes in the delay cost. After this modification, no errors occurred indicating no side effects caused by the new implementation.

To ensure correct behavior of the delay-cost calculation for all supported patterns, several test cases were written and executed to check the behavior of the implementation. All test cases are written in C and where compiled using the Intel MPI C compiler, prefixed with the Score-P instrumentor. Test cases were constructed for all MPI point-to-point(late sender and late receiver) and collective patterns(barrier, 1×N, N×1, N×N), the OpenMP barrier and hybrid executions of MPI point-to-point and collective patterns combined with an OpenMP barrier. Each pattern was tested with four different cases: (1) minimal execution with 2 processes to test behavior in the absence of propagation, (2) incoming waiting time smaller than local wait states(Case 1: Section 3.4.1), (3) incoming waiting time equal to sum of local wait states and computations(Case 2.1: Section 3.4.2), and (4) incoming waiting

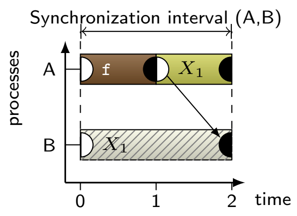


Figure 3.6.: Timeline diagram of two process providing a template for all MPI wait states, where X equals the corresponding operations.

time greater than local wait states, but smaller than sum of local wait states and computation(Case 2.2: Section 3.4.2).

However, OpenMP could only be tested for (1) and the hybrid execution, due to technical limitations of Scalasca.

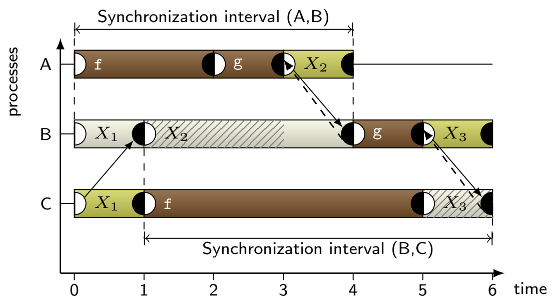
3.5.1. Correctness of calculations in the absence of propagated time

The first batch of tests is designed to check the behavior of the implementation, when no waiting time is propagated. This should ensure that the calculations are correct, even if no long-term costs are received, to show that the model still works if only short-term costs are distributed.

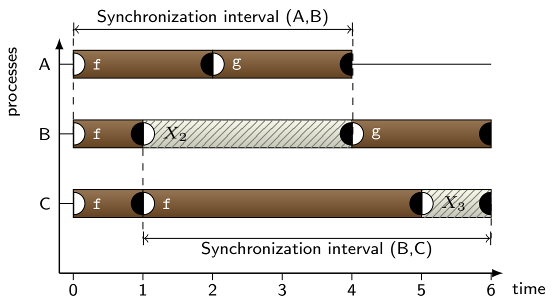
Figure 3.6 shows a template for all tests that where build for this case, based on a two process execution. The placeholder operations X are replaced by the corresponding operations for a specific test, i.e., in the case of a late sender, X on process **A** equals the send operation, while X on process **B** equals the receive operation. The arrow depicts the communication direction for the point-to-point patterns. In case of the collective operations no such direction exist, as the costs are sent towards the process that arrived at last. All test cases where analyzed using the implemented alternative delay-cost model and correctly assigned the one unit of waiting time. This was the only non-hybrid test case, that could be test for OpenMP, as propagation between barrier wait states is only possible in nested OpenMP parallelism, which is not yet supported by Scalasca. For OpenMP **A** in Figure 3.6 the computation f was placed inside a "single" directive, to ensure execution on only on thread without introducing an implicit barrier. In total seven test cases were constructed and executed for this case, two for the point-to-point patterns, four for the collective patterns and one for OpenMP.

3.5.2. Correct behavior in Case 1

For this and the following cases, all tests were executed by a Scalasca build using the current delay-cost model and a Scalasca build using the implementation of the alternative delay-cost model. If in a synchronization interval the incoming short-term waiting time is less than the length of the wait state, both models results


Synchronization interval (A,B)
processes
A
B
C
f
g
X_2
X_1
X_3
Synchronization interval (B,C)
0
1
2
3
4
5
6
time

(a) Template structure of the test cases for case 1 using placeholder operations X.

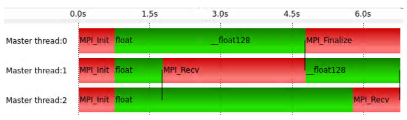

Synchronization interval (A,B)
processes
A
B
C
f
g
X_2
X_3
Synchronization interval (B,C)
0
1
2
3
4
5
6
time

(b) Template structure adjusted to compensate the short completion times of communication operations.

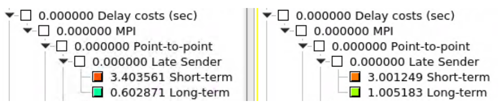
Figure 3.7.: Timeline diagrams of the test case for case 1.

in different distributions like it was explained for case 1 in Section 3.4.1. The difficulty in writing the test cases arises from the assumption made by the timeline diagram, that the communication and synchronization calls take at least one time unit to complete. This deviates from behavior of the implemented test cases. If no wait states occur, an operation like a MPI-Send only needs a few microseconds to be completed. This leads to a difference in the delay costs between the execution planned in a timeline diagram and the implemented test case. However, this behavior can be mitigated by adding dummy operations that fill-up the time not spent by the communication calls. Figure 3.7a shows the planned program to test the correctness for case 1. Again these figures use placeholders X_i for the operations, as all patterns can be tested by just exchanging the used MPI operation. The wait state on **C** is smaller than the wait state on **B**, representing the case where the incoming short-term waiting time does not suffice to fill the local wait state completely. Figure 3.7b provides a visualization of the adapted test case from Figure 3.7a to match the desired behavior in the execution. All computation fractions of MPI operations were replaced by adding or extending computation time equal to the time that is expected to be required for the operations to be performed in theory.

To ensure correctness, it is tested that the theoretically expected values for both models match the results of the execution. In this case, the current delay-cost model will assign delay-costs to the computation g on **B**, while the alternative model is expected to only assign delay-costs to the wait state, and therefore propagate it further to **A**. The trace-file visualizer VAMPIR was used to ensure coherency between the programmed and recorded test case. Figure 3.8a shows a screenshot of the trace recorded during the execution of the test case for the late sender pattern. The overhead at the beginning can be ignored, as it is caused by the initialization of MPI and is present on all processes. The MPI-Finalize at the end on process **A** can also be ignored as it has no impact on the calculations. When comparing Figure 3.8a to the expected behavior from Figure 3.7b, the similarity is sufficient to allow reasoning about the correctness. Figure 3.8b shows a screenshot from the



(a) Trace file of the late sender test case for case 1, visualized in VAMPIR.



```
▼ ☐ 0.000000 Delay costs (sec)
  ▼ ☐ 0.000000 MPI
    ▼ ☐ 0.000000 Point-to-point
      ▼ ☐ 0.000000 Late Sender
          🟧 3.403561 Short-term
          🟩 0.602871 Long-term
```

```
▼ ☐ 0.000000 Delay costs (sec)
  ▼ ☐ 0.000000 MPI
    ▼ ☐ 0.000000 Point-to-point
      ▼ ☐ 0.000000 Late Sender
          🟧 3.001249 Short-term
          🟩 1.005183 Long-term
```

(b) Analysis report in the Cube report explorer showing the delay-cost distribution for the late-sender test case for case 1.

Figure 3.8.: Screenshots of the trace analysis in VAMPIR and Cube of the test case for case 1.

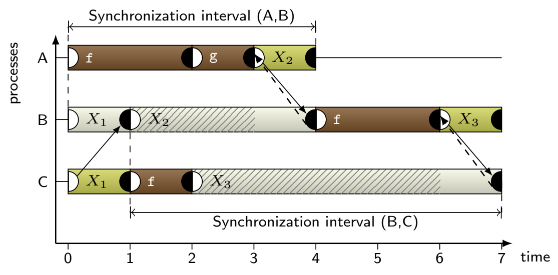
Cube report explorer that includes the assigned delay-costs for the late sender test case. The structure of the analysis report is explained later in the evaluation chapter in detail. For now, it is just used to grant a short insight in the cost distribution for the test cases. The left side shows the distribution for the current model, showing the splitting of the waiting time from the last receive operation into long and short-term costs. The right side shows the distribution for the alternative model. In this case the long-term costs are approximately one time unit proving the expected full propagation of the waiting time from process **C**.

All other patterns were tested using the same methodology and provided the expected results. The structure of the computations is slightly different in the hybrid test cases, as these also involve OpenMP threads. In order to test the calculation of long-term OpenMP barrier delay costs, it was necessary to use MPI operations on the master thread through the funnelled communication pattern. This allows MPI operations exclusively on the master thread of an OpenMP parallel region. In order to cover all patterns, this case required a total of nine test cases. Two for point-to-point, four for collective and three for the hybrid patterns.

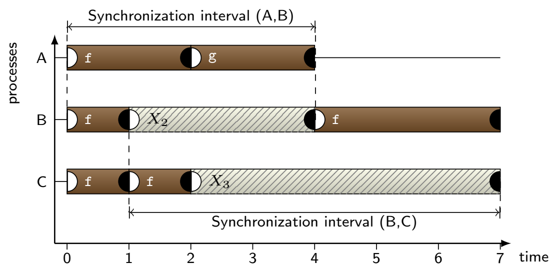
3.5.3. Correct behavior in Case 2.1

The third batch of test cases cover case 2.1 from Section 3.4.2. The used methodology is equal to the one used for case 1. In order to test the desired behavior of the alternative model, a test cases, shown in Figure 3.9a was constructed that meets the requirements for this case. This requires the direct waiting time experienced by process **C** in the synchronization interval (**B**,**C**) to be equal to the sum of the length of all local wait states on **B** and the computation included in the difference profile in the same synchronization interval. As both processes performed computations in form of routine *f*, the difference profile only includes two time units for that routine on **B**. Again, the adjusted version of this program is presented in Figure 3.9b.

