Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Design and Proto-typical Implementation of an Analysis Tool Interface for a Task-based PGAS Runtime

Patrick Franczak

Course of Study:	Informatik
Examiner:	Prof. Dr. Dirk Pflüger
Supervisor:	Dipl.-Inf. Joseph Schuchart, Dr. José Gracia
Commenced:	June 19, 2019
Completed:	December 19, 2019

Abstract

To build fast parallel applications, multiple programming models have been developed over the past years. In particular, the Partitioned Global Address Space (PGAS) model has emerged from the traditional shared memory and distributed memory models. The PGAS model offers decoupled synchronization and communication between processes. Recent approaches combine the PGAS model with task parallelism, e.g. DASH, UPC++ and X10. Tools are needed to verify the correctness and analyze the performance of parallel applications. To use such tools, an interface is needed to enable communication between the the underlying runtime and the tool itself. In this thesis, the design of a tools interface for the DASH C++ PGAS Framework is explored and evaluated. A modular plugin-based interface infrastructure is developed and implemented to connect external analysis tools. Plugins are used as a wrapper for external analysis tools, which do not offer native support for the developed tools interface. To show the functionality of the interface and the corresponding infrastructure, two analysis tools, namely Temanejo and Extrae are connected to the interface infrastructure. Finally, the interface infrastrucuture and the plugin to connect the Extrae performance analysis tool are evaluated by using both real-world and microbenchmarks to determine a possible overhead. For the infrastructure, no significant overhead can be measured with both real-world and microbenchmarks. However, with microbenchmarks, a significant overhead can be measured for the Extrae plugin. As the real-world benchmark reveals, the overhead is not relevant in practice, since no significant overhead can be measured here.

Kurzfassung

Um schnelle parallele Anwendungen zu entwickeln, wurden über die letzten Jahre verschiedene Programmiermodelle entwickelt. Besonders hervorzuheben ist hier das Partitioned Global Address Space (PGAS) Modell, welches sich aus dem Shared Memory und dem Distributed Memory Modell herauskristalisiert hat. Das PGAS-Modell entkoppelt die Synchronisation und die Kommunikation zwischen Prozessen. Neuere Ansätze kombininieren das PGAS-Modell mit Task-Parallelität, wie z.B. DASH, UPC++ und X10. Externe Tools werden benötigt, um die Korrektheit paralleler Anwendungen zu überprüfen sowie deren Performance zu analysieren. Um diese Tools zu benutzen, wird eine Schnittstelle benötigt, damit diese Tools mit der zugrundeliegenden Runtime kommunizieren können. In dieser Arbeit wird das Design einer Tool-Schittstelle für das DASH PGAS C++ Framwork entwickelt. Eine modulare Plugin-basierende Infrastruktur wird für diese Schnittstelle entwickelt, um externe Tools verbinden zu können. Diese Plugins fungieren als Zwischenschicht, um auch Tools verbinden zu können, die keine native Unterstützung für die entwickelte Tool-Schnittstelle bieten. Um die Funktionalität der entwickelten Schnittstelle zu zeigen, werden zwei verschiedene externe Tools, namentlich Temanejo und Extrae, mit der Infrastruktur der Schnittstelle verbunden. Zu guterletzt wird der Overhead der gesamten Infrastruktur sowie der Overhead des Plugins, welches benötigt wird um Extrae zu verbinden, mit Anwendungs- und Mikro-Benchmarks gemessen. Die Infrastruktur weist keinen signifikanten Overhead auf, jedoch kann ein signifikanter Overhead für das Extrae-Plugin mittels Micro-Benchmarks ermittelt werden. Jedoch zeigt der Anwendungsbenchmark, dass der gemessene Overhead nicht praxisrelevant ist, da hier kein signifikanter Overhead messbar ist.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

The development of parallel applications has become mandatory in order to use all the resources that modern hardware offers. Researchers developed multiple programming models that form the basis for builing fast parallel applications. In recent years, the Partitioned Global Address Space (PGAS) model has emerged from the traditional shared memory and distributed memory models. It combines the convenience of writing applications for shared memory systems with the scalability of distributed memory systems. The PGAS model offers decoupled synchronization and communication between processes, which offers potential to reduce communication overhead. Schuchart and Gracia [SG19] combined the PGAS model with the approach of task-based programming. This enables to mitigate the coarse-granularity of PGAS with the use of "fine-grained task synchronization across process boundaries" [SG19]. This approach has been implemented in the DASH C++ PGAS Framework.

More traditional programming models like OpenMP, XcalableMP or MPI offer tool interfaces (OMPT [EMS+14], XMPT [PTM+17], PMPI [Mes15]) to extract valuable information developers may need to write correct and fast applications. With those tools interfaces, developers are able to connect external analysis tools to verify the correctness and measure performance of their applications. Since none of those programming models cover the semantics of the DASH programming model, none of their tools interface mentioned above can be used for DASH. The goal of this thesis is to implement a tools interface for DASH that is loosely following the concepts of existing tool interfaces of more traditional programming models.

1.1 Motivation

As of today, a parallel programming approach is required in order to make full use of computational resources that modern systems offer. However, developing parallel applications is more demanding than developing sequential applications. Tools are needed to verify the correctness and to measure the performance of a parallel application. To use such tools, an interface is needed in order to enable communication between the application and the tool itself. Especially for new, hybrid programming models such as the Partitioned Global Address Space (PGAS) model combined with multiple classic approaches for parallel programming, e.g., tasking, there are no tools interfaces available.

To change this, this thesis explores and evaluates the design of a tools interface for a task-based PGAS runtime. This interface can be used to debug and analyze the performance of parallel applications. In former times, both use cases were strictly separated. Nowadays, it is a flaw when an application is not optimized for parallel systems. Hence, the performance of an application is as important as its correctness.

Already existing tools interfaces like OMPT [EMS+14], XMPT [PTM+17], PMPI [Mes15] are made for other, more traditional programming models (OpenMP, XcalableMP, MPI). They are not compatible with the DASH C++ PGAS Framework used in this thesis, since this framework

implements its own programming model called the Global Task Data Dependencies Model and is not compatible with other programming models. Although DART uses MPI as a backend, it would be possible to use a MPI tools interface with DASH. However, the obtained information would solely cover MPI function calls. Since MPI only operates as a backend, the obtained information does not contain valueable information considering the overlying DASH tasking model. In fact, all MPI function calls would be instrumented, also including internal calls from the scheduler that are not supposed to be instrumented.

For this reason, a tailor-made tools interface is designed for this framework. The tools interface should serve as a bridge between the DASH Runtime (DART) and an external analysis tool. Therefore it is possible to connect basically every tool to the interface infrastructure.

Outline

This thesis is structured as follows.

Chapter 2 - Background introduces fundamental models for parallel applications and presents the DASH C++ PGAS Framework.

Chapter 3 - Related Work puts this thesis in context with the work of other researchers in this particular topic by introducing similiar existing tools interfaces for more traditional programming models.

Chapter 4 - The Design of the DASH Tools Interface describes the design objectives of the tool interface and the resulting architecture.

Chapter 5 - Implementation gives a more detailed view on the tools interface and how the prototypical implementation is done.

Chapter 6 - Evaluation evaluates the interface with an emphasis on the resulting performance overhead.

Chapter 7 - Conclusion and Outlook concludes this thesis and gives suggestions for further work in this field.

2 Background

This chapter gives an overview over different parallel programming models and their underlying architectures used nowadays. In addition, the approach of task-based parallelization is introduced and compared with the data parallelism approach. Finally, the previously mentioned DASH C++ PGAS Framework will be presented and the underlying Global Task Dependencies Model will be explained in detail.

2.1 Parallel programming models

The most popular parallel programming models used today can be divided into three different categories. The first category consists of the shared memory models offering a global address space for all connected processes. Second, the distributed memory models differ from the shared memory by having a local address space for each process. The third category consists of the Partitioned Global Address Space (PGAS) model which is a hybrid of the first two categories. It combines the convenience of the shared memory programming model as well as the locality and performance control of the distributed memory programming model. The following subsection gives an overview of the different parallel programming models with their underlying architectures and representatives.

2.1.1 Shared memory model

The shared memory programming model is suited for shared memory architectures. In shared memory architectures, all processes are connected and have access to memory modeled as a global address space. Both communication and synchronization are handled by manipulating common variables residing in common shared memory. To implement the shared memory model, either so-called “heavy-weight” processes or “light-weight” processes can be used [Bar10]. Figure 2.1 shows a diagram of a shared memory architecture using threads.

A single “heavy-weight” process provides all resources which are needed to execute a program [Mic19], including its own virtual address space, the executable code itself, a unique process identifier, a security context, environment variables and at least one thread of execution.

A thread “is the entity within a process that can be scheduled for execution” [Mic19]. A process consists of at least one thread. All threads of a single process share its virtual address space and system resources. Additionally, each thread maintains its own thread identifier, exception handler, a scheduling priority and a set of structures to save the context of the thread until it is scheduled. POSIX threads (Pthreads) on Linux save the capabilities and the CPU affinity per thread as well [Lin19].

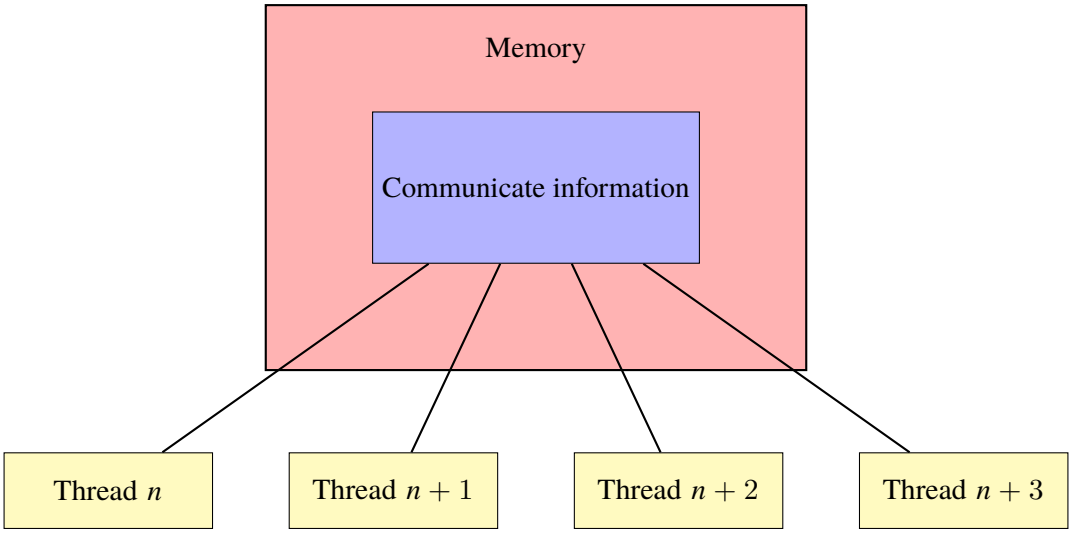

Figure 2.1: The shared memory model, used in e.g., OpenMP.

When using the shared memory model with processes, all processes share a common global address space, to which they can both read and write asynchronously. To ensure correctness for asynchronous access to common data, various well-known approaches like lock-mechanisms, semaphores or monitors can be used to implement mutual exclusion [AKH03].

The advantage of the shared memory model is its simplicity from the programmer's point of view. The model does not define a notion of data-ownership for the programmer, which means that the programmer does not have to explicitly handle the communication between processes, since all processes have equal access to the shared memory global address space [Bar10; HDT+15]. The resulting disadvantage is the increased difficulty to understand and manage data locality between multiple processes. Since data has no explicit owner, it can be difficult to keep data local to a single process in order to avoid unnecessary cache refreshes and bus traffic.

Implementation

The shared memory programming model is typically implemented using a multi-threaded programming model [Bar10]. POSIX Threads (Pthreads) and OpenMP are popular examples for this approach.

Pthreads is a C/C++ threading library [Lin19], whereas OpenMP is a directive-based shared memory API [DM98] with C/C++ and Fortran implementations are available.

In contrast to OpenMP, Pthreads requires significantly more attention to detail from the developer, since its definition of parallelism is very explicit [Bar10]. Pthreads can be used as a foundation for other implementations. The DASH C++ PGAS Framework uses Pthreads to bind a task to a thread in order to execute a task.

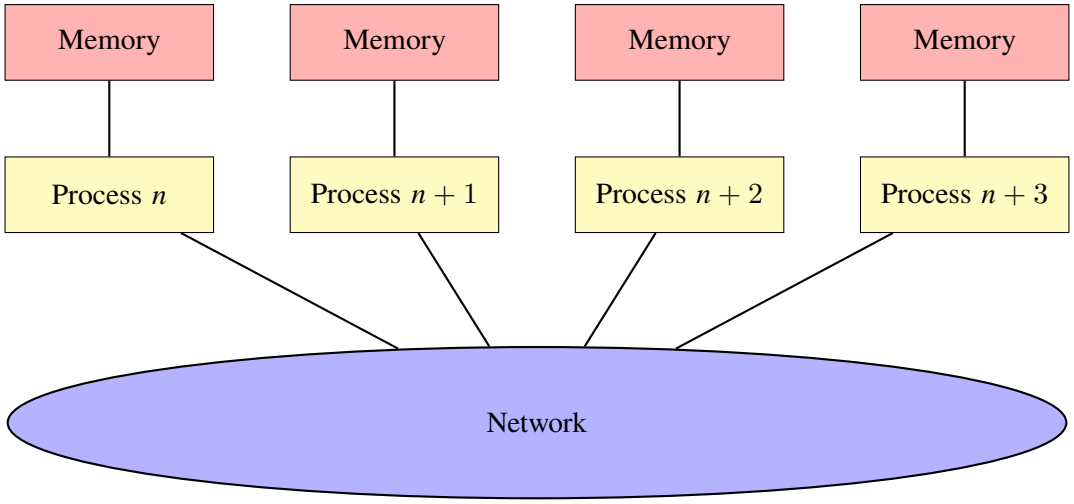

Figure 2.2: The distributed memory model, used in e.g., MPI. Figure adapted from [ST98].

2.1.2 Distributed memory model

The counterpart of the shared memory model is the distributed memory model. It is suited for distributed memory architectures, where each process has its own private local memory. Distributed memory architectures differ from shared memory architectures in the way, that processes do not communicate via common variables. Instead, processes communicate by exchanging messages over a network, to which all processes are connected. Unlike the shared memory model, processes in the distributed memory model do not have access to a global memory space. In order to do computation with multiple processes, programmers have to implement explicit communication. By sending and receiving messages, data can be transferred between multiple processes. This usually requires cooperate communication, i.e. a send operation has to match a receive operation [Bar10]. As a consequence, synchronization and data transfer are coupled.

Implementation

Since the 1980s, a variety of message passing libraries were developed, e.g. the Parallel Virtual Machine (PVM), which is compareable with the Message Passing Interface (MPI) [GL02]. MPI is the "de-facto" standard for using message passing with the distributed memory model [KLL15]. The first standard was published in 1994 [CGH94], the second version in 1996 [CGH94] and the third version in 2012 [Gro12]. The current MPI standard 3.1 was introduced in 2015 [Mes15]. MPI-2 introduced one-sided-communication, which will be explained in the following subsection.

One-sided communication

The standard MPI is based on a two-sided communication and collective communication model [Mes15]. In this model, both, the sender and receiver of a message are actively involved in the message exchange. As a consequence, the sending process has to wait for the receiving process to start receiving. By using the one-sided communication model introduced in MPI-2, only a single

process is involved in the message exchange. The underlying technique is called Remote Memory Access (RMA). Extended support by hardware acceleration is provided, if the system supports Remote Direct Memory Access (RDMA). The CPU of the remote process is not involved in the data transfer process, instead the network interface directly handles the transport of the transfered data when using RDMA [YBC+07]. With this technique, data transfer and inter-process synchronization is decoupled.

In the past couple years, research was done to combine shared memory approach with the distributed memory approach. As a result, the Partitioned Global Address Space model (PGAS) has emerged.

2.1.3 Partitioned Global Address Space (PGAS) model

The PGAS model consists of a set of processes. Each process has local memory attached, just like in the distributed memory model. The difference to the distributed memory model is that a process may share its local memory with other remote processes. Nowadays this memory sharing is implemented by either using a network device with software support or trough hardware shared memory with cache coherence [GBH18]. Nowadays, memory sharing is implemented via Remote Direct Memory Access (RDMA) [Alm11].

The PGAS model is a promising approach for programming applications running on current petascale and upcoming exascale computers [GPH+15]. It offers the programming convenience of the shared memory programming model and the locality and performance control of the distributed memory programming model. Especially when dealing with dynamic and irregular communication patterns, like in graph analytics or other data intensive fields, the PGAS approach is more suitable than the common two-sided-communication approach as used in MPI without the RMA extension [FFK16; GPH+15].

To ensure greater performance, PGAS languages rely on one-sided communication [YBC+07]. Schuchart and Gracia [SG19] point out that systems without RDMA capabilities show poor performance when using MPI-RMA operations instead.

2.2 The tasking approach

In order to exploit the power of modern computer architectures, the use of parallelism is key. Two popular forms of parallelization exist, data parallelism and task parallelism.

The use of data parallelism applies the same operation on multiple data values in parallel. A popular implementation of data parallelism on the instruction set level of modern processors are the so called Single Instruction Multiple Data (SIMD) extensions, e.g., SSE and AVX. With AVX for example, two double precision floats (64 bit) can be added per cycle. Another popular example for data parallelism is the OpenMP worksharing construct [Ope18]. Data parallelism works well on structured data structures, where such extensions can be applied.

Irregular data structures do not fit the data parallelism model. Here, task parallelism can be used. Task parallelism is an approach to to run multiple independent tasks in parallel, which leads to the definition of a task from [Cra14].

Definition 2.2.1 (Task)
"A unit of computation that can/should execute in parallel with other tasks."

A task consists of a set of instructions to be executed and a list of dependencies, which determine the execution order of the tasks.

In general, two types of dependencies exist. The first type is the *input* dependency. A task having an input dependency reads a variable. A task having an output dependency writes to a variable.

These dependencies determine at which point in time a task can be executed. Formally, dependencies describe a binary ordering between two tasks. If a task t_1 is dependent on a task t_0, t_1 can only be executed, if t_0 has successfully finished its execution.

In order to schedule tasks, i.e. to determine the order of the tasks to be executed, a scheduler has to create a task graph. The following definitions introduce the concept of cyclic graphs, which lead to the definition of a task graph.

Definition 2.2.2 (Directed graph)
A directed graph $G = (V, E)$ consists of two sets V and E, where V is a set of vertices and $E \subseteq V \times V$ is a set of tuples describing the edges.

Definition 2.2.3 (Cyclic graph)
A graph $G = (V, E)$ is cyclic if there is a path $(v_1, v_2, \ldots, v_n)$ with $v_i \in \mathbb{N}$ for $i = 1, \ldots n$ with $v_1 = v_n$.

The definition of a task graph should now be intuitive.

Definition 2.2.4 (Task graph)
A task graph is a directed acyclic graph, where the nodes represent the tasks itself and the edges represent the dependencies between two tasks. A task t is dependent on a task s, if there is an directed edge $(s, t) \in E$. If a task t has no incoming edges then this task can be executed at any time.

Figure 2.3 shows an example task graph based on the code in listing 2.1. Formally, the task graph can be described as follows.

$$V = \{t_0, t_1, t_2, t_3\} \qquad \text{with} \tag{2.1}$$
$$E = \{(t_0, t_1), (t_0, t_2), (t_1, t_3), (t_2, t_3)\}. \tag{2.2}$$

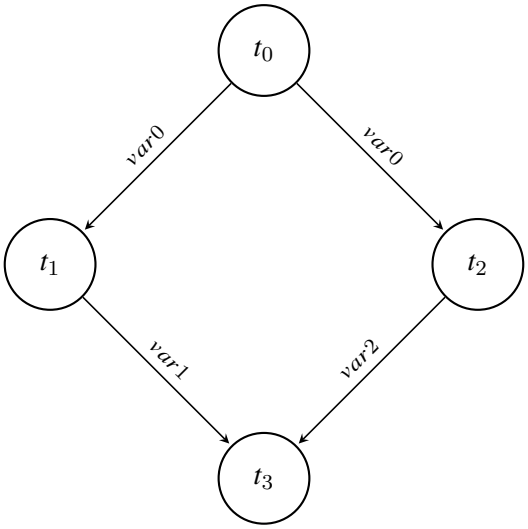

Figure 2.3: The task graph for the DASH application in Listing 2.1.

```
#include <stdio.h>
#include <libdash.h>

int main(int argc, char **argv) {
  dash::init(&argc, &argv);
  int var0 = 42;
  int var1; int var2; int var3;
  namespace dt = dash::tasks;

  dt::async("TASK0",
    [&]() { var0 = 42; printf("TASK0: var_0 = %d\n", var0); },
    dt::out(var0));

  dt::async("TASK1",
    [&]() { var1 = var0 + 10; printf("TASK1: var_1 = %d\n", var1); },
    dt::in(var0), dt::out(var1));

  dt::async("TASK2",
    [&]() { var2 = var0 - 10; printf("TASK2: var_2 = %d\n", var2); },
    dt::in(var0), dt::out(var2));

  dt::async("TASK3",
    [&]() { var3 = (var1 + var2)/2; printf("TASK3: var_3 = %d\n", var3); }
    dt::in(var1), dt::in(var2), dt::out(var3));
  dt::complete();
  dash::finalize();
  return 0;
}
```

Listing 2.1: An example of an DASH application with four tasks.

Task t_0 does not have incoming edges in the task graph. Hence, t_0 is not dependent on any other task, which means it can be executed at any time. Both, t_1 and t_2 are dependent on t_1, since they both require the updated value of $var0$. When the executing of t_0 is finished, t_1 and t_2 can be executed. If the number of resources is sufficient, they both can be executed in parallel. To execute t_3, the updated values of $var1$ and $var2$ are required. Therefore t_3 can only be executed, if and only if t_1 and t_2 have finished their execution.

2.3 The DASH PGAS Framework

DASH [FFK16] is a C++ 14 template library offering static and dynamic distributed data structures and parallel algorithms implementing a Partitioned Global Address Space (PGAS) approach. It realizes the PGAS model purely as a C++ template library, such that no additional compiler infrastructure is needed. This makes it easier to integrate DASH into already existing code bases. It also offers global-view data structures, inter-operability with the C++ standard template library (STL) algorithms and existing MPI applications.

2.3.1 DART as a part of DASH

The template library DASH is built on top of the DASH RunTime (DART). DART is a lightweight C implementation of the PGAS approach offering basic functionality for global memory allocation and global memory addressing using 128-bit pointers. It abstracts from a variety of one-sided communication backends such as GASPI and MPI-3 RMA [FFK16]. The tasking approach uses Pthreads as a backend to bind tasks to a thread in order to execute the task.

DASH follows the Single Program Multiple Data (SPMD) approach with hierarchical additions, i.e. there is a virtual root task creating all tasks. The individual participants in a DASH application are called units. These units can be grouped in teams, which form the basis for all collective synchronization, communication and memory allocation operations [FFK16]. Throughout this thesis these units will be called DASH units. DASH units are implemented as “heavy-weight” processes. Listing 2.1 shows a simple DASH application with four tasks including dependencies.

2.3.2 Global Task Data Dependencies Model

The global tasking model presented by Schuchart and Gracia [SG19] is a part of DASH. In this model, the main threads of each process execute the application’s main routine. Tasks are dynamically created by the main threads of each process and executed by a set of worker threads and the main threads itself. Hence, there are no explicit parallel regions like e.g., in OpenMP. In contrast to other tasking models, there is a virtual root task which is used as a parent task for tasks created in the main thread of execution. Tasks are able to create tasks themselves. When all dependencies for a single task have been released, it is added to the pool of runnable tasks. Dependencies are only synchronized between tasks that have the same parent task.

In the global tasking model in DASH, a task consists of an action and a set of dependency definitions. The action equals the function that shall be computed. A data dependency definition in DASH consists of a reference to the dependency itself either in local or global memory and the

associated dependency type. The global tasking model distinguishes between *input*, *output* and *copyin* dependencies. *Copyin* dependencies extend an input dependency by copying the referenced memory range into an user-provided buffer. This is especially helpful on systems, which do not support Remote Direct Memory Access (RDMA).

Creating and executing the Global Task Graph

Since no process has a global view on the task graph, individual scheduler instances exchange information on remote dependencies across process borders. These remote dependencies will be matched with either local or other remote tasks. Each process is responsible for handling dependencies referencing their own local memory.

While the main application threads of each process creates tasks, the scheduler populates the local task graph and communicates any remote dependencies to the scheduler instance which owns the referenced memory location. Tasks can be executed immediately if they are not dependent on any other task. The execution of a task is deferred, until their dependencies have been handled by the corresponding scheduler instance.

3 Related Work

This chapter builds on the foundations of the previous chapter. First, an overview of available performance analysis methods is given, which are used by analysis tools presented subsequently. Finally, existing tools interface for traditional programming models including OpenMP, XcalableMP and MPI are discussed in detail.

3.1 Performance analysis methods

This section is based on the work of Ilsche et al. [ISSH15]. Performance analysis is a process consisting of three consecutive steps. First, relevant performance information of the application during execution is obtained in the data acquisition step. After the data acquisition step, the data is stored in memory or disk in the data recording step. The last step is the data presentation step. This step defines how the obtained information is presented to the user in order to create further insight for more detailed optimization of the application.

Every step of the performance analysis process has its own terminology and techniques used, which will be presented in the following subsections.

3.1.1 Data acquisition

The first step of the performance analysis process is the collection of data. This can either be done via event-based instrumentation or sampling.

Event-based instrumentation refers to a technique modifying the source code of an application to insert further code to record data about intrinsic events. During run time, the data recording code is triggered by the measurement environment. Events can for example mark the entry and exit point of functions in order to count the number of function calls of a specific function. More sophisticated events regarding communication or I/O analysis are also possible.

Sampling in the context of performance analysis refers to a technique where an application is periodically interrupted while its state is inspected. To realize the interruption, timers like `setitimer` or overflow triggers of hardware counters, e.g, `PAPI_overflow` from PAPI are used. The most interesting part of the application state inspection are hardware counters and the call-path. The call-path provides information about all functions that are currently being executed. This information roughly corresponds to the function entry and exit point marking in the event-based instrumentation technique.

Compared with event-based instrumentation, the overhead of sampling is more predictable. The overhead of sampling mainly depends on the user-defined sampling rate, i.e. how many times per second the application is interrupted and inspected, rather than on the frequency of events. Hence,

the user can specify the trade-off between measurement accuracy and performance overhead by adjusting the sampling rate. Since sampling only obtains information at pre-defined time points, the data recording is not continuous, implying uncertainty. This is because it is unknown what happens between two subsequent time points. Hence, the recorded data from sampling can serve as a data basis for statistical analysis.

3.1.2 Data recording

The second step of the performance analysis process is the recording of gathered data. Two ways are common to save the gathered data, Logging and Summarization.

According to Ilsche et al. [ISSH15], "Logging is the most elaborate technique for recording performance data". With Logging, all information from the data acquisition step is maintained and additional timestamps are added to the data. It can be applied to both event-based instrumentation and sampling techniques. Logging creates lots of data, therefore it may cause a notable overhead due to heavy use I/O operations for writing log files to persistent media. The term tracing is often used as a synonym for logging, since the data created by logging is a trace of events.

By applying Summarization, the data from the data acquisition step is reduced by removing the temporal context of the data in order to minimize the overhead of the data recording step. Summarization of event-based instrumentation introduces recording of values like event count (e.g, how many function were called during run time), event time (e.g, what is the duration of an iteration?) or communication overhead. Summarization of sampling introduces similar values such as counting how many times a particular function was found on the call-path.

3.1.3 Data presentation

The third step of the performance analysis process is the presentation of the recorded data. The most common approaches to present performance data are either as a timeline or as a profile.

A timeline lets the user display the execution of an application over time. It does represent the relationships between previous events. This gives a view how the application runs on a particular machine. Especially idle times can easily be identified. Figure 3.2 shows a timeline drawn by Paraver. The x-axis represents the execution time whereas the y-axis represents the threads running in parallel. In order to display such a timeline, Logging has to be used as a technique for data recording, since the temporal information is mandatory for displaying a timeline.

A profile presents the performance data as a summary. In general, these profiles are grouped by "a factor such as the name of the function" [ISSH15]. Multiple types of profiles exist, such as flat or call graph ones. The GNU GCC compiler allows to create these using the GNU gprof tool. In contrast to a timeline, summarized data is sufficient to create profiles.

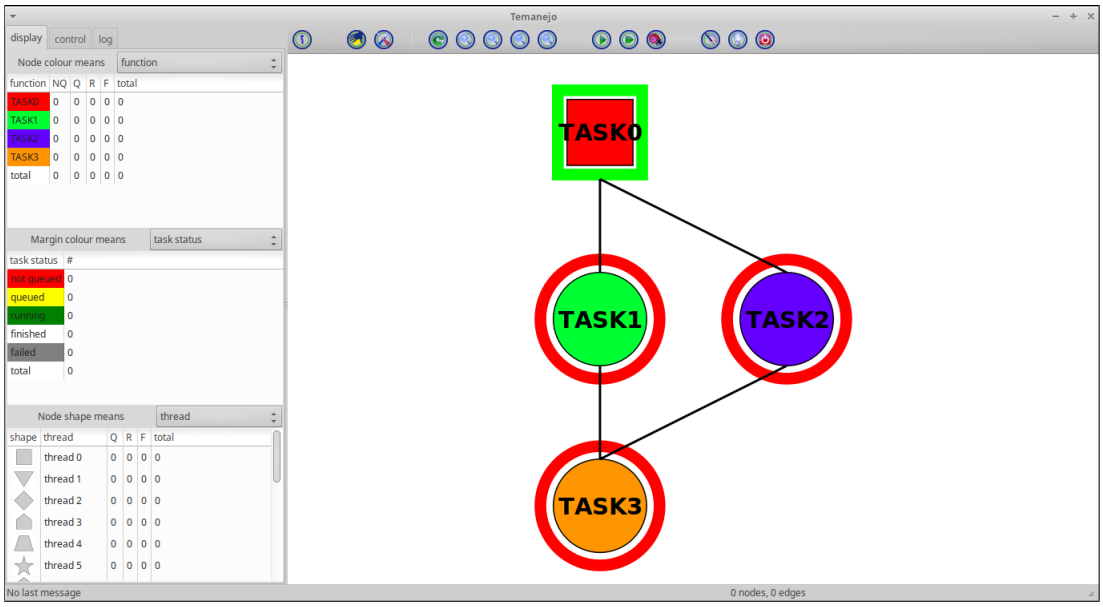

Figure 3.1: Screenshot of Temanejo debugging the DASH example shown in Listing 2.1.