The amount of performed test cases is equal to the previous case. Using the explained test case correctness could be proven for all cases except the hybrid variant using the funnelled communication. In this case it was not possible to exactly align the lengths of the routines, resulting in a small deviation between the expected and observed behavior. As far as it can be judged, this is not caused by a faulty



(a) Template structure of the test cases for case 2.1 using placeholder operations X.


Synchronization interval (A,B)
processes
A
B
C
f
g
f
X_2
f
f
f
X_3
Synchronization interval (B,C)
0 1 2 3 4 5 6 7
time

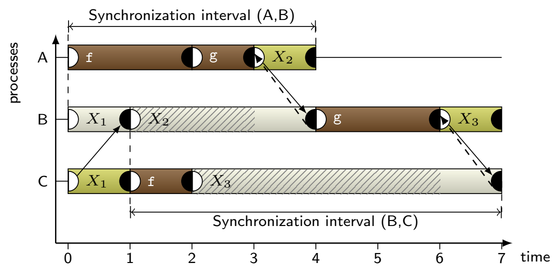
(b) Template structure adjusted to compensate the short completion times of communication operations.

Figure 3.9.: Timeline diagrams of the test case for case 2.1.

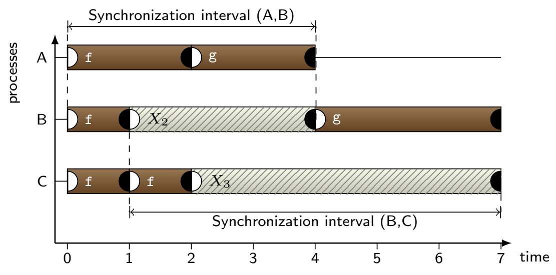
implementation of the alternative model, and is most likely to be caused by small fluctuations in the execution on the difference threads. As this case requires exact equality of the incoming waiting time and the amount that is distributed locally, it is not possible to check this case as long as this fluctuations can not be compensated. Since none of the benchmarks that are used in the evaluation of this thesis uses funnelled communication, the lack of a proofed correctness for this test cases is not critical.

3.5.4. Correct behavior in Case 2.2

The last batch of test cases concern case 2.2 as covered in Section 3.4.2. Performing the same steps as for the previous two cases, this case involves the incoming waiting time to be larger than the local wait state, but smaller than the sum of the local wait states and the computations from the difference profile as shown in Figure 3.10a. The adjusted version can be found in Figure 3.10b.



(a) Template structure of the test cases for case 2.2 using placeholder operations X.


Synchronization interval (A,B)
processes
A
B
C
f
g
f
X_2
g
f
f
X_3
Synchronization interval (B,C)
0 1 2 3 4 5 6 7
time

(b) Template structure adjusted to compensate the short completion times of communication operations.

Figure 3.10.: Timeline diagrams of the test case for case 2.2.

For this case correctness could be observed for all patterns, including a total of nine test cases covering this behavior.

3.6. Summary

At this point it was explained why an alternative delay-cost model may provide new insights into Scalasca's performance analysis. The weak-points of the current model were considered in the design of the alternative model. The provided formulas provide a full and sound model as shown the by given proofs. The alternative model was implemented for evaluation purposes and provides correct results in synthetic test cases. This leads to an examination of the alternative model's potential using real-world applications in the following evaluation.

4. Evaluation

The previous chapter covered the alternative model and provided an implementation that was checked for correctness on synthetic test cases and made certain predictions about its behavior. These are investigated in this chapter by using the model on real-world applications and comparing it to the analysis results provided by the previous delay-cost model. In the scope of this evaluation, the current delay-cost model is referred to as the “reference” model, while the proposed model is referred to as the “alternative” model. All experiments were performed on the c18m partition of the CLAIX18 supercomputer of the RWTH Aachen University. The partition features around 1250 nodes with two sockets per node, equipped with 24-core processors resulting in a total of 48 cores per node, supported by 192 GB of RAM [1]. All runs were either configured to fill up all cores of the used nodes, or run exclusively on the given nodes if the benchmark configuration did not allow an usage of all cores per node. A total of five different experiments were performed by using four different benchmarks: (1) the BT benchmark from the NAS Parallel Benchmarks(NPB) [2] in both a pure MPI and a hybrid MPI+OpenMP experiment, (2) Sweep3D [21], (3) SMG2000 [16], and (4) TeaLeaf [3]

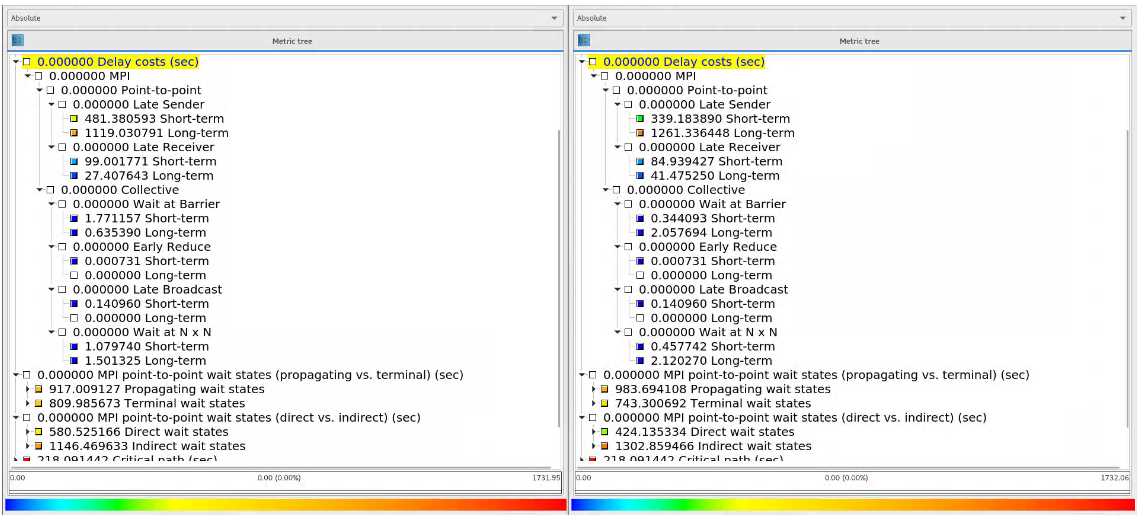
All of them were already analyzed by a version of Scalasca at some point [16] [36]. Each experiment used five trace files that were recorded by different executions of the corresponding benchmark, using the same configuration and inputs. These trace files were analyzed by both models and the resulting CUBE reports were aggregated to compute the mean for all metrics for both models, resulting in one averaged report per model and benchmark.

4.1. NPB BT

The NPB Block Tri-diagonal(BT) solver is one of the pseudo applications contained in the NPB [2] and is written in Fortran. It solves three sets of uncoupled systems of equations in three dimensions that result from a discretization of the Navier-Stokes equations [6]. The chosen problem size “D” performs 250 time-steps and was executed with 144 processes for the pure MPI version or 24 processes with 6 threads each for the hybrid version, which is known as the multi-zone variant of the benchmark. This number of processes was chosen, because the benchmark requires a quadratic number of processes and it completely fills up three nodes on CLAIX18. All runs were executed and measured with NPB version 3.3.

4.1.1. MPI only

As a first step it will be investigated how the alternative model changes the analysis results when being executed on the same traces.



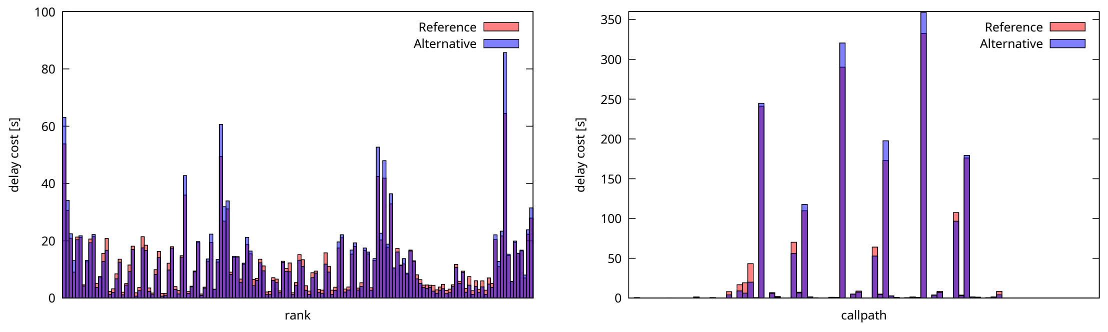
```
Absolute
Metric tree
▾☐ 0.000000 Delay costs (sec)
  ▾☐ 0.000000 MPI
    ▾☐ 0.000000 Point-to-point
      ▾☐ 0.000000 Late Sender
          481.380593 Short-term
          1119.030791 Long-term
      ▾☐ 0.000000 Late Receiver
          99.001771 Short-term
          27.407643 Long-term
    ▾☐ 0.000000 Collective
      ▾☐ 0.000000 Wait at Barrier
          1.771157 Short-term
          0.635390 Long-term
      ▾☐ 0.000000 Early Reduce
          0.000731 Short-term
          0.000000 Long-term
      ▾☐ 0.000000 Late Broadcast
          0.140960 Short-term
          0.000000 Long-term
      ▾☐ 0.000000 Wait at N x N
          1.079740 Short-term
          1.501325 Long-term
▾☐ 0.000000 MPI point-to-point wait states (propagating vs. terminal) (sec)
  ▸ 917.009127 Propagating wait states
  ▸ 809.985673 Terminal wait states
▾☐ 0.000000 MPI point-to-point wait states (direct vs. indirect) (sec)
  ▸ 580.525166 Direct wait states
  ▸ 1146.469633 Indirect wait states
▸ 218.091442 Critical path (sec)
0.00                         0.00 (0.00%)                         1731.95
```

```
Absolute
Metric tree
▾☐ 0.000000 Delay costs (sec)
  ▾☐ 0.000000 MPI
    ▾☐ 0.000000 Point-to-point
      ▾☐ 0.000000 Late Sender
          339.183890 Short-term
          1261.336448 Long-term
      ▾☐ 0.000000 Late Receiver
          84.939427 Short-term
          41.475250 Long-term
    ▾☐ 0.000000 Collective
      ▾☐ 0.000000 Wait at Barrier
          0.344093 Short-term
          2.057694 Long-term
      ▾☐ 0.000000 Early Reduce
          0.000731 Short-term
          0.000000 Long-term
      ▾☐ 0.000000 Late Broadcast
          0.140960 Short-term
          0.000000 Long-term
      ▾☐ 0.000000 Wait at N x N
          0.457742 Short-term
          2.120270 Long-term
▾☐ 0.000000 MPI point-to-point wait states (propagating vs. terminal) (sec)
  ▸ 983.694108 Propagating wait states
  ▸ 743.300692 Terminal wait states
▾☐ 0.000000 MPI point-to-point wait states (direct vs. indirect) (sec)
  ▸ 424.135334 Direct wait states
  ▸ 1302.859466 Indirect wait states
▸ 218.091442 Critical path (sec)
0.00                         0.00 (0.00%)                         1732.06
```

Figure 4.1.: Screenshot from the Cube-GUI comparing the means of the reference model(left) with the alternative model(right) for the BT benchmark, showing a transition from short-term to long-term costs for the delay costs metric.