3.2 External analysis tools

External tools can support users by giving insight such that the behavior of applications can be understood, evaluate and optimized later. Furthermore, performance bottlenecks can be identified and fixed. Every analysis tool gives developers a different view on the applications behavior. In this section, three different analysis tools are presented. Each analysis tool suits different purposes for application analysis.

3.2.1 Temanejo

Temanejo is a Task-Debugger developed at the HLRS by Stefan Brinkmann et al. [BGN13]. It offers a new debugging approach which acts on the task level and not on a thread level. With the lightweight ayudame library it is possible to give the user access to the task graph during run time. The library is used as a communication backend for the Temanejo front end, which visualizes the dependency graph and gives the user the possibility to control the execution of the underlying application. Additionally, Temanejo is capable of displaying the execution time of a task. However, Temanejo does not allow further performance analysis of applications and therefore does not count as a performance analysis tool.

Figure 3.1 shows a screenshot of the graphical frontend of Temanejo. The properties of the tasks are displayed on the left side of the Temanejo window. Every individual task gets its own node color, which can represent

- the function the task represents.
- the thread on which the task is running.
- the task status.

- the source through which Temanejo received information about the task.
- the distance to another marked task (in number of tasks).
- the task duration (in nanoseconds).

A task in the Temanejo task model can be in one of five states.

1. not queued
2. queued
3. running
4. finished
5. failed

The execution control of Temanejo can be used using the `Play` and `Forward` symbol buttons in the control bar above the displayed task graph. Inside the options menu, the user can determine how many steps are executed when pressing the Forward button. The default value is 10. The play button does always execute the next operation of the application, i.e. the next task.

3.2.2 Extrae and Paraver

Extrae [BSC19a] is a performance analysis tool developed by the Barcelona Supercomputing Center. Its main feature is the instrumentation of parallel applications using the shared memory model or the distributed memory model. Extrae is capable of generating traces, which are saved as files on disk, that later can be visualized and analyzed with Paraver.

Paraver [BSC19b] is a visualization and analysis tool also developed by the Barcelona Supercomputing Center, giving the user a global view on the behavior of parallel applications during run time. Figure 3.2 shows a trace analyzed with Paraver. Extrae intermediate trace files are merged with the Paraver Merger to three distinct files such that the created trace can be analyzed with Paraver.

3.2.3 Score-P

Score-P is a tool suite for profiling, event tracing and online analysis of HPC applications. It offers support for many analysis tools like Periscope, Scalasca, Vampir and Tau [KRM+12]. As explained by Knüpfer et al. [KRM+12], Score-P consists of an instrumentation framework, which allows users to insert additional code for measurement purposes into C/C++ and Fortran codes that collect performance measurement data during run time. In order to achieve this, one of many provided shared library plugins have to be linked against the application. Score-P provides libraries for serial execution, OpenMP, MPI or hybrid combinations. The collected data can be stored the Open Trace Format in Version 2 (OTF2), CUBE4 or TAU snapshot format or either be queried via an online access interface.

Unfortuantely, it was not possible to use Score-P with DASH, since Score-P only offers support for the `MPI_THREAD_FUNNELED` thread level, i.e. only one thread is able to execute MPI function calls, whereas DASH requires support for the `MPI_THREAD_MULTIPLE` thread level, i.e. multiple threads are able to execute MPI function calls.

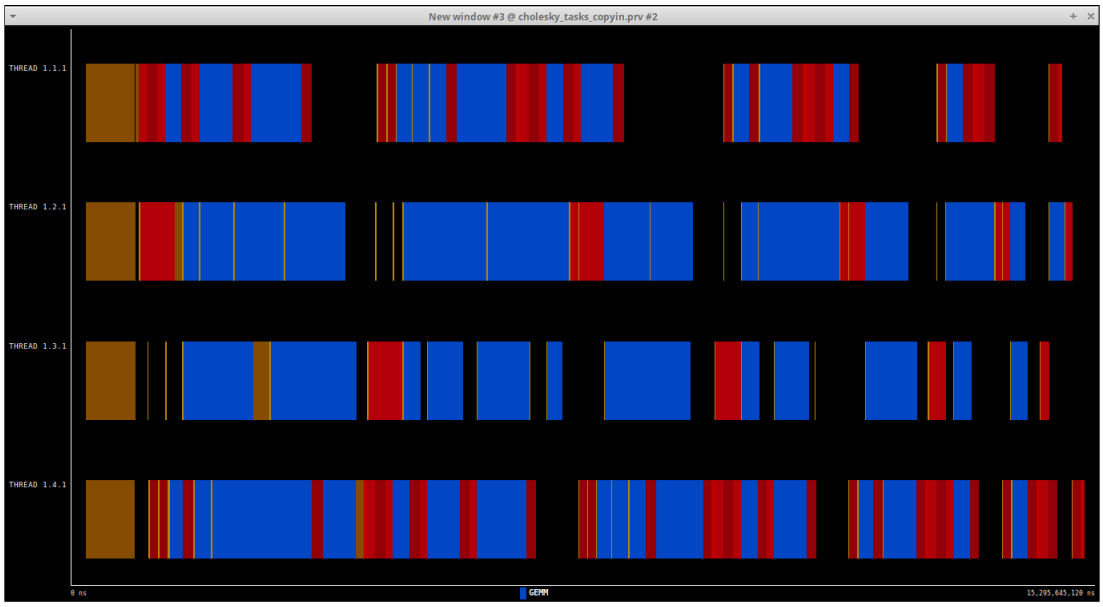

Figure 3.2: Screenshot of Paraver displaying a trace created by Extrae for a Cholesky decomposition.

3.3 Existing Instrumentation interfaces

In this section, existing tools interfaces for more traditional programming models including OpenMP, MPI and XcalableMP are presented.

3.3.1 OpenMP tools interface (OMPT)

The OMPT interface is an interface for first-party tools, thus for tools that are linked statically or dynamically to the application itself. In contrast to a third-party tool, which runs as a seperate process, a first-party tool "runs within the address space of an application process" [EMS+14]. The OMPT interface is build upon an event-based callback system to make internal performance information visible to external analysis tools.

As stated in the second technical report [EMS+14], the main design objectives of the OMPT interface were to support instrumentation-based tool approaches and sampling-based approaches. In detail, the tools interface should provide an interface for external sampling-based tools and another interface for instrumentation-based tools. Both interfaces should have a neglible performance overhead within the application and the OpenMP runtime system. Furthermore, adding the interfaces inside the OpenMP runtime and a compatible external tool should not put unreasonable burden on implementers.

Several steps are needed to extract information from the OpenMP runtime. First, the OMPT interface has to be initialized. In order to use the OMPT interface, an external tool has to implement the `omp_start_tool` function as shown below.

```
ompt_start_tool_result_t *ompt_start_tool(
  unsigned int omp_version,
  const char *runtime_version
);
```

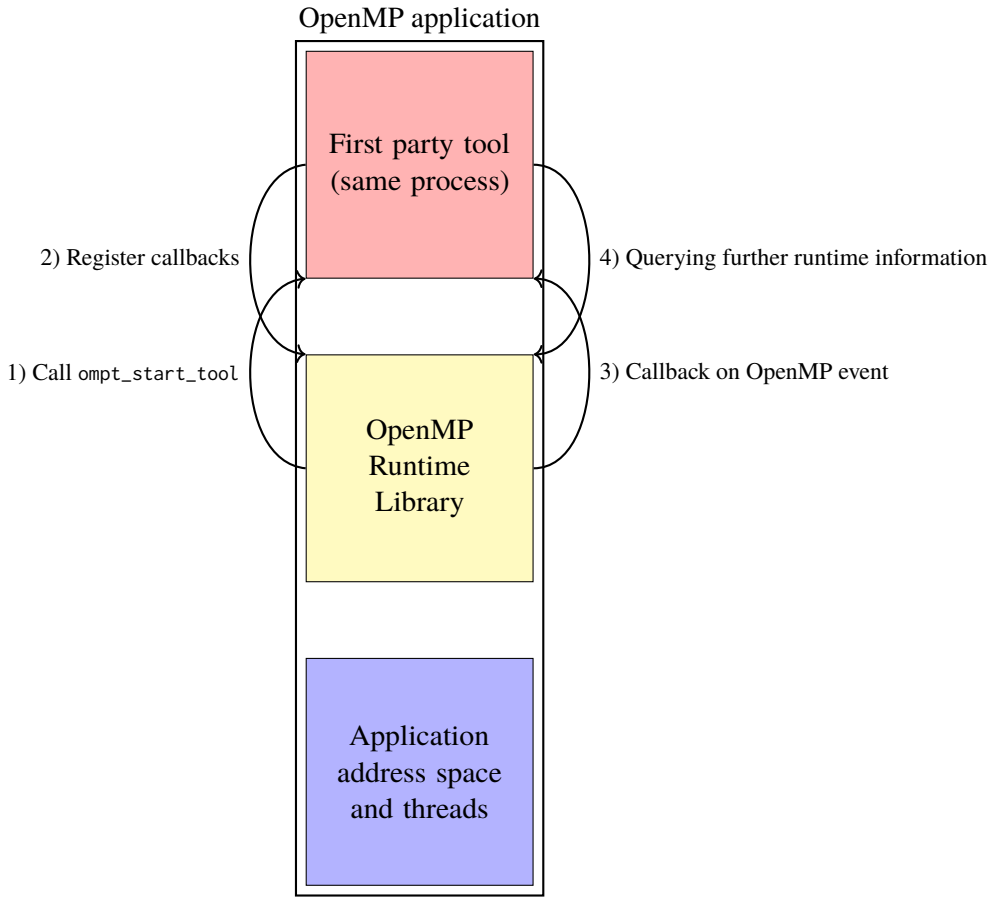

Figure 3.3: OpenMP Tools Interface (OMPT) architecture, adapted from the presentation held by Protze [Pro17].

Where `omp_version` is the value that is associated with the OpenMP API implementation used. This value identifies which version of the OpenMP API is supported by the OpenMP implementation. This also specifies which version of the OMPT interface it supports. Every OpenMP implementation provides an unique `runtime_version` argument that is used to identify which OpenMP implementation is used.

If a tool returns a non-null pointer to a `ompt_start_tool_result_t` structure the OpenMP implementation will initialize the tool interface. If a tool returns a null pointer, the interface will not be initialized. The definition of a `ompt_start_tool_result_t` structure can be found below.

```
typedef struct ompt_start_tool_result_t {
  ompt_initialize_t initialize;
  ompt_finalize_t finalize;
  ompt_data_t tool_data;
} ompt_start_tool_result_t;
```

This structure contains pointers to the initialization and finalization function inside the external tool and a pointer to the user data that can be provided additionally. Hence, an external tool has to implement an initialization and a finalization function according to the specificed signature. The initialization is invoked prior to any OpenMP event. The `ompt_initialize_t` structure is defined as shown below.

```
typedef int (*ompt_initialize_t) (
  ompt_function_lookup_t lookup,
  int initial_device_num,
  ompt_data_t *tool_data);
```

Where the `lookup` argument must provide a pointer to each runtime entry point in the OMPT interface. The `initial_device_num` argument maintains the value of the `omp_get_initial_device()` function. The `tool_data` argument is a pointer to the `tool_data` field in the `ompt_start_tool_result_t` structure returned by the `ompt_start_tool` structure.

The `ompt_finalize_t` structure is defined below.

```
typedef void (*ompt_finalize_t) (
  ompt_data_t *tool_data
);
```

Similiar to the `ompt_initialize_t` structure above, `ompt_finalize_t` contains a pointer to the `tool_data` field in the `ompt_start_tool_result_t` structure returned by the `ompt_start_tool` structure.

In order to use callbacks, the callback function has to be registered first. This is done by using the `ompt_set_callback` entry point defined below.

```
typedef ompt_set_result_t (*ompt_set_callback_t) (
  ompt_callbacks_t event,
  ompt_callback_t callback);
```

The `event` argument indicates a specific event, e.g., when a parallel region begins, for which the callback is registered. The `callback` argument is a pointer to a function inside the external tool. If the pointer is `NULL`, then callbacks associated to the event are disabled.

An excerpt of the available callbacks are shown below.

```
typedef enum ompt_callbacks_t {
  ompt_callback_thread_begin            = 1,
  ompt_callback_thread_end              = 2,
  ompt_callback_parallel_begin          = 3,
  ompt_callback_parallel_end            = 4,
  ompt_callback_task_create             = 5,
  ...
  ompt_callback_dependences             = 18,
  ompt_callback_task_dependence         = 19,
  ...
  ompt_callback_cancel                  = 30,
} ompt_callbacks_t;
```

If the event occurs, the corresponding callback function inside the external tool is called.

More details can be found in the current OpenMP standard [Ope18].

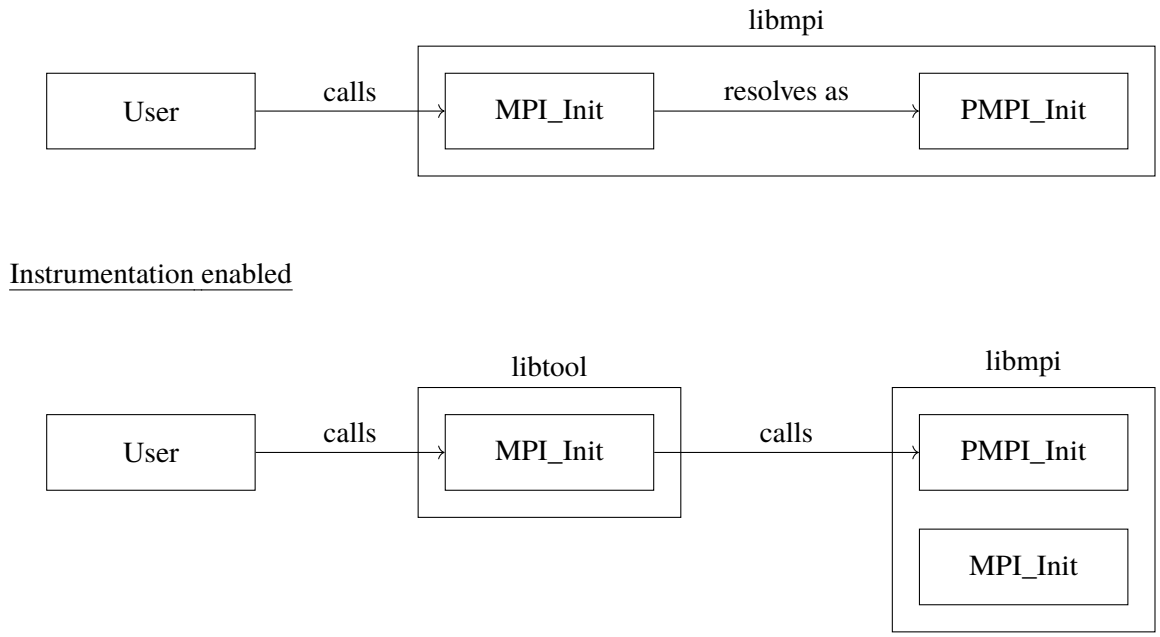

Figure 3.4: Visualization of the PMPI interface showing the disabled instrumentation above and the enabled instrumentation below. When the instrumentation is disabled, i.e. no external tool is linked between the MPI application and the MPI library, the user calls the `MPI_Init` function inside the MPI library, which resolves as `PMPI_Init` at link time. The `PMPI_Init` function contains the actual implementation of the initialization function. Below, the user has enabled instrumentation. Now, the user calls the `MPI_Init` function inside the tool. This function contains additional code for the instrumentation. Inside this function, the `PMPI_Init` function is called, which contains the actual implementation of the initialization function.

3.3.2 MPI tools interfaces (PMPI, P^{n}MPI, MPI_T, QMPI)

The current MPI standard [Mes15] offers two different tool interfaces. The first one is PMPI, which was introduced in MPI-1 in 1994 [CGH94]. MPI-3 standardized another tool interface, MPI_T. Both follow different approaches. PMPI "supports the interception and inspection of MPI calls" [Mes15], whereas the MPI_T "supports the inspection and manipulation of MPI control and performance variables" [Mes15].

Thus, PMPI can be used for instrumenting application, whereas MPI_T can be used to read and modify performance variables inside the MPI library.

PMPI The MPI profiling interface (PMPI) allows developers to replace the standard implementation of MPI routines like `MPI_Send` by own implementations using weak symbols. Weak symbols operate at link time. For each MPI function, there is a similiar PMPI function. The only difference between both functions is the prefix. A tool has to redefine the function to be intercepted, e.g., `MPI_Send` with an own definition. Figure 3.4 visualizes the concept of PMPI.

The current MPI standard [Mes15] offers an example for using PMPI, which is shown in Listing 3.1 below. In this example, a tool is able to accumulate the total time needed for the `MPI_Send` call by defining global variables and measuring the time with the `MPI_Wtime` function. The `PMPI_Send` function is called in Line 9. `PMPI_Send` is pointing to the actual implementation of the `MPI_Send` function inside the MPI library as shown in Figure 3.4.

```
1  static int totalBytes = 0;
2  static double totalTime = 0.0;
3
4  int MPI_Send(const void* buffer, int count, MPI_Datatype datatype,
5               int dest, int tag, MPI_Comm comm)
6  {
7     double tstart = MPI_Wtime();       /* Pass on all arguments */
8     int size;
9     int result    = PMPI_Send(buffer,count,datatype,dest,tag,comm);
10
11    totalTime  += MPI_Wtime() - tstart;         /* and time          */
12
13    MPI_Type_size(datatype, &size);  /* Compute size */
14    totalBytes += count*size;
15
16    return result;
17 }
```

Listing 3.1: Example of a profiler instrumenting the MPI_Send call. Example taken from [Mes15], Chapter 14.

PMPI has three major drawbacks according to [SD06].

First, by using weak symbols, only a single strong attributed function can replace a corresponding weak attributed function at link time. This means that only a single tool can be linked between the MPI application and the MPI library.

Second, if a user wants to run an application with multiple tools linked, every tool has to be relinked and the application has to be run again, which can be complex in large projects. Since multiple tools analyze different applications runs, the application has to have a "highly deterministic application execution" [SD06] to obtain comparable results.

The third disadvantage is a consequence of the first one. According to Schulz and De Supinski [SD06], many tools require similar functionality like logging, handle replacement or piggyback messaging. Since PMPI does not provide this particular functionality, the tools have to implement these features themselves. As PMPI only allows a single tool to be linked between the MPI

application and the MPI library, it is not possible to implement such features as modules which serve as independent tools. Therefore all functionality has to be integrated into a single tool, which makes the design of a tool for PMPI more complex.

P^{n}MPI To eliminate the major disadvantages, P^{n}MPI was developed by Schulz and De Supinski [SD07]. P^{n}MPI consists of three components. The first component is the stub, which is linked between the MPI application and the MPI library in order to intercept the MPI calls of the MPI application. The second component is the core component, which creates and executes the tool stacks. The last component is the configuration and loader component, which initializes the tool stacks by reading the user-defined configuration file.

To make use of P^{n}MPI, the user specifies a configuration file, defining which P^{n}MPI tools to launch. The core component then creates a tool stack of all P^{n}MPI tools the user defined. Finally, the stub component activates the tool stack by wrapping all MPI calls to at least one loaded tool on the stack.

MPI_T The MPI Tool Information interface MPI_T provides mechanisms to expose variables, which represent a particular property, a setting or a specific performance measurement, e.g., time, inside the MPI library. The interface is split in two parts. The first part provides information about control variables, whereas the second part provides access to performance variables that can provide insight into performance internals of the MPI implementation used. The interface itself can be used independently from the MPI communication, since it can be called before the call to the function `MPI_Init` occurs and it can be still used, even after `MPI_Finalize` was called. This is possible, as `MPI_T` uses its own initialization and finalization routines.

Listing 3.2 shows an example of reading the value of a control variable using the `MPI_T` interface. In order to read to the the control variable `val` with the a given index, an handle has to be allocated. This is done in Line 8. After the handle was successfully allocated, the control variable at the index bound to the handle can be read. This is done in line 14. Finally, the handle is freed in line 17.

```c
int getValue_int_comm(int index, MPI_Comm comm, int *val) {
  int err,count;
  MPI_T_cvar_handle handle;

  /* This example assumes that the variable index */
  /* can be bound to a communicator */

  err=MPI_T_cvar_handle_alloc(index,&comm,&handle,&count);
  if (err!=MPI_SUCCESS) return err;

  /* The following assumes that the variable is */
  /* represented by a single integer */

  err=MPI_T_cvar_read(handle,val);
  if (err!=MPI_SUCCESS) return err;

  err=MPI_T_cvar_handle_free(&handle);
  return err;
}
```

Listing 3.2: Example of reading the value of a control variable. Example taken from [Mes15], Chapter 14.

MPI_T is described in detail in Section 14.3 in the current MPI standard [Mes15]. The idea of `PMPI` is closer to the idea of the DASH tools interface than `MPI_T`. This is due to the fact that we want to instrument applications and not inspect or manipulate any variables inside the source code.

QMPI In September this year, Elis et al. [EYS19] presented a possible successor for PMPI, called QMPI. In contrast to PMPI, QMPI does not just a fixed name interface for the next layer of possible MPI functions to be called. Instead, QMPI adds "a set of a dynamically assigned set of function pointers cointaining the specific call information for the particular layer" [EYS19]. A layer is a part in the wrapping hierarchy, in order to enable tool nesting, i.e. the use of multiple tools at the same time. Conceptually, each tool exposes a complete MPI implementation to the next outer layer.

3.3.3 XcalableMP tools interface (XMPT)

Since the XMPT interface is not yet contained in the current XcalableMP (XMP) standard from 2018 [Xca18], there is only little information about the actual XMPT implementation. The OMPT interface serves as a model for the upcoming XMPT interface. Based on the draft presented by Protze et al. [PTM+17], the basic working principle of the XMPT does not differ from the OMPT interface.

The XMP runtime is looking for a tool at initialization time. This tool has to implement the XMPT initialization function. After that, the tool is able to register callbacks at the XMP runtime for events of interest. During the execution of the application, the XMP runtime calls the registered callback function that was coupled with an event of interest, if that particular event takes place.

4 The Design of the DASH Tools Interface

In order to use external analysis tools, an interface is required such that those tools can receive information about the internal application state. In this thesis, a concept for a tool interface for the DASH RunTime (DART) is explored and implemented. Users should gather sufficiently valuable information directly from DART such that external tools may process this information to give users insight into their applications. This chapter is structured as follows.

Section 4.1 presents requirements for external tools that will be used to communicate with an application via an interface from a user's point of view. Section 4.2 describes the design objectives, which stand as a basis for the development of this interface. The next Section 4.3 presents the design of the interface infrastructure. In Section 4.4, the main design decisions that lead to the design of the DASH tools interface are presented. After that, Section 4.5 gives an overview of all events supported by the DASH tools interface.

4.1 Requirements

Before designing an analysis tool or the corresponding interface for communication, the requirements have to be made clear. This section explores the requirements of tools and the tools interface for debugging and performance analysis from a user's point of view. The requirements are divided into functional requirements and non-functional requirements, i.e. qualitative requirements.

4.1.1 Requirements for debuggers

A debugger has to fulfill the following functional requirements.

Correctness An application should run inside the debugging environment without changes.

Interception of application flow Users should be able to set breakpoints and pause the execution. This is enabled by implementing a step-by-step execution.

Display execution information The debugger should be able to print out information about the current step of the program, e.g. which function or task is currently processed.

A debugger may implement the manipulation of variables during debugging. This serves as a qualitative requirement.

A delay between the interface used for communication between the debugger and the application is negligible, if it does not affect the user experience while debugging applications.

4.1.2 Requirements for performance analysis tools

Minimal performance impact The performance tool should distort the results by a minimum.

Measuring performance metrics For example, the run time of function or task, the idle time of particular threads or the memory consumption of a particular part of the application.

Reproducible results If the environment does not change, the results should not change.

In contrast to the debugger, performance information has to be delivered as fast as possible to the corresponding performance analysis tool.

4.2 Design objectives of the DASH tools interface

The design objectives for the DASH tool interface are similar to those presented for the OpenMP tools interface (OMPT) [EMS+13] and the MPI profiling interface (PMPI) [Mes15].

The Application Programming Interface (API) should enable external tools to obtain sufficiently valuable information about an application running on the DASH RunTime (DART).

1. The API should provide information about the task graph of the application during run time. This includes information about state changes in the task graph as well as information on local and remote dependencies.
2. An external tool using the API should receive the requested information instantly, i.e. with no delay.
3. The interface infrastructure should be modular such that external tools can be connected, even if they do not provide native support for DASH applications.
4. The performance overhead when using the API should be minimal.
5. The API should be easy to understand for external tool developers and should be extendable in the future.

The functionality of the resulting implementation of the interface and its infrastructure is demonstrated by connecting the task-debugger Temanejo presented in Section 3.2.1 to the interface infrastructure. Since Temanejo is not capable of displaying advanced performance information, Extrae, presented in Section 3.2.2, will be used to demonstrate the functionality of the interface for performance analysis.

4.3 Interface infrastructure design

The infrastructure design is crucial for the design of the interface, since it determines how information is transferred to external analysis tools. The DASH tools interface infrastructure ensures the communication between the DASH RunTime (DART) and an external analysis tool.

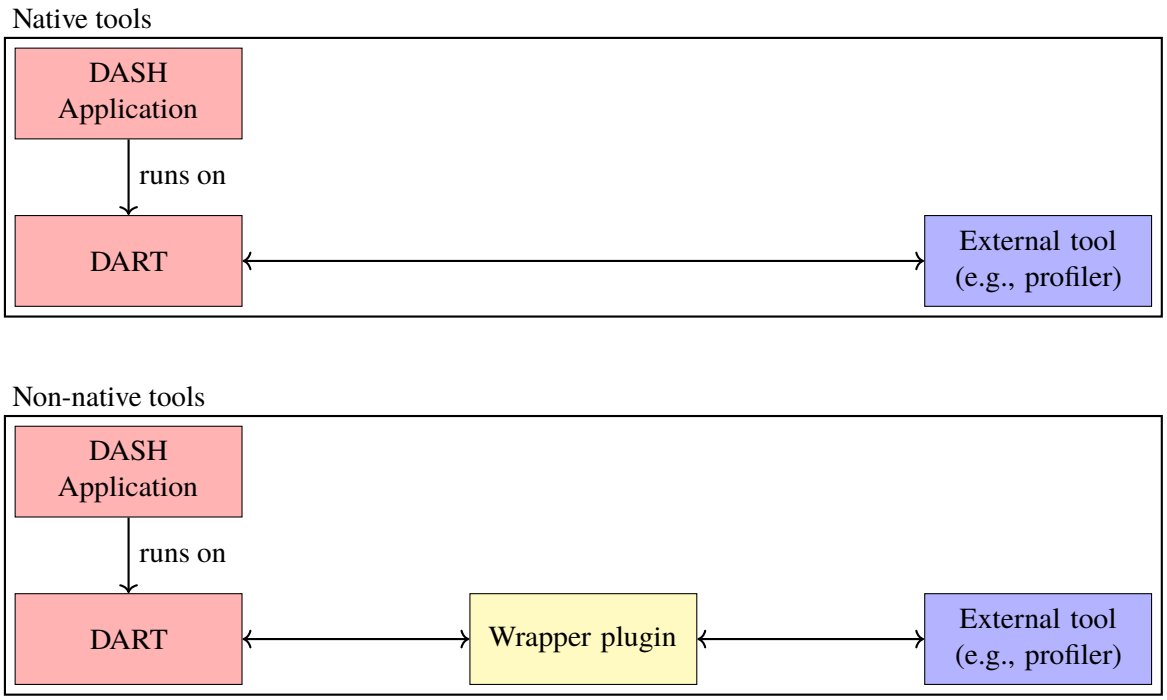

Figure 4.1: The infrastructure of the DASH tools interface allows external tools to connect to the DASH tools interface in two ways. If the external tool offers native support for DASH, it is able to connect directly to the DASH tools interface implemented inside DART, which is shown at the top. Tools without native DASH support need a tailor-made wrapper plugin that processes and forwards information from the DASH tools interface to the external tool. This is shown at the bottom.

The interface implemented inside DART is visible to external analysis tools residing in the same namespace as the DASH application. From the interface's view, each external tool is a consumer obtaining information during run time. Figure 4.1 depicts the two possibilities of using the DASH tools interface.

The first possibility is to use a tool with native support for DART. In this case, an external tool is able to connect directly to the DASH tools interface if it is residing in the same namespace as the DASH application running on DART. However, as of now, no such tool does exist.

The second possibility enables to use already existing external analysis tools, since the communication between DART and the external analysis tool is handled by a wrapper plugin. A wrapper plugin receives information from the DASH tools interface, processes and forwards the information to the external analysis tool, according to the capabilities of the tool. The external analysis tool determines which wrapper plugin is used. Metaphorically speaking, a wrapper plugin serves as a bridge between DART and the external analysis tool.

By using wrapper plugins, external analysis tools without native DASH support can be connected to the DASH tools interface. This ensures the third design objective set in Section 4.2.

Technically, all wrapper plugins are implemented as shared libraries, which makes them exchangeable. As of now, every external analysis tool needs a wrapper plugin in order to operate with the DASH tools interface. In the future, native tools may be available that do not need a wrapper plugin between DART and the tool itself.

The DASH tools interface does not distinguish between a wrapper plugin or a full external analysis tool, therefore both are consumers extracting information from the interface. Furthermore, the wrapper plugins are referenced as *shared library plugins* throughout this thesis, in order to emphasize their implementation as shared libraries.

Depending on which external analysis tool is used, the user signals DART which corresponding plugin to load by setting the environment variable `DART_TOOL_PATH` before launching a DASH application. The user sets the value of the environment variable to the path of the corresponding shared object (*.so) file of the desired plugin.

For example, if the user wants to use Extrae as the external analysis tool, the shared library plugin `libpextrae` has to be used. In order to to use this plugin, the value of the environment variable has to be set in bash like this.

```
$ export DART_TOOL_PATH=$HOME/opt/lib/libpextrae.so
```

If the environment variable `DART_TOOL_PATH` is set, DART tries to open this shared library and execute the initialization function of the plugin. The name of this initialization function is known in advance, since it is hard-coded in DART. This is done to make versioning possible. Every future version of the interface will have its own initialization function name to ensure that the loaded plugin is compatible with this particular version.