Figure 4.1 shows a screenshot from the Cube-GUI that shows the metric tree for both analyses. The values on the left side show the reference model, while the right side refers to the alternative model. The metrics can be divided into delay costs, which are further split into point-to-point and collective patterns, and the classifications into propagating/terminal wait states and direct/indirect wait states. Comparing the delay costs of both models shows a general trend of a redistribution of short-term cost to long-term cost. In total 1731.95 s of delay costs were measured by the reference model and 1732.06 s by the alternative model. The slight deviation in total delay costs is most likely caused by rounding errors due to floating-point arithmetic. In case of the late-sender wait state, which caused a majority of the total delay costs, approximately 30% of the short-term costs assigned by the reference model were instead considered long-term cost by the alternative model. Similar behavior can be observed for the remaining patterns. This trend corresponds to the expected behavior that was theoreticized in the previous chapter and reflects the increased priority of intermediate wait states. Besides the delay costs, the classification of wait states now also relies on the calculations from the alternative delay-cost model. The alternative model classifies more wait states as propagating, which matches the desire for a better mapping of costs for increased delay-cost propagation and therefore reducing the amount of waiting time assigned to terminal wait states. Furthermore, the increased propagation results in classifying more wait states as indirect, which is caused by the higher priority of wait states.

To check whether the alternative distribution provides different insights into the behavior of the benchmark, the next step involves comparing the distribution of the delay costs among ranks and callpaths.

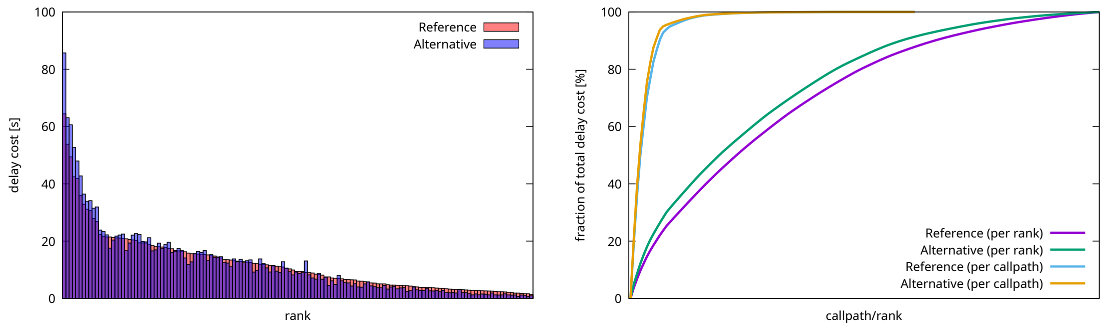


(a) Aggregated by rank.

(b) Aggregated by callpath.

Figure 4.2.: Aggregated delay costs of both models for the BT benchmark show a stronger highlighting of previous maxima.

Figure 4.2a shows the delay costs per rank. A red bar tip indicated that the assigned cost was higher in the reference model, a blue bar shows higher cost in the alternative model and a purple bar shows the value of the respective other model. It is visible that the ranks with the highest delay cost in the reference model got even more delay costs through the alternative model. At the same time nearly all smaller ranks lost assigned cost, creating a larger gap between the largest and smallest ranks. This leads to a stronger consolidation of costs on the maxima and strengthens the statements of the reference model, as both models have the same largest ranks, providing a clearer classification of the most probable root cause. A



(a) Sorted based on severity in the reference model.

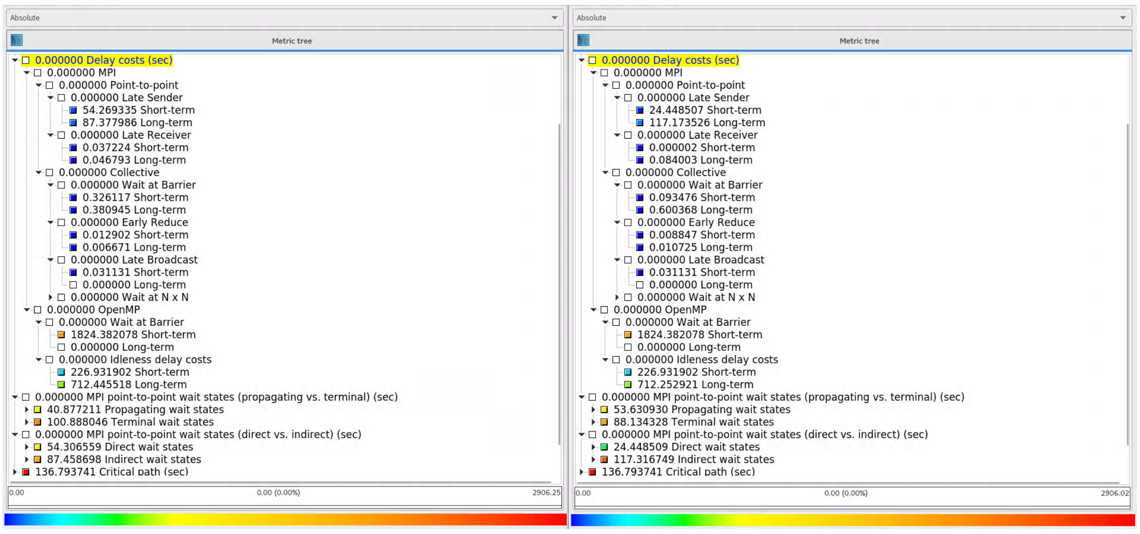
(b) Visualizing the fractions of total delay costs sorted independently by severity for each model.

Figure 4.3.: Aggregating the delay costs of both models by rank and sorting them in descending order results in a stronger delimitation from the maxima for the BT benchmark.

similar behavior can be seen in Figure 4.2b, where the delay costs are aggregated in order of severity for each callpath that was measured in the trace. The delimitation of ranks with high delay costs from ranks with lower delay costs is better visualized in Figure 4.3a. The ranks are sorted by the amount of assigned delay costs in the reference model, visualizing which ranks are promoted through the alternative model. Wait states with low delay costs are most likely propagating and loose delay cost in the alternative delay-cost model. Following the pareto principle, Figure 4.3b visualizes how many ranks/callpaths are required to reach a certain fraction of the total delay cost. Therefore each dataset was sorted independently by the amount of delay costs. For both, ranks and callpaths, the alternative model shows a steeper incline, corresponding to the stronger consolidation of costs on ranks starting a cost propagation chain.

4.1.2. Hybrid

The hybrid version of the BT benchmark is officially known as the multi-zone (MZ) version of the benchmark and allows the use of multiple parallel programming models simultaneously. For this experiment the hybrid MPI+OpenMP execution uses the same problem size as the MPI only experiment, but with an adjusted distribution across processes and threads like mentioned earlier.

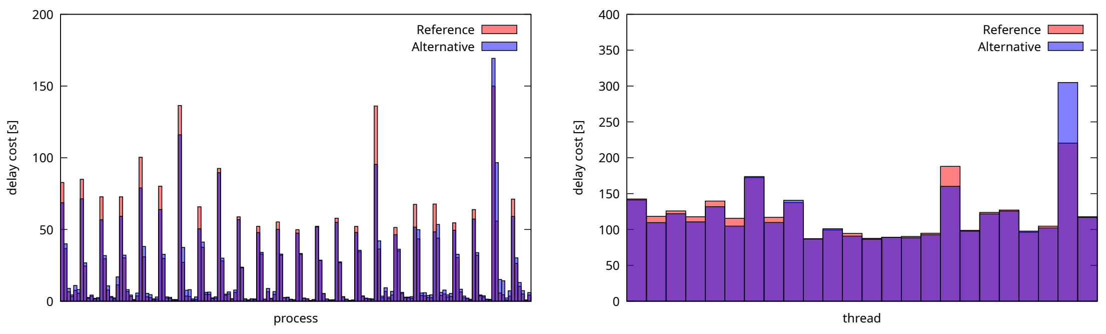


```
0.000000 Delay costs (sec)
  0.000000 MPI
    0.000000 Point-to-point
      0.000000 Late Sender
        54.269335 Short-term
        87.377986 Long-term
      0.000000 Late Receiver
        0.037224 Short-term
        0.046793 Long-term
    0.000000 Collective
      0.000000 Wait at Barrier
        0.326117 Short-term
        0.380945 Long-term
      0.000000 Early Reduce
        0.012902 Short-term
        0.006671 Long-term
      0.000000 Late Broadcast
        0.031131 Short-term
        0.000000 Long-term
      0.000000 Wait at N x N
  0.000000 OpenMP
    0.000000 Wait at Barrier
      1824.382078 Short-term
      0.000000 Long-term
    0.000000 Idleness delay costs
      226.931902 Short-term
      712.445518 Long-term
0.000000 MPI point-to-point wait states (propagating vs. terminal) (sec)
  40.877211 Propagating wait states
  100.888046 Terminal wait states
0.000000 MPI point-to-point wait states (direct vs. indirect) (sec)
  54.306559 Direct wait states
  87.458698 Indirect wait states
136.793741 Critical path (sec)
0.00          0.00 (0.00%)          2906.25
```

```
0.000000 Delay costs (sec)
  0.000000 MPI
    0.000000 Point-to-point
      0.000000 Late Sender
        24.448507 Short-term
        117.173526 Long-term
      0.000000 Late Receiver
        0.000002 Short-term
        0.084003 Long-term
    0.000000 Collective
      0.000000 Wait at Barrier
        0.093476 Short-term
        0.600368 Long-term
      0.000000 Early Reduce
        0.008847 Short-term
        0.010725 Long-term
      0.000000 Late Broadcast
        0.031131 Short-term
        0.000000 Long-term
      0.000000 Wait at N x N
  0.000000 OpenMP
    0.000000 Wait at Barrier
      1824.382078 Short-term
      0.000000 Long-term
    0.000000 Idleness delay costs
      226.931902 Short-term
      712.252921 Long-term
0.000000 MPI point-to-point wait states (propagating vs. terminal) (sec)
  53.630930 Propagating wait states
  88.134328 Terminal wait states
0.000000 MPI point-to-point wait states (direct vs. indirect) (sec)
  24.448509 Direct wait states
  117.316749 Indirect wait states
136.793741 Critical path (sec)
0.00          0.00 (0.00%)          2906.02
```