4.4 Design of the interface

The design of the DASH tools interface has to meet the objectives set in Section 4.2. This subsection explains the fundamental design decisions which led to the interface design.

Internally, DART does not save the task graph explicitly as a task-dependency graph as defined in Section 2.2. The task graph is saved in a dependency-oriented manner inside a hash map to speed up the performance of DART. In particular, every dependency holds pointers to all dependent tasks. Therefore there is no human-readable task-oriented graph maintained inside DART. To get a human-readable task-oriented graph, information on dependencies has to be gathered and transformed at multiple positions inside the DART source code, which would result in severe disadvantages. A more sophisticated approach is needed.

A more practicable solution is to extract information concerning the task graph while it is build and when parts of it change. To make this approach practicable, the information the task graph offers has to be categorized. The interface distinguishes between information on the task state, i.e. in which particular state a task currently resides and the dependencies between tasks.

Events are introduced to categorize the data that is send to a consumer by DART. An event can be described as a “significant state change” [CCC07]. Every time a task changes its state inside DART, e.g., it starts to run, a notification is created. The consumer shall immediately receive this

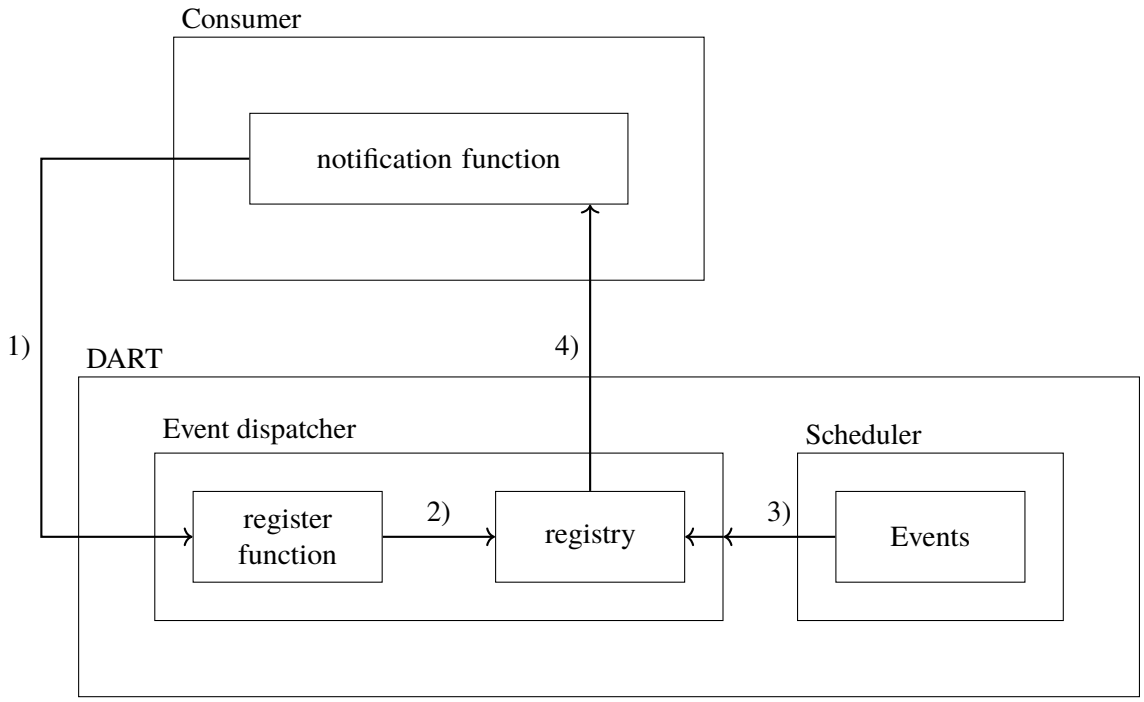

Figure 4.2: A simple diagram of the event-based callback architecture used in the DASH tools interface. First, a consumer of the interface, which may be an external tool or a shared library plugin, implements a notification for a particular event. This notification function is registered at the event dispatcher (1) and its reference is saved in a registry inside the event dispatcher (2). If this particular event occurs (3), the event dispatcher receives the event information and finally calls the previously registered notification function.

notification with no further delay such that the time of receiving the notification inside the external shared library corresponds to the time the event has taken place. The consumer shall be able to decide for which events a notification should be received.

To implement this concept, an event-based callback interface similar to the OpenMP Tools Interface (OMPT) fits these requirements the best. A simple diagram of the architecture used is depicted in Figure 4.2. This consumer has to implement a notification function, also known as callback function, for a particular event. A consumer might be an external tool or a shared library plugin. This function is called when the particular event takes place. First, this notification function registered at the event dispatcher inside DART by calling the register function for the particular event and passing the reference of the notification function. Internally, the event dispatcher saves this reference linked to the particular event. When the event takes place, the scheduler passes this information to the event dispatcher. The event dispatcher calls the notification function that was registered previously with that particular event.

When the event takes place, the event dispatcher immediately receives information about the event from the scheduler and directly calls the notification function inside the consumer with no further delay. A notification function for a particular event is only called, if and only if it is registered before the event takes place. This ensures the second design objective set in Section 4.2.

The alternative to the event-based callback interface is to use an interface based on weak symbols, like it is used in PMPI [Mes15]. This approach is not feasible for the DASH tools interface for two reasons. Due to the nature of weak symbols, only a single weak attributed function can be replaced with a single strong attributed function. This makes it impossible to attach multiple external analysis simultaneously, since this requires at least two separate functions for communication, resulting in a loss of flexibility.

Another reason for not using weak symbols is that they operate at link time. This stands in contrast to the callback approach that makes it possible to change the program flow at run time, e.g., by using environment variables. P^{n}MPI and QMPI presented in Section 3.3.2 eliminate that particular disadvantage from PMPI allowing to connect multiple tools at the same time.

In contrast to the OMPT interface, the DASH tools interface implements own register- and callback functions for every event. This is done for two reasons. First, to ensure type safety by having tailor-made function signatures for every event. Second, to improve readability of the source code by explicitly stating all parameters and the corresponding types for every event. The single disadvantage of this approach is the increased code line count inside the event dispatcher.

Similar to the OMPT interface, a consumer has to implement an initialization function. In order to provide versioning, the name of this initialization function is hard-coded, to match the capabilities of the consumer with the DASH tools interface.

4.5 Event design

The goal of the DASH tools interface is to provide information about the task graph. This includes the current state of each task including dependencies between tasks. Since the scheduler inside DART operates as a state machine, it is possible to track state changes of all tasks. This can be done by creating events. Currently, the DASH tools interface implements ten different events, of which seven track task state changes. These are explained further in the next subsection.

4.5.1 Task state changes

Every task inside DART lives trough a cycle. Figure 4.3 visualizes this task lifecycle as a state machine. The nodes represent the states, the edges represent a possible transition between states. An existing edge label refers to a particular event with the same name. For edges without a label, a corresponding event does not exist. Initially, a task does not exist in memory (`NULL`). After a task is created (`Created`), it either can be put into a task queue (`Queued`) if all dependencies are released or it can be deferred if not all dependencies are released (`Deferred`). If a task is added into the task queue, its state is set to `Queued` and it is marked as runnable and is waiting for computational resources in order to be executed. A task in the state `Deferred` is not yet ready to run, since it is waiting for dependencies to be released. When all dependencies of a particular task are released, the state of the task is set to `Running` and the task is executed. If the task ends its work without a cancellation request, its state is set to `Finished`. Otherwise its state is set to `Cancelled`. If the task is yielded during execution, its state is set to `Suspended`. In this state the task is waiting to be requeued

in order to continue its work. Similar to that, the state of a task can be set to `Blocked`, if a task is waiting for a handle to continue its execution. Tasks in the states `Cancelled` or `Finished` will be destroyed by scheduler after some time.

The focus while designing the following events was to extract data with a minimum of required computation effort on the runtime side. Therefore, variables which already exist within the runtime should directly be provided through the interface. Every task state transition has its own event if it is relevant from a users' point of view. In particular, no internal scheduling information shall be provided through the interface. For example, DART creates so-called *dummy dependencies*, if a particular task does not yet exist in order to continue building the task graph. Once this particular task exists, the dummy dependency will be matched with the actual dependency. Since this is an implementation detail of the scheduler, information about dummy dependencies are not part of the interface. The information about the `Blocked` and the `Deferred` state are also part of the implementation of the scheduler and therefore do not have a corresponding event.

task_create This event occurs every time a task is created inside DART. When a task is created, the following information is available through the interface.

- The reference of the task, which is unique for each DASH unit.
- The priority of the task.
- The name of the task.

The reference of the task will be used to identify a task locally, since it is persistent throughout the whole task lifecycle. DASH enables developers to give tasks a name, which can be used to identify the function the task computes. This information is useful to group tasks by the computed function, which can be done in e.g., Temanejo. At this point, the information on which particular thread a task runs is not yet available.

task_add_to_queue This event occurs every time a task is added to a task queue in order to be run. When a task is enqueued, it is marked as runnable. Hence, the task is waiting for resources in order to be run. All dependencies are satisfied. The following information is available through the interface.

- The reference of the task itself, which is unique for each DASH unit.
- The thread that executes the task.

The thread adding a task into a queue, which either can be a thread-private or a global queue, is not necessarily the thread that will execute the task. The time between the adding into the queue and the beginning of a task is interesting for performance analysis, as this is the time lost due to insufficient resources.

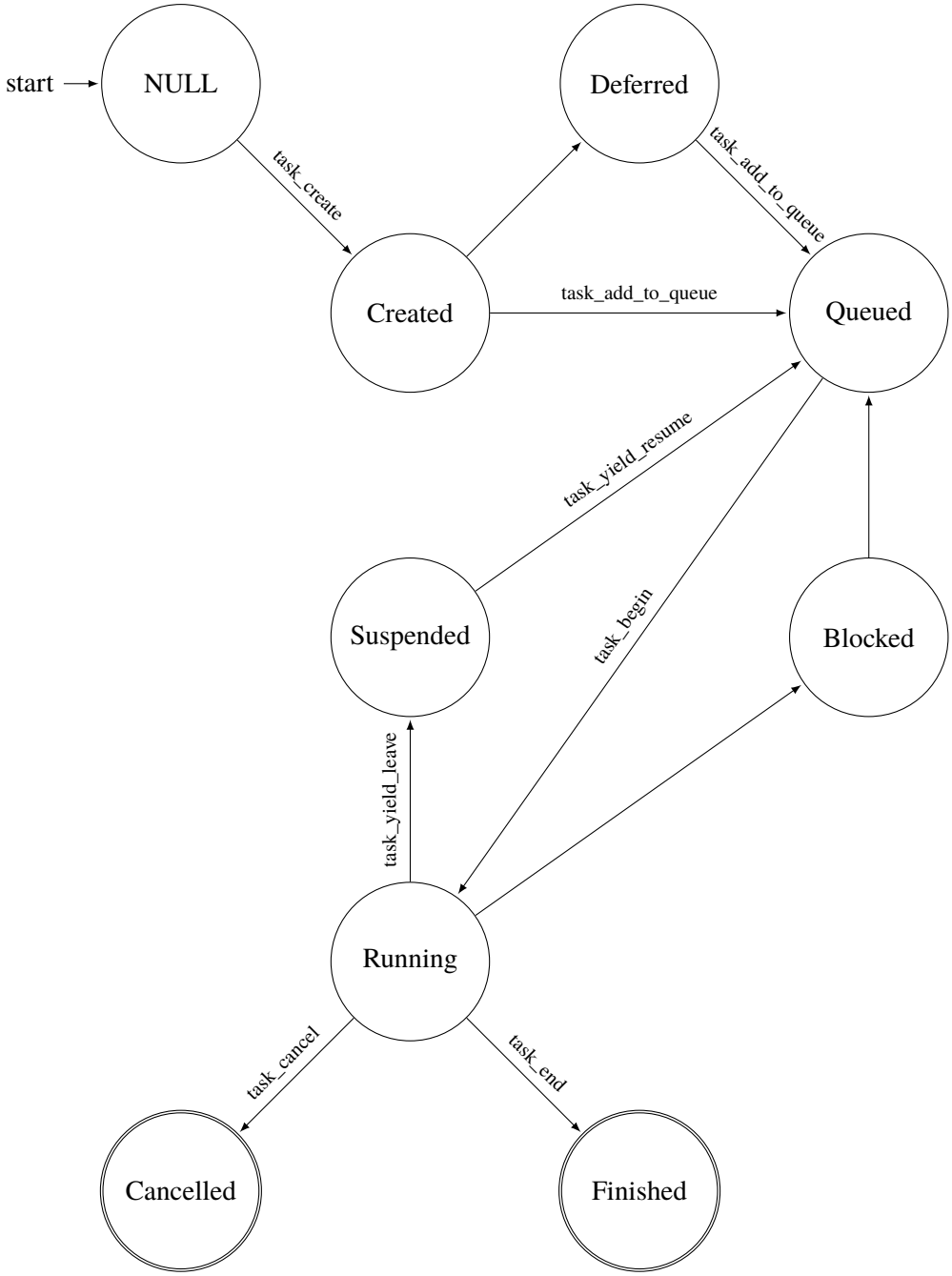

Figure 4.3: A simplified version of the task lifecycle of a task inside DART. Every node represents a state, whereas an edges represents a possible transition between two states. Labeled edges have a corresponding event available in the DASH tools interface.

task_begin This event occurs every time a task is being executed. This event marks the point in time, where the task gets executed by a thread. The following information is available through the interface, analogue to the `task_add_to_queue` event.

- The reference of the task, which is unique for each DASH unit.
- The thread that executes the task.

At this point, the thread on which the task runs is determined. The reference of a task is the same as it was when this task was created. Therefore, the reference enables the tracking of the task during the whole task life cycle.

task_end This event occurs every time a task has finished its execution. This event marks the time the task has finished its execution. The time when this event occurs subtracted the time when the task began, i.e. the event `task_begin` occurred, is the actual run time of the task. However, this is only valid if the task was not yielded. The following information is available through the interface.

- The reference of the task itself, which is unique for each DASH unit.
- The thread that executes the task.

task_cancel This event occurs every time a task is canceled. A task can be canceled by the user using commands inside the DASH application, e.g., to cancel a loop. Similar to the `task_end` event above, the `task_cancel` event can be used to determine the run time of a task with the `task_begin` event. The following information is available through the interface.

- The reference of the task itself, which is unique for each DASH unit.
- The thread that executes the task.

task_yield_leave This event occurs every time a task is yielded. If a task is yielded, its execution is interrupted and its thread executes another task. The following information is available through the interface.

- The reference of the task itself, which is unique for each DASH unit.
- The thread that executes the task.

task_yield_resume This event occurs every time a task resumes its work after it was yielded. To identify the task which resumes its work, the reference of the task itself and the reference of the thread are made available through the interface. The following information is available through the interface.

- The reference of the task itself, which is unique for each DASH unit.
- The thread that executes the task.

task_finalize This event occurs, if a DASH unit completed all scheduled tasks. This event is used to determine the end of the application. If all DASH units complete all scheduled tasks, DART will finalize the execution of the application. No further information is made available through the interface.