Figure 4.4.: Screenshot from the Cube-GUI comparing the means of the reference model(left) with the alternative model(right) for the BT-MZ benchmark.

This variant of the benchmark shows similar behavior as the MPI only variant, showing an overall redistribution of delay-costs from short-term to long-term, as shown in Figure 4.4. It is visible that the majority of costs is caused by OpenMP patterns. As already mentioned in the section about the capabilities of the delay-cost models, the idleness delay-costs are calculated separately and are not affected by the changes. The small observable difference in long-term costs is most likely caused by non-determinism in this calculation, as the threads add their costs in an arbitrary order, making the sum susceptible to rounding errors. The changes in the wait-state classification are also similar to the previously observed changes, with a most notably reclassification of direct wait states, resulting in a larger gap between the amount of direct and indirect waiting time. As this experiment includes both processes and threads, they will be referred to as locations if no further differentiation is required. In comparison to the pure MPI variant, the multi-zone experiment yields different results in the delay-cost distribution across locations.



(a) All threads of the processes considered separately. (Large peaks correspond to main threads of a process).

(b) All threads of a process are aggregated together.

Figure 4.5.: Delay costs of both models aggregated by location show a significant increase for one process for the BT-MZ benchmark.

While it is still visible in Figure 4.5a that both models assign the highest costs to the same location and the alternative model assigns more costs to this location, the other locations with high costs became smaller. An exception to this is the location right beside the maximum, which experienced a significant boost in delay-costs. The figure suggests that the multi-zone variant leads to an increase of delay-costs through the alternative model on the threads with the smallest assigned costs in the reference runs. The observed behavior is caused by a redistribution of the computational load as an effect of the division of each process into six threads. For a cleaner representation, all threads belonging to the same process are merged together in Figure 4.5b. This representation changes the previously made statement, that the threads with the smallest severity were assigned higher values by using the alternative model. Therefore the following figures will continue to use the merged threads.

Further difficulties in evaluating the analysis results for this benchmark arise when viewing the delay-cost distribution across callpaths in Figure 4.6. Three callpaths contain the majority of the delay-costs. These are the parallel regions for each dimensional direction computed by the benchmark. The detailed structure is masked by the parallel region and is not visible to Scalasca's analyzer. This does not allow

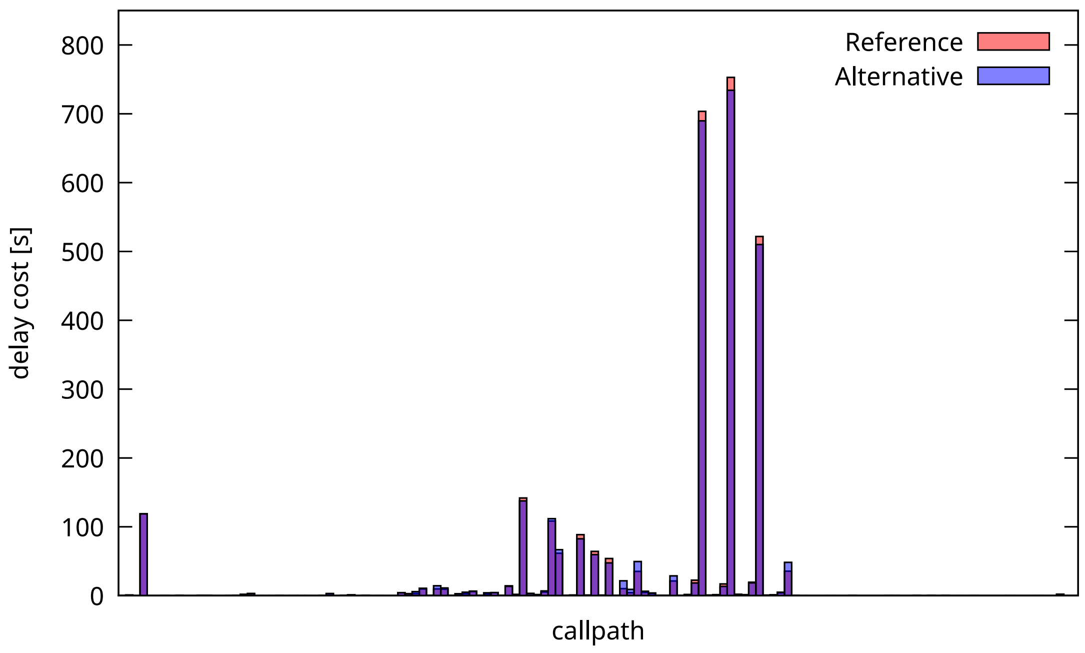


Figure 4.6.: Delay costs of both models aggregated by callpath for the BT-MZ benchmark.

further investigation of the cost distribution, and while the overall costs of these regions are smaller when using the alternative model, it could still have the predicted effects on the sub-routines.

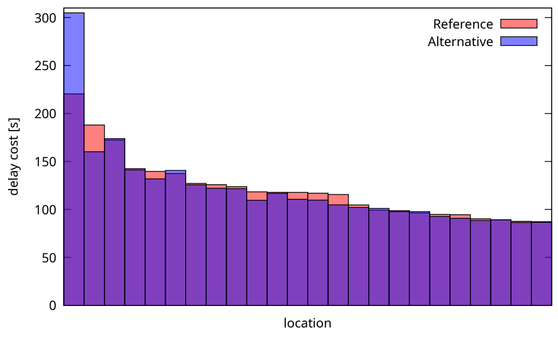
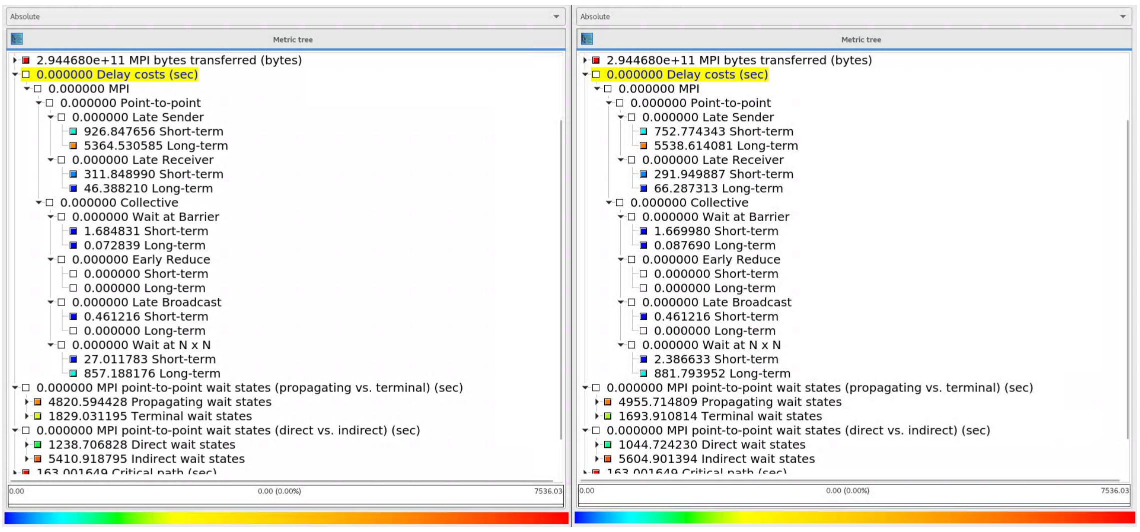


Figure 4.7.: Delay costs of both models aggregated by location and sorted based on the location order of the reference model.

Sorting the ranks by the amount of delay costs assigned by the reference model, as depicted in Figure 4.7, shows a significant increase for the largest root-cause, creating a larger difference between the top-two candidates. As the detailed structure of the delay-costs per callpath can not be investigated, a comparison between the distribution per location and per callpath is not suitable.