4.5.2 Dependencies between tasks

A dependency describes the control and the expected data flow between two tasks. Formally, a dependency can be described as a binary ordering. As explained by Schuchart and Gracia [SG19], dependencies determine the relative execution order of multiple tasks. Assume a task t_1 defines a dependency d_1 and a task t_2 defines a dependency d_2. If d_1 matches d_2, the execution of t_2 can begin if and only if t_1 has successfully finished its execution. Hence, dependencies have to be added to the task graph in order to gather information about the execution order between multiple tasks.

Since DART uses the Global Task Data Dependencies model presented in Section 2.3.2, it distinguishes between local and remote dependencies. A local dependency between two different tasks occurs if both run on the same DASH unit, whereas remote dependencies are communicated between different scheduler instances running on different DASH units. To keep the semantics of the DASH Global Task Data Dependencies Model throughout the tools interface, two distinct events for gathering dependencies were created to distinguish between local and remote dependencies.

local_dep This event occurs when a local dependency is found. A local dependency is a dependency between two tasks running on the same DASH unit. In order to define a local dependency, the following information has to be made available through the interface.

- The reference of the first task.
- The reference of the second task.

Both references are used as an identifier for both tasks. This information enables to draw an edge in the task graph. Since this is only very basic information, it would be helpful to gather more information about the actual dependency. Hence, it is possible to acquire more information about the local dependency, which is the following.

- The data hazard case, which determines whether the dependency is a Read-After-Write (RAW), Write-after-Write (WAW) or Write-After-Read (WAR) dependency.
- The referenced memory location of the dependency.

Especially the data hazard case can be a very helpful information for developers to get a feeling where which case occurs in order to optimize the code of their DASH applications. References are unique throughout run time.

remote_dep This event occurs when a remote dependency is found. A remote dependency is a dependency between two tasks running on different DASH units. In order to define a remote dependency, the following information has to be made available through the interface.

- The reference of the first task.
- The reference of the second task.
- The unit ID of the first task.
- The unit ID of the second task.

Like within local dependencies, all tasks involved in remote dependencies need to be identified globally. In order to identify tasks globally, a unique identifier is required for each task. Since references are only unique within DASH units, they cannot serve solely as an unique identifier. This is due to the implementation of DASH units.

DASH units are implemented as “heavy weight” processes. Processes have their own virtual address space. It is possible that two processes have a variable with the same virtual memory address, which does not imply that this virtual memory address points to the same physical memory address. References paired with the unit ID serve as an unique identifier, since the unit ID is unique among all DASH units. Within DASH units, references serve as a unique identifiers. Therefore both, the reference and the DASH unit ID are required to create a global unique identifier for tasks inside DART.

5 Implementation

This chapter discusses details about the actual implementation of the interface. First, Section 5.1 gives an overview over the techniques used for this particular implementation of the DASH tools interface. Next, Section 5.2 describes the initialization process of the tools interface. In Section 5.3, the supported events of the DASH tools interface are presented. After that, Section 5.4 the connection of Temanejo and Extrae along with challenges coped on the way are explained. Finally, in Section 5.5 changes to Temanejo and the Ayudame interface are proposed to make full use of the semantics used in DART and other future massive parallel processing frameworks.

5.1 Overview

All shared library plugins are written in C++ 17 and compiled with the current GNU GCC 7.3.0. To avoid name mangling, the callback functions have the `extern "C"` keyword in front of the function signature. DASH applications have to be compiled with the `rdynamic` option in order to operate with the tools interface. This option instructs the linker to add support for loading shared libraries during run time by adding all symbols and not just the used ones to the dynamic symbols table [Fre19].

In this thesis, two shared library plugins and the interface infrastructure are implemented.

5.2 Initialization process

The DASH tools interface is initialized inside the `dart__tasking__init_tools_interface` function.

The initialization of the tools interface is handled as follows. First, the value of the environment variable `DART_TOOL_PATH` is obtained via the `getenv()` function. This particular environment variable is set by the user in advance, since it determines which particular shared library plugin or external tool is loaded during run time. This environment variable contains the path to the shared object file to be opened.

If the environment variable is set, the shared library is opened using the `dlopen()` function. If the environment variable is not set or it is empty, no shared library is loaded, i.e. the tool interface will be disabled during run time. Hence, no external analysis tool can be attached.

The `dlopen()` function returns a `void*` handle. If this handle is not null, the address of the initialization function inside the consumer is obtained via the `dlsym` function. The name of the initialization function is known in advance and it is hard-coded inside DART in the variable `DART__TOOLS_TOOL_INIT_FUNCTION_NAME` in order to enable versioning.

DART passes the number of threads used, the number of DASH units and the current DASH unit ID of the instance to the initialization function of the consumer. A shared library plugin has to implement the following initialization function.

```
int init_ext_tool(int num_threads, int num_units, int32_t myguid) {}
```

`num_threads` contains the number of threads used per DASH unit. The `num_units` parameter contains the total number of DASH units, whereas `myguid` contains the global unit ID of the current DASH unit.

The initialization function returns an signed integer. After successful initialization, the initialization function of the tool must return zero.

5.3 Event implementation

The DASH tools interface provides function signatures representing every event as defined in Section 4.5. Listing A.1 shows all functions signatures for the task state change event, whereas Listing A.2 shows all functions signatures for the dependencies. A compatible consumer must implement at least one notification function with a correct signature in order to receive notifications from DART. This function will be called if the corresponding event occurs.

The `edge_type` parameter provides the information about the occurring data hazards. In particular, the value of this variable determines if a local dependency is a Read-after-Write (RAW), a Write-after-Read (WAR) or a Write-after-Write (WAW) dependency. For remote dependencies, the value `edge_type` gives insight of which particular type a dependency is. In detail, a remote dependency can either be a *copyin*, *input* or *output* dependency. Paired with the `local_dep_type` and `remote_dep_type`, the data hazard case can be obtained.

5.4 Connection of external analysis tools

In order to demonstrate the functionality of the DASH tools interface, two already existing analysis tools were connected to the interface infrastructure. First, the Temanejo task-debugger was connected to the infrastructure allowing users to step through the global task graph of a DASH application. Second, the Extrae performance analysis tool was connected to the infrastructure allowing users to create traces of DASH applications and to analyze them using Paraver.

5.4.1 Connection of Temanejo

The task-debugger Temanejo is used to display the global task graph and control the execution of an application by using step-by-step execution including the placement of breakpoints.

By connecting Temanejo to the DASH tools interface, those features can use be used to check the correctness of an application as well as debugging it if it is not running correctly.

The Ayudame shared library serves as the backend for Temanejo. Ayudame offers an interface for debugging task parallel applications. To communicate with Temanejo, Ayudame opens a TCP socket and waits for a client to connect, which in this case is Temanejo. Ayudame offers a `C` interface defined in the header file `ayu_events.h`, giving users the possibility to send data in form of events to the front-end Temanejo.

The shared library plugin `libayudame` was developed to enable the communication of Ayudame with DART. The `libayudame` plugin obtains and processes information from the DASH tools interface and sends it to the Ayudame interface.

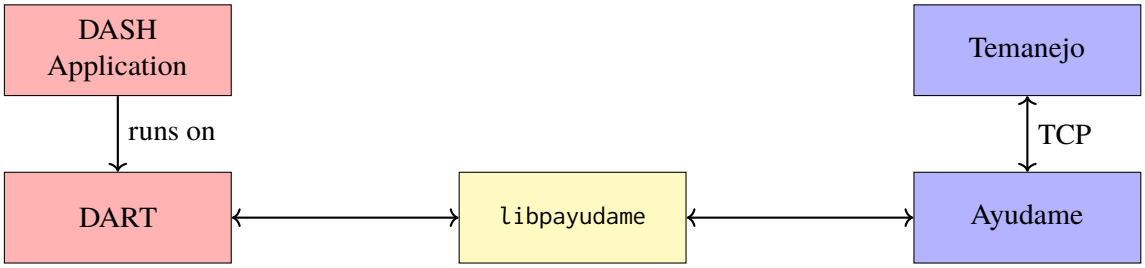

Figure 5.1: The architecture of the DASH tools interface infrastructure when the Temanejo task-debugger is used. The `libpayudame` plugin obtains information from DART through the tools interface and processes it such that the information can be sent directly to the Ayudame interface. After that, Ayudame sends the information to Temanejo, which displays the task graph. Ayudame and Temanejo communicate by using TCP sockets.

Since Temanejo was originally developed for `StarsS` task parallel applications, the task model of Temanejo does not fit exactly the Global Task Data Dependencies Model of DASH [BGN13]. Especially the lack of distinction between local and remote dependencies results in a loss of information, since local and remote dependencies are treated as generic dependencies between two tasks. This section discusses details and solutions designed solely to connect Temanejo to the tools interface infrastructure introduced in Chapter 4.

Connection to Ayudame

To enable the `libpayudame` plugin, it has to be loaded into the namespace of the application. In practice, this is done by setting the environment variable by either statically or dynamically linking the Ayudame shared library to the application.

Since Ayudame and Temanejo communicate via TCP sockets, both parties have to agree on a common port number. On startup, Temanejo and Ayudame read the environment variable `AYU_PORT`, which specifies the port used for communication. This is easy, when only a single unit of an DASH application communicates with Ayudame. In that case, the environment variable is set before the application launch. Ports for multiple instances can also be set using a comma between the individual port number. If multiple units want to connect to Ayudame, the environment variable has to be set in advance for every unit. Since is is not useful to hard-code the port number into the shared library plugin, the `AYU_PORT` environment variable needs to be exported dynamically for every unit based on unique information of every unit.

Since DASH Units are implemented as processes, the system's Process ID (PID) can be used as a port number, which is obtained by calling `getpid()` on every `libpayudame` instance. The PID is suited for the use as the port number for the following reasons.

- The PID is unique among all running processes on the system.
- The Linux kernel assigns PIDs to processes in a sequential manner.
- Typically, the maximum PID is 32768 on most systems. This can be verified by typing `cat /proc/sys/kernel/pid_max` into a terminal.

Port numbers less than 1024 are called the system ports [CET+11]. These ports may not be used without admin rights on most systems. Hence, the shared library will not use port numbers smaller than 1024.

A gather function provided by DART similar to MPI is used to fill a buffer in unit 0 with all the PIDs of each DASH unit. The `libpayudame` plugin on DASH unit 0 outputs the port numbers in a comma separated string on the `stdout`. The user copies this comma separated string and starts Temanejo by typing

```
$ Temanejo -c :12340,12341
```

into a terminal for the port numbers 12340 and 12341 in this example. By pressing the `Connect` button inside the connection dialogue, Temanejo automatically connects to the application via Ayudame. If multiple nodes on a cluster are used, the port numbers are printed on the node that executed the DASH unit 0.

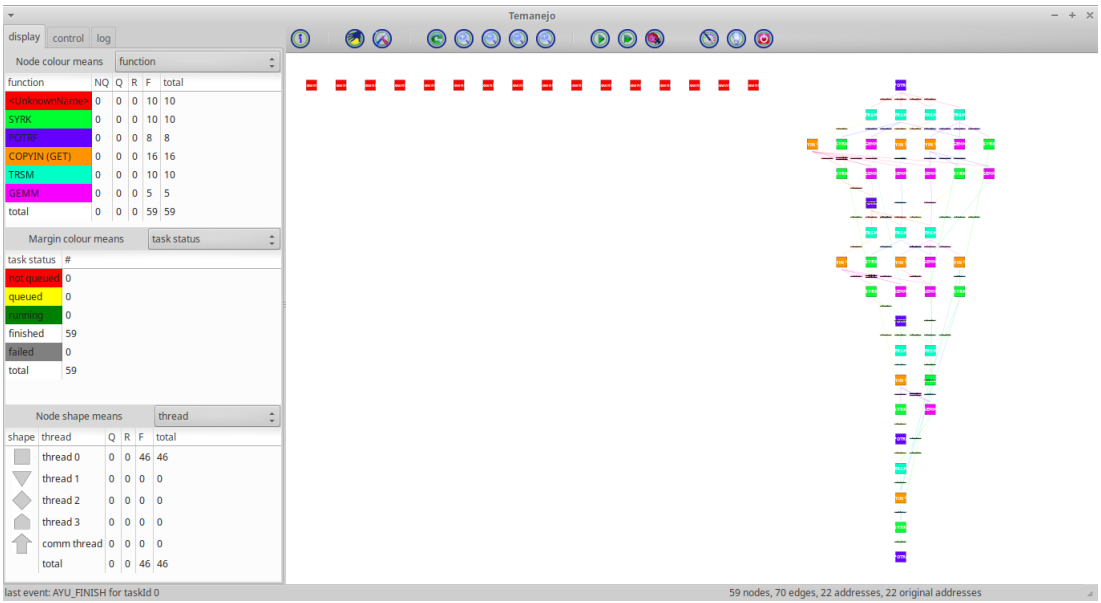

Figure 5.2: Screenshot of Temanejo debugging a Cholesky decomposition of a 200×200 matrix a block size of 20×20, resulting in 100 blocks. The application was run using 4 DASH units.

Unique identifier for tasks

The Ayudame interface requires a unique 64 bit unsigned integer to identify a task globally. Memory addresses are not feasible, since they may not be unique when running multiple instances of the same application. This is because the pointer to the task itself is a virtual memory address, which is translated to a physical memory address in the Memory Management Unit (MMU) inside the CPU.

Although current `x86-64` architectures have a register width of 64 bit, virtual memory addresses only use the lower 48 bit instead of the full 64 bit. As of today no `x86-64` architecture uses more than 48 bit for memory addressing, although the upcoming Ice Lake processor generation from Intel is said to support a technique called *5-Level Paging* [Int17] in order to support 57 bit memory addresses. This extension is already implemented in the Linux kernel [Ker19]. Even with 48 bit memory addressing, 2^{48} bytes ≈ 256 TB can be addressed.

Hence, a possible solution to have a global identifier for a task is to take the 48 bit memory address, which is cast to a 64 bit unsigned integer and add a unique identifier to the remaining upper 16 bit. The format of the 64 bit unsigned integer is shown in Figure 5.3. This gives the possibility to assign

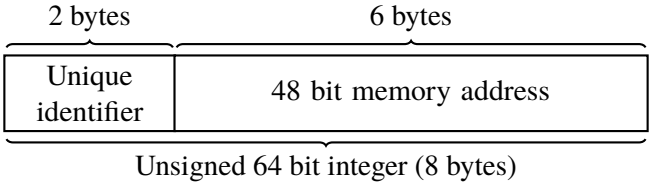

Figure 5.3: The structure of the 64 bit unsigned integer used to identify tasks globally. The first 6 bytes contain the 48 bit memory address referenced by the task, whereas the upper 2 bytes are assigned an global identifier.

each task an unique 64 bit identifier for up to $2^{16} = 65536$ instances of the application if memory addresses with 48 bit length are used.

5.4.2 Connection to Extrae

The performance analysis tool Extrae is used to create traces for DASH applications. These traces are loaded into Paraver for further analysis.

By connecting Extrae to the DASH tools interface, users are offered an in-depth view of an application running on a particular machine on a timeline such that the distribution of computational time can be understood. Especially the idle time of computing resources becomes easily viewable.

Extrae offers users to create single timestamped events. Timestamped events are used to mark the beginning and the end for every task execution, such that the user is able to view the duration of a particular task. Human-readable information can be appended for those timestamped events, which will later appear in the graphical user interface of Paraver. Events are displayed as a form of electronic signals. An event has an associated value for the time it is valid. If no event occurs,

this vale is set to zero. This makes events displayed per default as rectangles filled with color. The additional human-readable information will be used to display the name of the corresponding task when the user uses a mouse to hover it over the rectangular event area.