4.2. Sweep3D

The Sweep3D benchmark solves a particle transport problem in three-dimensional space [21]. The domain resembles a cube and the calculations causing a diagonal wavefront passing through the domain. The benchmark, which is written in Fortran77, uses only MPI for parallelization and was used in version 2.2b. The number of processes is required to be a power of two and was therefore chosen as 256, because the problem works best on quadratic domains. The problem size on the x and y axis were set to 2048 to achieve a reasonable runtime and use most of the available memory. As the chosen number of processes is not perfectly distributable across full nodes on CLAIX18, it was ensured that the corresponding jobs were executed on exclusive nodes to prevent influences by other applications running on the same nodes.



```
Absolute
Metric tree
2.944680e+11 MPI bytes transferred (bytes)
0.000000 Delay costs (sec)
  0.000000 MPI
    0.000000 Point-to-point
      0.000000 Late Sender
        926.847656 Short-term
        5364.530585 Long-term
      0.000000 Late Receiver
        311.848990 Short-term
        46.388210 Long-term
    0.000000 Collective
      0.000000 Wait at Barrier
        1.684831 Short-term
        0.072839 Long-term
      0.000000 Early Reduce
        0.000000 Short-term
        0.000000 Long-term
      0.000000 Late Broadcast
        0.461216 Short-term
        0.000000 Long-term
      0.000000 Wait at N x N
        27.011783 Short-term
        857.188176 Long-term
0.000000 MPI point-to-point wait states (propagating vs. terminal) (sec)
  4820.594428 Propagating wait states
  1829.031195 Terminal wait states
0.000000 MPI point-to-point wait states (direct vs. indirect) (sec)
  1238.706828 Direct wait states
  5410.918795 Indirect wait states
163.001649 Critical path (sec)
0.00                0.00 (0.00%)                7536.03
```

```
Absolute
Metric tree
2.944680e+11 MPI bytes transferred (bytes)
0.000000 Delay costs (sec)
  0.000000 MPI
    0.000000 Point-to-point
      0.000000 Late Sender
        752.774343 Short-term
        5538.614081 Long-term
      0.000000 Late Receiver
        291.949887 Short-term
        66.287313 Long-term
    0.000000 Collective
      0.000000 Wait at Barrier
        1.669980 Short-term
        0.087690 Long-term
      0.000000 Early Reduce
        0.000000 Short-term
        0.000000 Long-term
      0.000000 Late Broadcast
        0.461216 Short-term
        0.000000 Long-term
      0.000000 Wait at N x N
        2.386633 Short-term
        881.793952 Long-term
0.000000 MPI point-to-point wait states (propagating vs. terminal) (sec)
  4955.714809 Propagating wait states
  1693.910814 Terminal wait states
0.000000 MPI point-to-point wait states (direct vs. indirect) (sec)
  1044.724230 Direct wait states
  5604.901394 Indirect wait states
163.001649 Critical path (sec)
0.00                0.00 (0.00%)                7536.03
```

Figure 4.8.: Screenshot from the Cube-GUI comparing the means of the reference model(left) with the alternative model(right) for Sweep3D.

The analysis results shown in Figure 4.8 provide a similar view onto the cost redistribution like the BT MPI variant, further confirming the expected behavior of the implemented model.
The aggregated delay costs across ranks in Figure 4.9 also show similar patterns for the alternative model by further increasing delay-costs on ranks that were already among the top contributors in the analysis of the reference model. The exception are two ranks on the right edge of the plot, that loose some notable costs. The overview about cost distribution per callpath in Figure 4.10 includes one callpath that contributes nearly the entire waiting time. This callpath received some additional cost in comparison to the reference model. The zoomed views of the bars show, that the callpath with the second highest waiting time looses delay costs in the alternative model, which mainly shifted towards the largest callpath.
Sorting the aggregated delay-costs by rank, as shown in Figure 4.11 underlines the

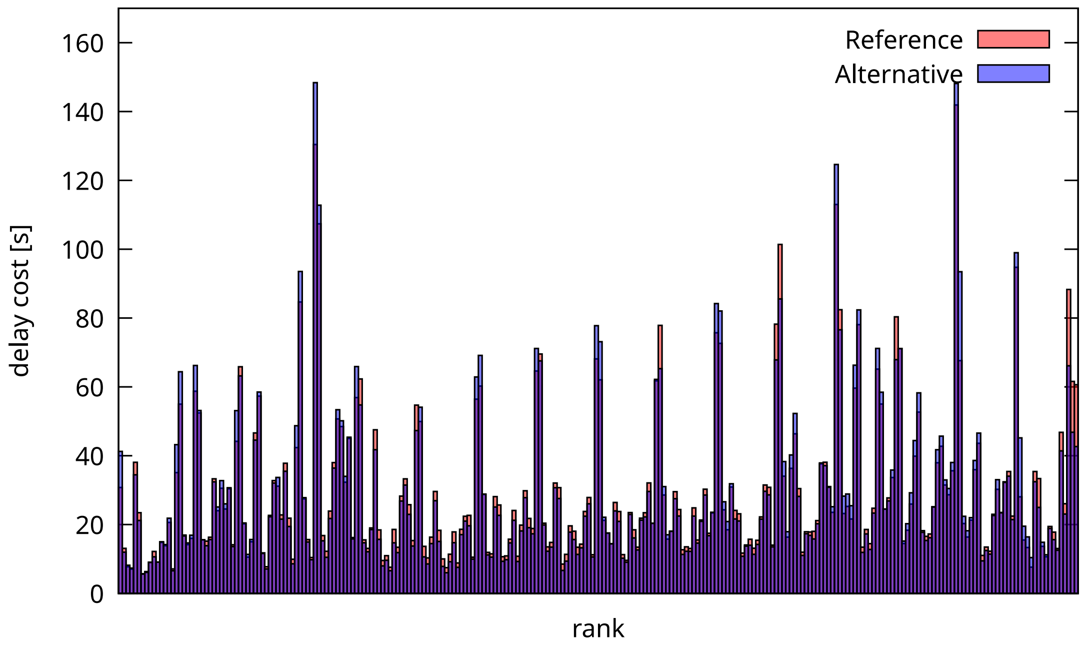


Figure 4.9.: Delay cost of both models aggregated by rank for Sweep3D.

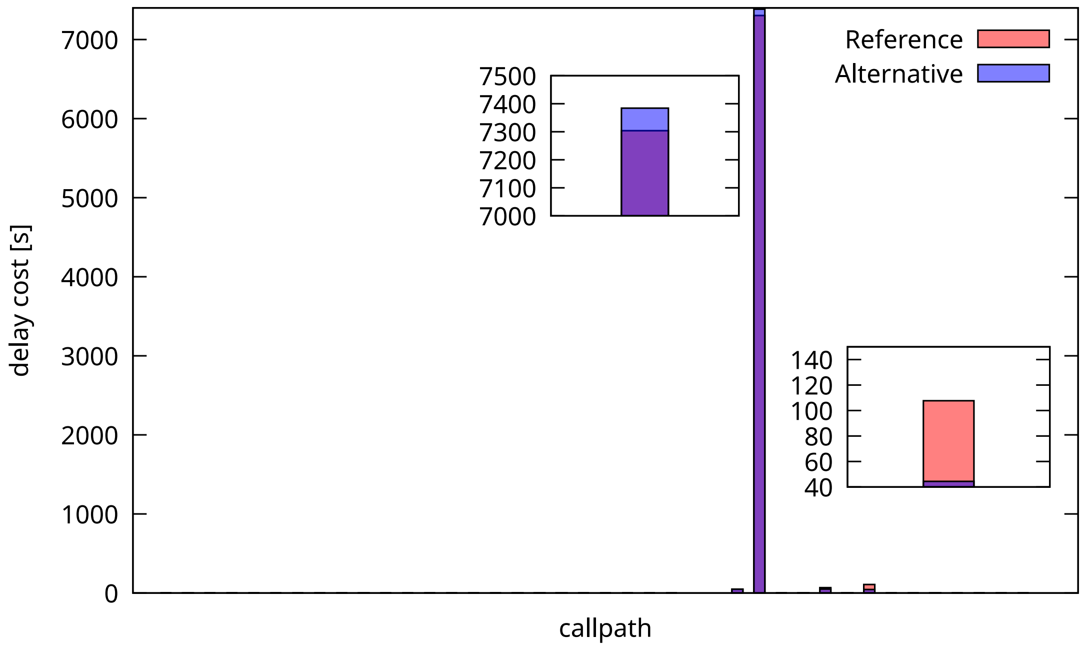


Figure 4.10.: Delay costs of both models aggregated by callpath for Sweep3D.

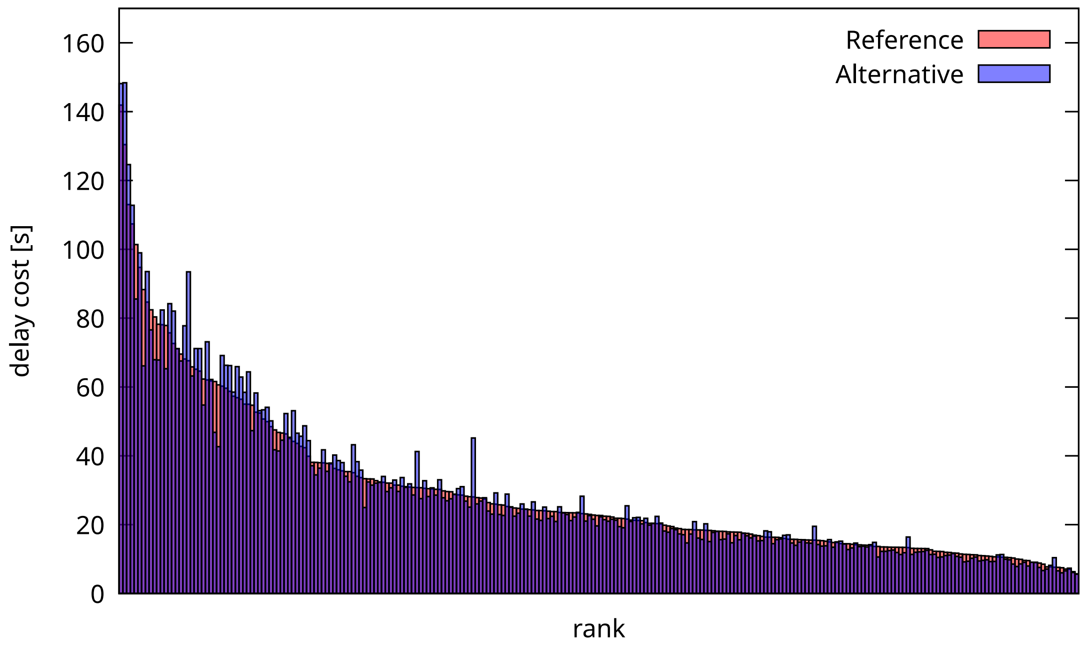


Figure 4.11.: Delay costs for Sweep3D of both models aggregated by rank and sorted based on the severity in the reference model, indicating an increase on the ranks with high delay costs.