As a result, the connection to Extrae enables users to analyze the following coherences.

- Users are able to understand, how their application is scheduled. This includes the ability to view on which DASH unit a particular task is executed.
- Users are able to compare multiple task durations which correspond to a specific function inside the code, e.g., the duration of multiple iterations inside a loop.
- Users are directly able to see at which point in time and on which DASH unit idle time occurs.
- Users may be able to spot the consequences of dependencies between multiple tasks.

In order to use Extrae, it has to be linked dynamically to the application. To send events to Extrae, the the path to the corresponding `libpextrae` shared library plugin has to be set in the `DART_TOOL_PATH` variable in advance. This shared library plugin handles the creation and the transfer of events to Extrae.

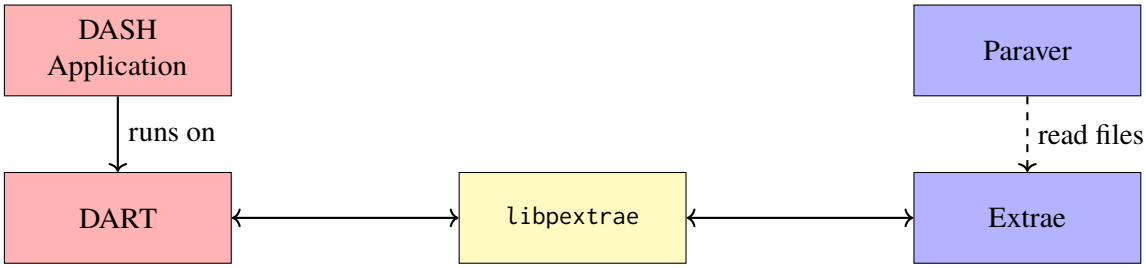

Figure 5.4: The architecture of the DASH tools interface infrastructure when the Extrae performance analysis tool is used. The `libpextrae` plugin obtains information from DART through the tools interface and processes it such that the information can be send directly to the Extrae interface. Extrae saves the resulting trace files on disk. To analyze the traces, these files are opened in Paraver. This results in asynchronous communication between Extrae and Paraver. Hence, the line connection both is dashed.

The communication between the DASH tools interface and the Extrae library is synchronous, however the resulting traces are saved in multiple files in the working directory. After Extrae has finished saving these files, the `*.prv` file can be opened in Paraver to analyze the traces. This makes the connection between Extrae and Paraver asynchronous. To support Extrae, the shared library plugin `libpextrae` was developed. Extrae is capable of writing traces, which later can be viewed and analyzed with Paraver. DASH uses MPI and Pthreads as backends, therefore the Extrae library `libptmpitrace` will be used, since it supports tracing of MPI and Pthreads simultaneously.

To create a trace with custom events, the event routines defined in `$EXTRAE_HOME/include/extrae.h` have to be used. In particular, the `Extrae_event` routine

```
void Extrae_event (extrae_type_t type, extrae_value_t value)
```

is mandatory to create traces with custom events.

The `Extrae_define_event_type` routine

```
  void Extrae_define_event_type (extrae_type_t *type, char *description, unsigned *nvalues, extrae_value_t
  ↪  *values, char **description_values)
```

is used when human readable information for custom events is necessary. Here, the `Extrae_event` routine is used to add a single custom timestamped event into the timeline. This can be used to either identify the iteration of a loop or to identify task or routines at any point in the application.

```
for (int i = 0; i < N; ++i) {
  Extrae_event(50000, i);
  /* do something */
  Extrae_event(50000, 0);
}
```

The first call of `Extrae_event` sets the type to 50000 and the value to `i` itself. To mark the end of the loop, the second call of `Extrae_event` sets the value to 0. As shown in the example, a unique value for the loop is set to identify this particular loop later in the trace.

```
void task1 (void) {
  Extrae_event(50001, 1);
  /* do something */
  Extrae_event(50001, 1);
}

void task2 (void) {
  Extrae_event(50001, 2);
  /* do something */
  Extrae_event(50001, 2);
}
```

Similarly, Extrae can be used to identify tasks or routines at any point in the application.

To give the events an expressive, human readable description, the `Extrae_define_event_type` routine is used. For the example above, the `Extrae_define_event_type` routine is called with the following parameters.

```
  static extrae_type_t extrae_t = 5001;
  static extrae_value_t extrae_values[] = {0, 1, 2};
  static char *extrae_names[] = {"NONE", "TASK1", "TASK2"};

  Extrae_define_event_type (&extrae_t, "Tasks", 3 , &extrae_values , extrae_names);
```

The shared library plugin `libpextrae` for Extrae does all this automatically.

In detail, the `libpextrae` plugin uses an `std::unordered_map` to save the task ID as an **`uint64_t`** and the task name as a `std::string` every time a task is created. A `std::string` to save the task name is used to have a copy of the name, since a **`char`**`*` may vanish over run time. As mentioned above, Extrae needs a unique value for each task or routine to identify it throughout the trace. This unique value is generated for every task by computing the hash of its name using the `std::hash` function. This hash along with the task ID is saved in another `std::unordered_map`. The hash will be displayed in the Paraver user interface along with the corresponding task names.

Every time a task is executed or resumes its work after it was yielded, the plugin checks which corresponding hash value is linked to the current task ID. This hash value is then sent to Extrae by calling the `Extrae_event` routine with the hash value as the second parameter. Every time a task has

finished, is canceled or yields the `Extrae_event` routine is called with an event value of 0. When the plugin receives the finalization notification of the current DASH unit, the unit sends its information about the events to Extrae by using the `Extrae_define_event_type` routine.

The `libpextrae` plugin is thread-safe to ensure correct results when using multiple threads. For critical sections, e.g., writing to data to an `std::unordered_map`, the `std::shared_mutex` implementation of C++17 is used. The `std::shared_mutex` is an implementation of a readers-writer-lock included in C++17 [Sta19]. According to the official C++ documentation, it offers two levels of access.

- shared - multiple threads can share the ownership of the same mutex
- exclusive - only one thread can own the mutex

This speeds up the case when multiple threads need to own a mutex solely for reading data in contrast to the classic mutex, where only a single thread is able to read or write at the same time.

5.5 Proposed changes in Temanejo

During the integration of Temanejo, semantic differences between the task model of DART and the task model of Temanejo were encountered. Both Ayudame and Temanejo were initially not designed to work with the task model used in DART, instead it was originally developed to support the StarSs and OmpSs task programming models [BGN13]. To enable Temanejo making full use of the semantics of the task model used in DASH, this thesis proposes changes in the Temanejo front end and the Ayudame interface.

5.5.1 Temanejo front-end

First, Temanejo does not distinguish between multiple types of dependencies between tasks. As a result, Temanejo displays local and remote dependencies coming from DART with the same type of line. This leads to two problems. The first problem is the loss of information, since the user cannot distinguish if a specific dependency drawn in Temanejo is a local or remote dependency. The second problem is related to the internal handling of drawing the task graph. Temanejo uses the `networkx` module available for Python 2 for graph operations. It tries to keep partitions of the graph in order to make it easier for the user to see patterns in the task graph. If an application with multiple (e.g., 4) DASH units connects to Temanejo with remote dependencies disabled, the observed graph is a disconnected graph with multiple (e.g., 4) subgraphs. In this view, the user can easily observe patterns in the task graph. If the users enables the instrumentation of remote dependencies, the graph loses its disconnectivity.

As a consequence, `networkx` draws the task graph without any observable subgraphs. This makes it hard for users to observe patterns in the task graph.

One possibility to fix this issue is to exclude remote dependencies from making changes to the graph layout. Remote dependencies may be inserted after the graph layout with its partitions is determined, such that no changes are made to the graph layout. Temanejo has to distinguish between local and remote dependencies to implement that. How this can be achieved, will be addressed in the next subsection.

Temanejo was originally not designed to work with multiple processes, i.e. DASH units, at the same time. Therefore it does not distinguish which task was received from which DASH unit, respectively port. From a user's perspective, it would be desirable to know on which DASH unit a task is running. Since every DASH unit connects through an unique port number, Temanejo does know which DASH unit is behind a particular port number. Hence, an option to distinguish various DASH units with multiple colors could be implemented easily. This issue has been fixed during the work on this thesis.

Due to the fact that Temanejo was not originally designed to operate with the Global Task Dependencies Model of DART, it only expects a single call of the `ayu_finalize` function in order to determine the end of the execution of the application. As the `libpayudame` instances operate independently, every instance calls `ayu_finalize` once when the corresponding DASH unit has finished executing its part of the application. Since Temanejo is aware of how many DASH units are connected, it should only accept the last call of `ayu_finalize` and ignore every former call of this function. This issue has been fixed during the work on this thesis.

The original version of Temanejo displayed the 64 bit long task ID inside the nodes. Since this was not visually appealing from a user's perspective, node labels now display the transmitted task name. If the developer does not set a name for a particular task, DART sets its name to <*UnknownName*>.

5.5.2 Ayudame interface

The original Ayudame interface defines function signatures for its interface in the file `ayu_events.h`. The content of this file is shown in Listing A.3. To make Ayudame aware of Massive Parallel Processing (MPP) frameworks like DASH, new functions shall be added to the Ayudame interface. First a brief introduction on the original Ayudame interface is given. The `libpextrae` plugin uses the interface functions offered by Ayudame to send information to Temanejo.

To initialize Ayudame, both `ayu_event_preinit` and `ayu_event_init` have to be called. When a task is created inside DART, the `ayu_event_registerfunction` is called to register the function the task computes at Ayudame. Here, the `func_id` parameter is the hash of the task name computed by the `std::hash` function. To ensure that a particular task is only registered once, all incoming task names are saved and compared to the current.

Every time a task is being executed, the `ayu_event_preruntask` function and the `ayu_event_runtask` function is called. Analogical, the `ayu_event_postruntask` function and the `ayu_event_removetask` are called when the task has finished its execution.

For dependencies, only the `ayu_event_adddependency` function exists. After a DASH unit has finished, the `ayu_event_finalize` function is called.

Since this interface does not match the requirements of the Global Task Data Dependencies Model that DASH uses, the following changes are proposed.

Initialization A new initialization function `ayu_event_init_mpp` is added to support the characteristics of DASH, including the use of multiple units.

```
void ayu_event_init_mpp(uint64_t unit_id, uint64_t nthreads);
```

In particular, Ayudame shall receive the unit ID through the initialization function to make it directly aware of the number of active units. Ayudame shall save the unit ID after initializing such that it has to be transferred only once.

Registration of functions and strings In Ayudame, functions have to be registered. For DASH, a function is a task, which computes a specific function. This task is registered along with its name by calling the `ayu_event_registerfunction`. The old `ayu_event_registerfunction` function is extended by an integer that contains the length of the function name, i.e. task name, since a `char*` does not contain any length information.

```
void ayu_event_registerfunction(uint64_t func_id, const char *name, uint32_t length);
```

A new function called void `ayu_event_registerstring` gives the user the possibility to register strings that may later serve as edge labels. To register a string, a user provided string ID stored as an `uint64_t`, a string stored as a `char*` and its length stored as a `uint32_t` are required.

```
void ayu_event_registerstring(uint64_t string_id, const char *string, uint32_t length);
```

Dependencies To support remote dependencies, the `ayu_event_adddependency_mpp` function is added. Since remote dependencies are dependencies across unit borders, Ayudame needs to know from and to which DASH unit the dependency occurs. Hence, two parameters `from_unit_id` and `to_unit_id` are added respectively. Local dependencies are added via the `ayu_event_adddependency` function, however the ability to pass the memory addresses of the dependencies is removed. This was useful in in the StarsS programming model, however the benefit using it with DASH is limited. The `dep_id` parameter determines the data hazard case for both local and remote dependencies.

```
void ayu_event_adddependency_mpp(uint64_t dep_id, uint64_t to_id, uint64_t from_id, uint64_t to_unit_id,
↪  uint64_t from_unit_id);
void ayu_event_adddependency(uint64_t dep_id, uint64_t to_id, uint64_t from_id);
```

6 Evaluation

The prototypical implementation presented in the previous chapter will be evaluated in this chapter. First, benchmarks are performed to measure a possible interface overhead. These overhead measurements are conducted using a disabled interface, a minimal shared library plugin, and the `libpextrae` shared library plugin. Next, the results are discussed.

6.1 Benchmark Methodology

All benchmarks were performed on a single node on the *vulcan* cluster at HLRS. The underlying hardware has the following specifications.

- Processors: 2 Intel Xeon Gold 6138 @ 2.0GHz (Codename Skylake) with 20 cores each.
- Memory: 192 GB DDR4 @ 2666MHz (Hexa-Channel)

A single node was used to minimize the communication overhead and to showcase the overhead of the DASH tools interface with minimal external performance overhead. By using multiple nodes, the inter-node communication would add more unnecessary communicated overhead to the baseline measurements. The cluster was chosen in order to retrieve reproducible and reliable results, since these systems are designed to have minimal background activities. All codes were compiled with the GNU Compiler 7.3.0 and linked against Open MPI 3.1.2, which also was compiled with the GNU Compiler 7.3.0. The DASH framework was build in *Release* mode. All threads were pinned to their respective core by DASH which is using the hwloc library. The pinning offered by Open MPI was disabled using the `--bind-to none` option. Unless stated otherwise, all benchmarks are performed using all 40 cores and 80 threads of the node.

6.1.1 Selection of benchmarks

To conduct various measurements, multiple categories of benchmarks are used. The benchmarks can be divided in two distinct categories: Microbenchmarks and real-world benchmarks.

Microbenchmarks aim at measuring only specific features of a system [SGC93]. Usually, microbenchmarks only consist of a small amount of code, whereas real-world benchmarks tend to have a much larger codebase.

Real-world benchmarks originate from applications, which are commonly used in practice. In contrary, microbenchmaks are written to solely fit the purpose of measuring specific features of the system, which makes them not represent real-world performance. To measure real-world performance in applications, microbenchmarks are not optimal. Bershad et al. [BDF92] mention cache effects that may affect the quality of the results when using microbenchmarks.

Taskbench The first microbenchmark, referred as `taskbench`, creates a fixed number of empty tasks n, of which $1 \leq m \leq n$ tasks belong to the same class of tasks. An empty task does not contain any instructions. Both n and m are passed as parameters to the benchmark application. A task class only contains tasks with the same assigned task name.

For example, the following call of `taskbench`

```
./taskbench 10000 10
```

creates 10000 empty tasks with 10 unique task classes.

The elapsed time is measured internally by the benchmark application itself. Only the part of the task creation is measured by taking a time stamp directly before and directly after the benchmark routine call. Afterwards, the elapsed time is calculated by subtracting the time before the call from the time after the call.

Depbench The second microbenchmark, referred as `depbench`, creates a fixed number of empty tasks n with $m = 1$, i.e. there is only a single task class. In addition to that, every task is assigned d local dependencies. For example, the following call of `depbench`

```
./depbench 10000 16
```

creates 10000 empty tasks of which every task has 16 local dependencies. Analogous to the `taskbench`, only the creation of the tasks with their corresponding dependencies is measured.

Cholesky decomposition benchmark The real-world performance is represented by a DASH application performing a Cholesky decomposition of a square matrix. The benchmark measures the time it takes to compute this decomposition as well as the Floating Point Operations per Second (FLOPS). The benchmark offers to specify the dimension of the matrix and the block size as command line parameters. All measurements of this benchmark were conducted using a 20000×20000 matrix with blocks of size 500×500 resulting in 40×40 blocks, if not stated otherwise. The Intel Math Kernel library was used in version 2019.2. The benchmark creates 11480 tasks in this configuration to compute the decomposition. This is sufficient to keep all threads busy.