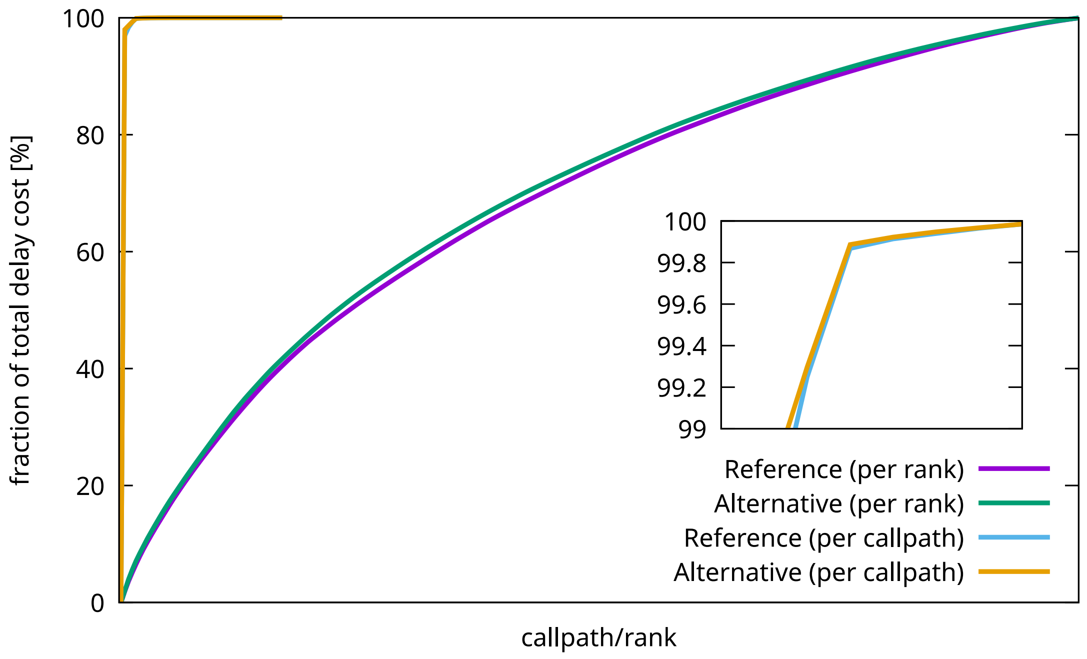
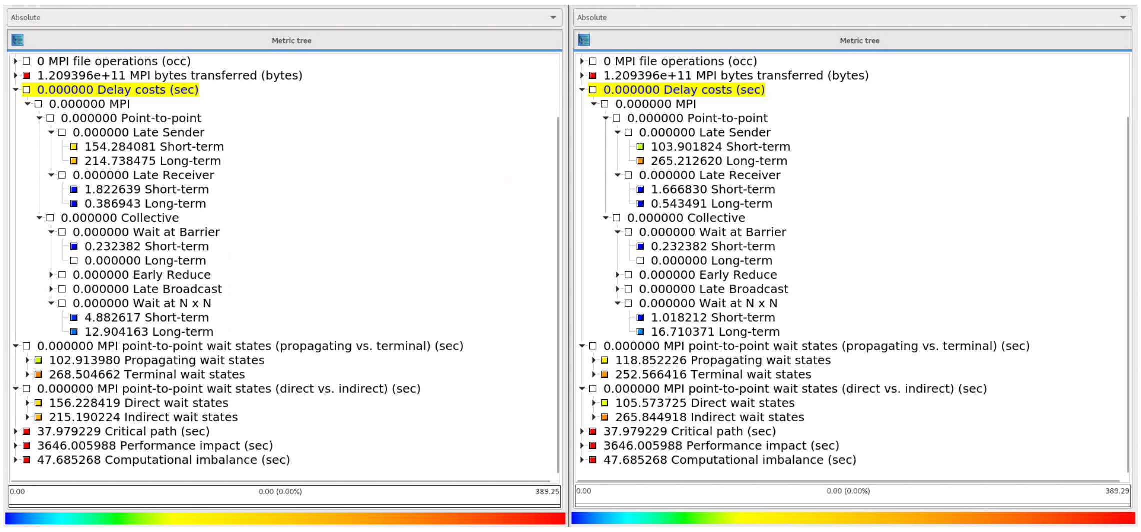


Figure 4.12.: Visualizing the fractions of total delay costs aggregated by rank of both models sorted by severity for each model.

previous statements, but reveals a bit more fluctuations on medium to large severities (on the left), where a notable shift among some ranks can be observed. Examining the pareto principle for Sweep3D in Figure 4.12 shows only minor differences for the per rank plot in favor of the alternative model. The difference per callpath is stronger, but lies primarily in the center, which argues against a notable increase in significance for the largest callpath.

4.3. SMG2000

The SMG2000 benchmark is a parallel semicoarsening multigrid solver for linear systems [10] and features a lot of MPI communication [16]. This makes it suitable to investigate the effects on delay-cost propagation, as lots of communication potentially creates many synchronization point, which increase the potential for propagation. It was executed with 96 MPI processes filling two nodes of CLAIX18.

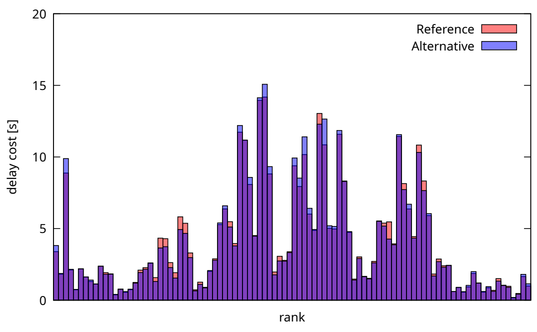


```
0 MPI file operations (occ)
1.209396e+11 MPI bytes transferred (bytes)
0.000000 Delay costs (sec)
  0.000000 MPI
    0.000000 Point-to-point
      0.000000 Late Sender
        154.284081 Short-term
        214.738475 Long-term
      0.000000 Late Receiver
        1.822639 Short-term
        0.386943 Long-term
    0.000000 Collective
      0.000000 Wait at Barrier
        0.232382 Short-term
        0.000000 Long-term
      0.000000 Early Reduce
      0.000000 Late Broadcast
      0.000000 Wait at N x N
        4.882617 Short-term
        12.904163 Long-term
0.000000 MPI point-to-point wait states (propagating vs. terminal) (sec)
  102.913980 Propagating wait states
  268.504662 Terminal wait states
0.000000 MPI point-to-point wait states (direct vs. indirect) (sec)
  156.228419 Direct wait states
  215.190224 Indirect wait states
37.979229 Critical path (sec)
3646.005988 Performance impact (sec)
47.685268 Computational imbalance (sec)

0.00                    0.00 (0.00%)                    389.25
```

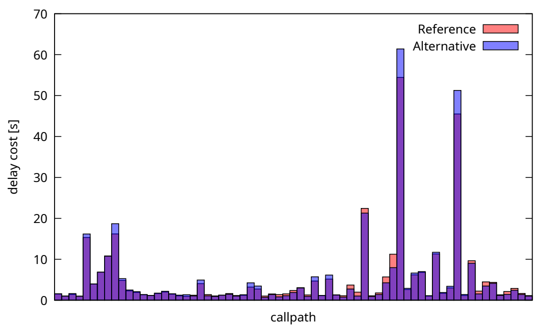
```
0 MPI file operations (occ)
1.209396e+11 MPI bytes transferred (bytes)
0.000000 Delay costs (sec)
  0.000000 MPI
    0.000000 Point-to-point
      0.000000 Late Sender
        103.901824 Short-term
        265.212620 Long-term
      0.000000 Late Receiver
        1.666830 Short-term
        0.543491 Long-term
    0.000000 Collective
      0.000000 Wait at Barrier
        0.232382 Short-term
        0.000000 Long-term
      0.000000 Early Reduce
      0.000000 Late Broadcast
      0.000000 Wait at N x N
        1.018212 Short-term
        16.710371 Long-term
0.000000 MPI point-to-point wait states (propagating vs. terminal) (sec)
  118.852226 Propagating wait states
  252.566416 Terminal wait states
0.000000 MPI point-to-point wait states (direct vs. indirect) (sec)
  105.573725 Direct wait states
  265.844918 Indirect wait states
37.979229 Critical path (sec)
3646.005988 Performance impact (sec)
47.685268 Computational imbalance (sec)

0.00                    0.00 (0.00%)                    389.29
```

Figure 4.13.: Screenshot from the Cube-GUI comparing the means of the reference model(left) with the alternative model(right) for SMG2000.

Figure 4.13 contains the analysis results for both models. The behavior is again similar to the previously investigated MPI examples.
The delay-cost aggregation per rank in Figure 4.14a confirms the previous hypothesis that delay costs shift from short-term to long-term costs. As SMG2000 features more than 1800 different callpaths, Figure 4.14b only contains callpaths with more than 1 s delay-cost. This modification was made, as the majority of callpaths only caused minor amounts of waiting time and the top contributors caused much more waiting time. Two callpaths in particular caused large amounts of waiting time and get even more costs assigned in the alternative delay-cost model.
The changes per rank are much weaker than the increases observed for the BT benchmark, as seen in Figure 4.15a. Figure 4.15b supports that statement by

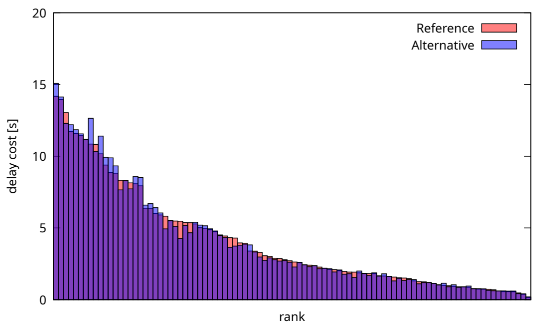


(a) Aggregated by ranks.

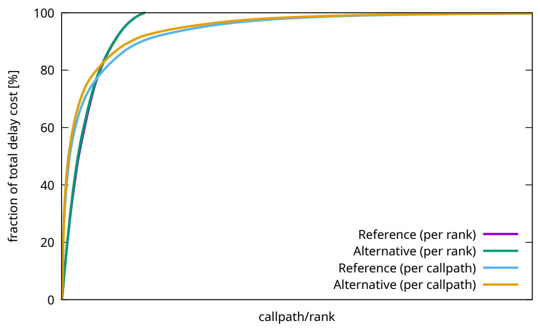


(b) Aggregated by callpath.

Figure 4.14.: Aggregated delay costs of both models for SMG2000.



(a) Sorted based on the rank order of the reference model.



(b) Visualizing the fractions of total delay costs sorted independently for each model.

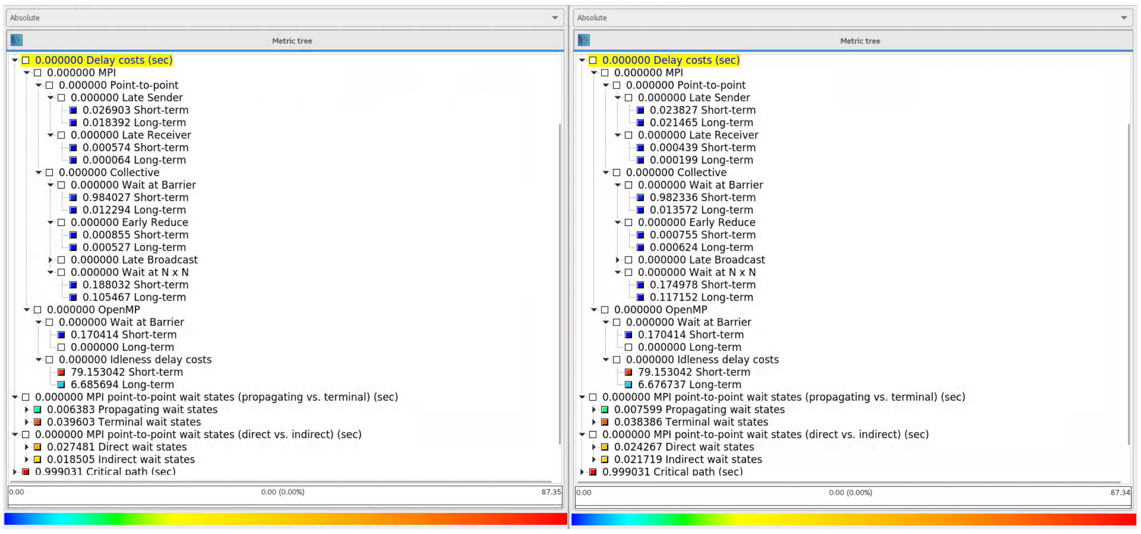
Figure 4.15.: Delay costs of both models aggregated by rank and sorted in descending order for SMG2000.

showing an overlap of the lines for the aggregation per rank, while the sum for the callpaths shows a steeper incline for the alternative model.

4.4. TeaLeaf

TeaLeaf is an application that solves the linear heat conduction equation on a spatially decomposed regularly grid [3]. The benchmark computes two kernels and uses a hybrid MPI+OpenMP model. The traces for this benchmark were recorded for an execution on 96 cores, divided into 16 processes with 6 threads each.

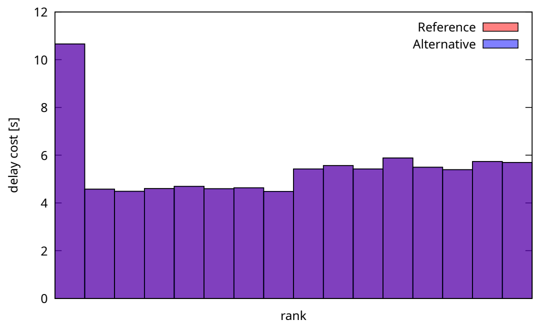
The comparison between both the analyses of both models in Figure 4.16 shows only minor changes that are insignificant compared to the OpenMP idleness costs. As there are no notable wait states that are influenced by changes in the delay-cost model, the analysis yields similar results for both models.
Also no changes in distribution across location (Figure 4.17a) or across callpath (Figure 4.17b) can be observed. The visualization of delay costs across callpaths



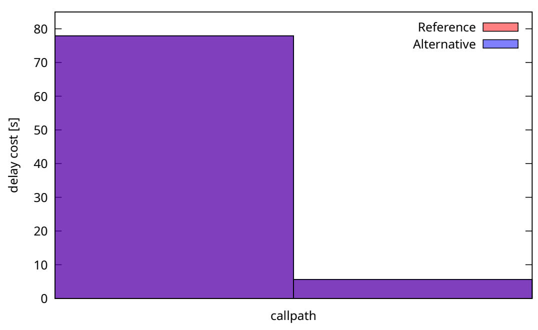
```
Absolute
Metric tree
▾ □ 0.000000 Delay costs (sec)
  ▾ □ 0.000000 MPI
    ▾ □ 0.000000 Point-to-point
      ▾ □ 0.000000 Late Sender
          ■ 0.026903 Short-term
          ■ 0.018392 Long-term
      ▾ □ 0.000000 Late Receiver
          ■ 0.000574 Short-term
          ■ 0.000064 Long-term
    ▾ □ 0.000000 Collective
      ▾ □ 0.000000 Wait at Barrier
          ■ 0.984027 Short-term
          ■ 0.012294 Long-term
      ▾ □ 0.000000 Early Reduce
          ■ 0.000855 Short-term
          ■ 0.000527 Long-term
      ▸ □ 0.000000 Late Broadcast
      ▾ □ 0.000000 Wait at N x N
          ■ 0.188032 Short-term
          ■ 0.105467 Long-term
  ▾ □ 0.000000 OpenMP
    ▾ □ 0.000000 Wait at Barrier
        ■ 0.170414 Short-term
        □ 0.000000 Long-term
    ▾ □ 0.000000 Idleness delay costs
        ■ 79.153042 Short-term
        ■ 6.685694 Long-term
▾ □ 0.000000 MPI point-to-point wait states (propagating vs. terminal) (sec)
  ▸ ■ 0.006383 Propagating wait states
  ▸ ■ 0.039603 Terminal wait states
▾ □ 0.000000 MPI point-to-point wait states (direct vs. indirect) (sec)
  ▸ ■ 0.027481 Direct wait states
  ▸ ■ 0.018505 Indirect wait states
▸ ■ 0.999031 Critical path (sec)
0.00                    0.00 (0.00%)                    87.35
```

```
Absolute
Metric tree
▾ □ 0.000000 Delay costs (sec)
  ▾ □ 0.000000 MPI
    ▾ □ 0.000000 Point-to-point
      ▾ □ 0.000000 Late Sender
          ■ 0.023827 Short-term
          ■ 0.021465 Long-term
      ▾ □ 0.000000 Late Receiver
          ■ 0.000439 Short-term
          ■ 0.000199 Long-term
    ▾ □ 0.000000 Collective
      ▾ □ 0.000000 Wait at Barrier
          ■ 0.982336 Short-term
          ■ 0.013572 Long-term
      ▾ □ 0.000000 Early Reduce
          ■ 0.000755 Short-term
          ■ 0.000624 Long-term
      ▸ □ 0.000000 Late Broadcast
      ▾ □ 0.000000 Wait at N x N
          ■ 0.174978 Short-term
          ■ 0.117152 Long-term
  ▾ □ 0.000000 OpenMP
    ▾ □ 0.000000 Wait at Barrier
        ■ 0.170414 Short-term
        □ 0.000000 Long-term
    ▾ □ 0.000000 Idleness delay costs
        ■ 79.153042 Short-term
        ■ 6.676737 Long-term
▾ □ 0.000000 MPI point-to-point wait states (propagating vs. terminal) (sec)
  ▸ ■ 0.007599 Propagating wait states
  ▸ ■ 0.038386 Terminal wait states
▾ □ 0.000000 MPI point-to-point wait states (direct vs. indirect) (sec)
  ▸ ■ 0.024267 Direct wait states
  ▸ ■ 0.021719 Indirect wait states
▸ ■ 0.999031 Critical path (sec)
0.00                    0.00 (0.00%)                    87.34
```

Figure 4.16.: Screenshot from the Cube-GUI comparing the means of the reference model(left) with the alternative model(right) for TeaLeaf.



(a) Aggregated by location (threads merged for same process).



(b) Aggregated by callpath (all paths with costs <1 omitted).

Figure 4.17.: Delay costs of both models for TeaLeaf.

has been cut down to only include values larger or equal to one. This decision was made, because values below that threshold would not be observable in the plot, as the two callpaths with the highest delay costs have much higher values exceeding the other callpaths by a factor from 10 to 100. leaving only the two kernels of the benchmark as relevant computations. The analysis of OpenMP patterns in Scalasca currently misses certain functionalities and therefore does not allow an in-depth comparison of both models for applications that suffer mainly from OpenMP wait states.

4.5. Summary

Comparing the analysis results of both model implementations yielded one general trend. In the proposed model, delay-costs tend to propagate more, indicating that for the investigated applications the waiting time present within a synchronization interval was penalized less than it should have (not all waiting time capable of propagating was actually allowed to propagate by the reference model). This also led to an even stronger consolidation of delay costs to single contributors. Also a reduction of delay costs could be observed for smaller wait states in the majority of the conducted experiments, most probably indicating wait states that previously got a higher amount of direct waiting time assigned, showing effects of the increases propagation priority. In real-world scenarios this could help to void ambiguous interpretations when multiple high delay cost wait states are present. Furthermore, no miss-classifications by the alternative delay-cost model were observed, leaving no obvious negative arguments against the alternative model. As both models showed the same trends, it can be assumed that the delay-cost model that is currently used in Scalasca is already working well for the identification of culprits and optimization potential.

5. Conclusion

Identifying imbalances and their root-causes in the communication and synchronization performed in parallel applications is the first step to optimize performance of an application. Scalasca's trace-based analyzer identifies common wait state patterns and uses the measured waiting time to expose their root-causes. The current delay-cost model used by the analyzer assigns cost to intermediate wait states and computations only proportional to their length. This thesis explored, whether an alternative delay-cost model that prioritizes cost assignment to intermediate wait states within a synchronization interval over local delays, can provide new insights into the analysis of applications that were already analyzed by Scalasca in the past. This is done, such that local delays are only penalized when the aggregated intermediate waiting time alone is less than the waiting time experienced at the end of the corresponding synchronization interval. Therefore the alternative model was defined and implemented in Scalasca. Synthetic tests were used to verify the correctness of the implementation, followed by an in-depth analysis of the results provided by the alternative model. The overall results showed a stronger consolidation of delay costs on wait states that were already the largest wait states, when analyzed by Scalasca's current model. This could provide a more unambiguous way for the user to identify the most severe root-cause for the largest optimization potential. While the alternative implementation did not uncover additional root-causes it still managed to verify the ones found by the current model, indicating that the current model already identified the main culprits even when it tends to spread costs across more individual delays. This makes the alternative delay-cost model a suitable extension. However, both models should be used together on more real-world applications to verify the findings of this thesis for a wider range of applications. If the observed effects prove to help users in the process of optimization, the proposed model could be considered to replace the model used so far, as no downsides could be observed. Until this point is reached, using both models side-by-side may help to fully identify the potential of analyzed applications. Once Scalasca supports additional wait state patterns and more types of locations like, e.g., GPUs, both models should be reevaluated to check for differences that remain hidden with the current functionalities.

A. Measurement data

A.1. Data used for calculation of the averaged reports

Run	1		2		3		4		5	
	Runtime[s]	Delay cost[s]	Runtime[s]	Delay cost[s]	Runtime[s]	Delay cost[s]	Runtime[s]	Delay cost[s]	Runtime[s]	Delay cost[s]
BT (MPI)	31333	1735	31503	1670	31291	1514	31452	2109	31361	1630
BT-MZ	19906	3134	19733	2925	19629	2814	19661	2838	19637	2818
Sweep3D	41910	7557	41686	7563	41572	7573	41684	7537	41640	7448
SMG2000	3662	369	3626	382	3609	363	3644	417	3667	413
TeaLeaf	130	111	95	78	97	80	94	78	105	87

Table A.1.: Total runtime and delay-costs for all measurement runs performed on each benchmark, that were later aggregated to provide averaged results, rounded up to integers. No differentiation between the models is nescessary, as the total delay-costs are not influenced by the models.

	Avg.[s]	Min[s]	Max[s]	Diff from Min to Avg.[s]	Diff from Min to Avg.[%]	Diff from Avg. to Max[s]	Diff from Avg. to Max[%]
BT (MPI)	1731.6	1514	2109	-217.6	-13	377.4	22
BT-MZ	2905.6	2814	3134	-91.6	-3	228.4	8
Sweep3D	7535.6	7448	7573	-87.6	-1	37.4	0.5
SMG2000	388.8	363	417	-74.2	-19	28.2	8
TeaLeaf	86.8	78	111	-8.5	-10	24.5	28

Table A.2.: Deviations of the minimum and maximum delay-cost values from the averaged reports for all performed benchmark runs. Values for min and max are rounded to integers. No differentiation between the models is nescessary, as the total delay-costs are not influenced by the models.

Bibliography

[1] Claix18 hardware. `https://help.itc.rwth-aachen.de/en/service/rhr4fjjutttf/article/e018f684c5624ae6b9bf7f0994d399f2/`. Accessed: 2023-02-14.

[2] NAS parallel benchmarks. `https://www.nas.nasa.gov/software/npb.html`. Accessed: 2023-02-14.

[3] Tealeaf. `https://uk-mac.github.io/TeaLeaf/`. Accessed: 2023-02-14.

[4] Top 500 list of supercomputers. `https://top500.org`. Accessed: 2022-11-21.

[5] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[6] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[7] S. Biersdorf, C. Bischof, K. Diethelm, D. Eschweiler, M. Gerndt, A. Knüpfer, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, et al. Score-P: A unified performance measurement system for petascale applications. In *Competence in High Performance Computing 2010*, pages 85–97. Springer, 2011.

[8] D. Böhme. *Characterizing Load and Communication Imbalance in Parallel Applications*, volume 23. Forschungszentrum Jülich, 2014.

[9] D. Böhme, M. Geimer, L. Arnold, F. Voigtlaender, and F. Wolf. Identifying the root causes of wait states in large-scale parallel applications. *ACM Transactions on Parallel Computing (TOPC)*, 3(2):1–24, 2016.

[10] P. N. Brown, R. D. Falgout, and J. E. Jones. Semicoarsening multigrid on distributed memory machines. *SIAM Journal on Scientific Computing*, 21(5):1823–1834, 2000.

[11] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne. Perfexpert: An easy-to-use performance diagnosis tool for hpc applications. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.

[12] L. Dagum and R. Menon. OpenMP: An industry standard API for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[13] E. Del Rosario, M. Currier, M. Isakov, S. Madireddy, P. Balaprakash, P. Carns, R. B. Ross, K. Harms, S. Snyder, and M. A. Kinsy. Gauge: An interactive data-driven visualization tool for HPC application I/O performance analysis. In *2020 IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW)*, pages 15–21. IEEE, 2020.

[14] D. Eschweiler, M. Wagner, M. Geimer, A. Knüpfer, W. E. Nagel, and F. Wolf. Open trace format 2: The next generation of scalable trace formats and support libraries. In *Applications, Tools and Techniques on the Road to Exascale Computing*, pages 481–490. IOS Press, 2012.

[15] M. Geimer, F. Wolf, A. Knüpfer, B. Mohr, and B. J. Wylie. A parallel trace-data interface for scalable performance analysis. In *International Workshop on Applied Parallel Computing*, pages 398–408. Springer, 2006.

[16] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurrency and computation: Practice and experience*, 22(6):702–719, 2010.

[17] M. Geimer, F. Wolf, B. J. Wylie, and B. Mohr. A scalable tool architecture for diagnosing wait states in massively parallel applications. *Parallel Computing*, 35(7):375–388, 2009.

[18] M. Gerndt, K. Fürlinger, and E. Kereku. Periscope: Advanced Techniques for Performance Analysis. In *ParCo*, pages 15–26, 2005.

[19] M. Harlacher, A. Calotoiu, J. Dennis, and F. Wolf. Analysing the scalability of climate codes using new features of scalasca. In *Proc. of the John von Neumann Institute for Computing (NIC) Symposium*, pages 343–352, 2016.

[20] M.-A. Hermanns. *Understanding the Formation of Wait States in One-Sided Communication*. PhD thesis, Dissertation, RWTH Aachen University, 2017, 2018.

[21] A. Hoisie, O. Lubeck, and H. Wasserman. Performance analysis of wavefront algorithms on very-large scale distributed systems. In *Workshop on Wide Area Networks and High Performance Computing*, pages 171–187. Springer, 2007.

[22] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel. Introducing the open trace format (OTF). In *Computational Science–ICCS 2006: 6th International Conference, Reading, UK, May 28-31, 2006. Proceedings, Part II 6*, pages 526–533. Springer, 2006.

[23] D. Luebke. CUDA: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pages 836–838. IEEE, 2008.

[24] G. Mao, D. Böhme, M.-A. Hermanns, M. Geimer, D. Lorenz, and F. Wolf. Catching idlers with ease: A lightweight wait-state profiler for MPI programs. In *Proceedings of the 21st European MPI Users' Group Meeting*, pages 103–108, 2014.

[25] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

[26] P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, volume 710. Citeseer, 1999.

[27] A. Munera, S. Royuela, G. Llort, E. Mercadal, F. Wartel, and E. Quiñones. Experiences on the characterization of parallel applications in embedded systems with extrae/paraver. In *49th International Conference on Parallel Processing-ICPP*, pages 1–11, 2020.

[28] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno. Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis. *IEEE access : practical innovations, open solutions*, 5:2747–2762, 2017.

[29] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. 1996.

[30] V. Pillet, J. Labarta, T. Cortes, and S. Girona. Paraver: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and Occam Developments*, volume 44, pages 17–31. Citeseer, 1995.

[31] J. Reinders. *VTune Performance Analyzer Essentials*, volume 9. Intel Press Santa Clara, 2005.

[32] P. Saviankou, M. Knobloch, A. Visser, and B. Mohr. Cube v4: From performance report explorer to performance analysis tool. *Procedia Computer Science*, 51:1343–1352, June 2015.

[33] S. S. Shende and A. D. Malony. The TAU parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[34] F. Wolf and B. Mohr. EARL—A programmable and extensible toolkit for analyzing event traces of message passing programs. In *International Conference on High-Performance Computing and Networking*, pages 503–512. Springer, 1999.

[35] F. Wolf and B. Mohr. EPILOG Binary Trace-Data Format (Version 1.1). Technical Report ZAM-IB-2004-06, Forschungszentrum, Zentralinstitut für Angewandte Mathematik, Jülich, 2004.

[36] B. J. Wylie, D. Böhme, B. Mohr, Z. Szebenyi, and F. Wolf. Performance analysis of Sweep3D on Blue Gene/P with the Scalasca toolset. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[37] W. Yoo, M. Koo, Y. Cao, A. Sim, P. Nugent, and K. Wu. Patha: Performance analysis tool for hpc applications. In *2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2015.

[38] K. Yoshikawa, S. Tanaka, and N. Yoshida. A 400 trillion-grid Vlasov simulation on Fugaku supercomputer: Large-scale distribution of cosmic relic neutrinos in a six-dimensional phase space. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2021.

[39] I. Zhukov and B. J. Wylie. Assessing measurement and analysis performance and scalability of scalasca 2.0. In *European Conference on Parallel Processing*, pages 627–636. Springer, 2013.