If a CPU core on modern CPUs is idling, its clock speed is lower than under load. When load occurs, the cores need a small amount of time to increase their clock speed. Therefore a warm-up phase is placed before the actual benchmark, in order to minimize the effect of the CPU governor on the CPU clock speed. Here, the warm-up phase consists of creating $n = 10^4$ empty tasks of which all belong to a single task class.

All benchmarks are repeated 50 times, if not stated otherwise.

6.1.2 Zero-tool-test

Since one of the requirements presented in Section 4.2 is to minimize the overhead of the interface, it needs to be verified if the overhead actually is minimal. Various benchmarks are conducted to measure a possible overhead.

Setup

The shared library plugin `libpdummy.so` is used to solely measure the overhead caused by the plugin infrastructure. This plugin offers empty notification functions for all available events the DASH tools interface provides, which means that all notification functions contain zero instructions. Hence, once a notification function is called it returns immediately.

The initialization function of the plugin saves the global unit ID into a variable and registers all notification functions at the event dispatcher inside DART. This shared library plugin provides a minimal version of a plugin in order to use all events of the DASH tools interface, however no external tool can be attached.

The `taskbench` and the `depbench` are run in multiple configurations, in order to measure a difference under a higher artificial load compares to less artificial load. The `taskbench` was performed with $n \in \{10^5, 10^6, 10^7\}$ tasks with $m = 1$ classes. The `depbench` was performed with a fixed number of tasks $n = 10^5$ and a various number of dependencies $d \in \{0, 2, 4, 8, 16\}$. All benchmark runs were once conducted with a disabled interface and once conducted with the `libpdummy.so` shared library plugin loaded. The benchmark duration was measured for every benchmark run. All benchmark runs were conducted using only a single thread in order to remove the communication overhead between threads.

Results

The results of the three runs of the `taskbench` are depicted in Table 6.1. The average duration to create $n = 10^5$ tasks with disabled interface amounts to 0.188s with a standard deviation of $\sigma = 0.002$. Compared to the average duration to create the same number of tasks with a minimal plugin attached (0.188s with $\sigma = 0.0006$) the duration is equal within the measurement accuracy. Therefore, no statistical significant overhead can be obtained for the $n = 10^5$ run. By having more tasks to create, the number of function calls to the notification functions inside the shared library plugin inreases. This should cause a higher overhead, if present. However, for $n = 10^6$ and $n = 10^7$, the duration also only differs within the measurement accuracy. Therefore, no statistical significant overhead can be obtained for all three runs conducted with the `taskbench`.

The results of the five runs of the `depbench` are depicted in Figure 6.1. The x-axis shows the duration of the `depbench` in seconds, whereas the y-axis shows the number of local dependencies per task. The blue bars show the duration when the tools interface is disabled, whereas the red bars show the duration when the `libpdummy` plugin is used. However, no significant interface overhead can be obtained in all four benchmark runs, since the variance of the results is still within the measurement inaccuracy.

The results of the Cholesky decomposition benchmark are depicted in Figure 6.2. Additionally, in this figure the `libpextrae` is also measured, however this particular benchmark run will be discussed in the next subsection.

With a disabled tools interface, the decomposition took 2.761s ($\sigma = 0.02$) on average, whereas the decomposition with the `libpdummy` plugin took 2.763s ($\sigma = 0.03$). The measured FLOPs show the same behavior. With a disabled tools interface, 965 GFLOPs ($\sigma = 7$) were measured, whereas 964 GFLOPs ($\sigma = 10$) were measured with the `libpdummy` plugin. The standard deviation σ is

Interface state	$n = 10^5$		$n = 10^6$		$n = 10^7$	
Disabled	0.188	($\sigma = 0.002$)	1.766	($\sigma = 0.01$)	17.593	($\sigma = 0.04$)
Enabled	0.188	($\sigma = 0.0006$)	1.768	($\sigma = 0.00$)	17.634	($\sigma = 0.09$)

Table 6.1: The run time of the `taskbench` in seconds for creating n tasks with a single task class ($m = 1$) for a disabled tool interface and a minimal shared library plugin (`libpdummy`) to enable the interface.

higher in both cases, however both averages differ within the measurement accuracy, therefore no significant overhead can be obtained. However, counting the number of instructions used for the `taskbench` for $n = 10^6$, the `libpdummy` plugin requires $\approx 1.3\%$ more instructions to complete the benchmark. For a run of the Cholesky decomposition of a 20000×20000 matrix with a block size of 500×500, only $\approx 0.03\%$ more instructions are required to complete the benchmark. Hence, the overhead caused by the interface infrastructure is minimal.

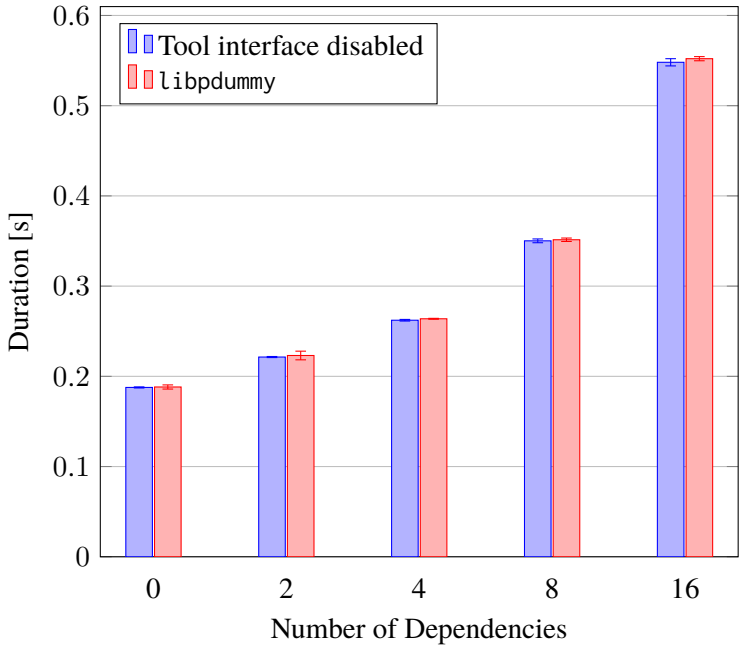

Figure 6.1: The duration of the `depbench` for creating $n = 10^5$ tasks with an increasing number of dependencies. The blue bars show the duration when the tool interface is disabled. The red bars show the duration when the `libpdummy` plugin is used. Only a single thread was used for this benchmark.

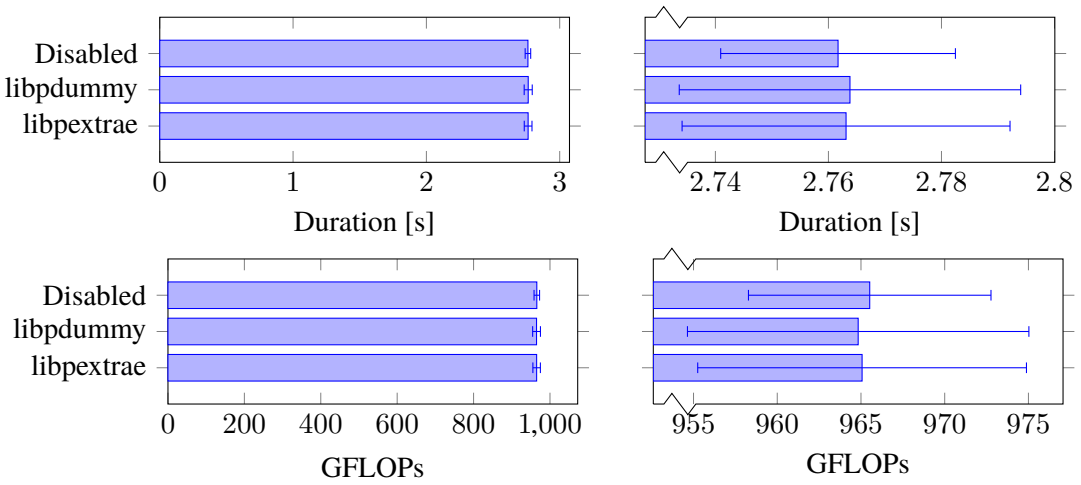

Figure 6.2: Showing the duration and the GFLOPs for the Cholesky decomposition of a 20000×20000 matrix using a 500×500 block size. Both plots on the right give further insight, since the differences between the measurements are small.

6.1.3 Extrae

To solely evaluate the overhead caused by the `libpextrae` shared library plugin without the internal overhead of Extrae and the overhead caused by saving the gathered traces on disk, the calls to Extrae (`Extrae_event` and `Extrae_define_event_type`) are commented out inside the code. Apart from that, the code remains identical.

Setup

The `taskbench` revealed that the overhead caused by the former `libpextrae` plugin is not satisfying. With the definition of the DASH tools interface as it is, it is not be possible to decrease the overhead of the `libpextrae` plugin without extensive optimization of the hash maps used by implementing custom hash maps.

To achieve a better performance, the definition of the interface is slightly changed. For testing purposes, the *task_begin* and *task_end* events include a parameter for the name of a task, just like the *task_create* event already had included before. This enables a more simpler architecture of the `libpextrae` plugin. Every time a task is created, its name and the hash of its name is stored inside a `std::unordered_map`. This is required to hand over two arrays to Extrae, of which one contains the name of the tasks and the other one contains their corresponding values. When a task is being executed, the hash of the task name is computed and send directly to Extrae with an `Extrae_event` routine call. This is implemented in the *alternative* `libpextrae` plugin.

Inside the original `libpexatre` plugin, without the change of the interface definition, a second `std::unordered_map` saves the mapping between task ID and the hash of the task name. Every time a task was created, it is necessary to lookup the hash for the current task ID. Inside the alternative `libpexatre` plugin, with the slightly change of the interface, it is possible to compute the hash of

the task name every time a task is being executed. The computation of a hash mainly consists of arithmetic operations, whereas the use of an `std::unordered_map` mainly consists of creating and copying objects in memory with additional hash computation.

Impact of increasing the number of task classes

In theory, it is computationally more expensive if more unique task classes are created when using the `libpextrae` shared library plugin, since it requires insert operations to the `std::unordered_map` and additional hash computations. In fact, every new occurring task class requires a single insert operation with thread synchronization. The alternative `libpextrae` plugin reduces the number of `std::unordered_map` from 2 to 1. However it requires more hashes to be computed. In fact, a hash is always computed when a task is executed.

To measure the overhead of having a large number of task classes, the duration of the `taskbench` was measured with an increasing number of task classes m for $n = 10^6$ tasks created.

Results The results are depicted in Figure 6.3. The x-axis shows the number of task classes starting with $m = 10^0 = 1$ with every step increasing the number by multiplying the previous value with 10. The y-axis shows the duration of the `taskbench` in seconds. The plot indicates how much time is required to create 10^6 tasks with an incremented number of task classes. The blue line shows the time required if the tool interface is disabled, i.e. no instrumentation is done during run time. The brown line shows the time required if the alternative version of the `libpextrae` plugin with a slightly change of the interface definition is used. Finally, the red line shows the time required for the `libpextrae` plugin with no further interface changes.

As expected, the number of task classes has no impact on the duration when the interface is disabled, since DART does no computationally expensive processing associated to the task name.

As expected, both libraries show a higher benchmark duration when the number of task classes is incremented, since the computation of the hash for every task class is computationally expensive. However, the alternative version of the library requires overall significantly less time to finish the benchmark compared to the old version of the library. In detail, for $m < 10^5$, the alternative version is ≈ 2.6 times faster than the old library. For $m = 10^6$ it is still ≈ 2.1 times faster. The reason for the duration decrease is the omission of one `std::unordered_map` container in the alternative version that saves the relation between the task ID and the hash of the task name. As a consequence, the writer-lock is only established if a new task class occurs. Otherwise it is not needed, since no write operations are made to global data structures.

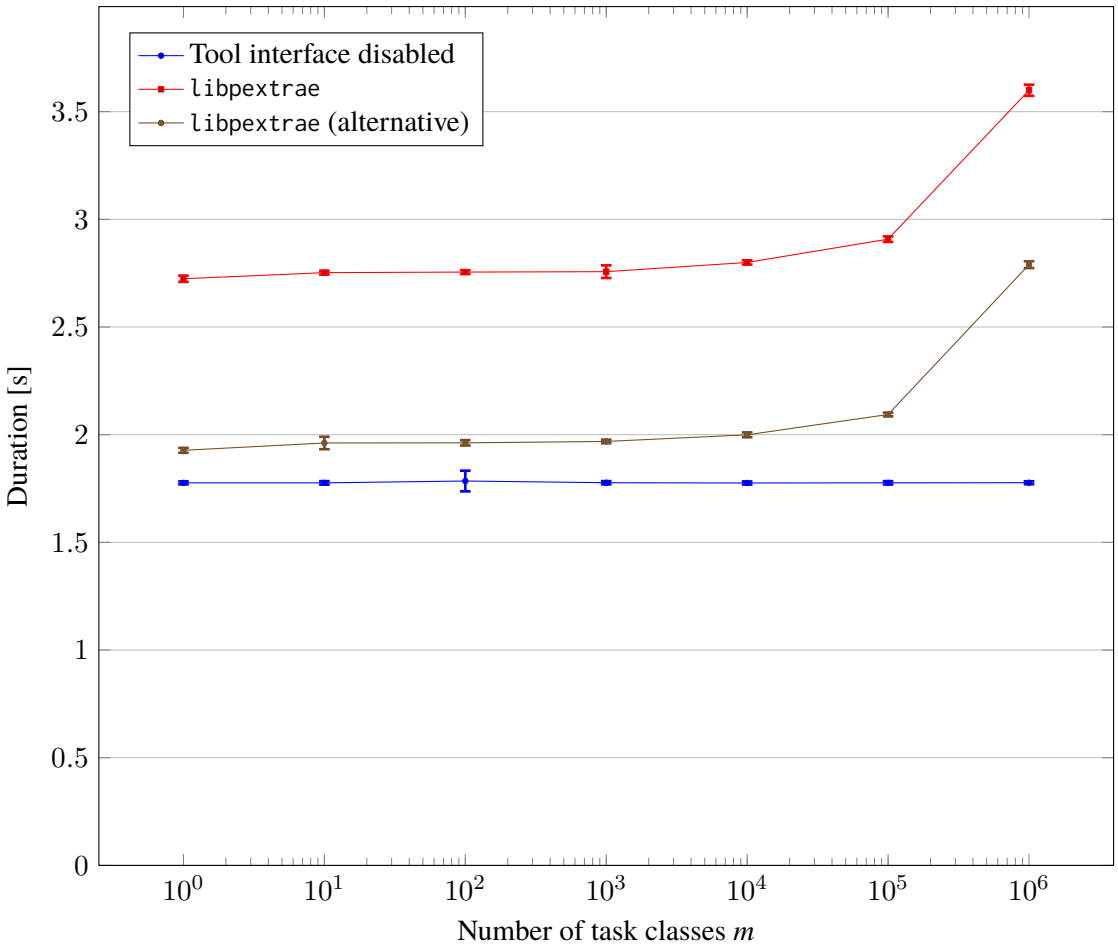

Figure 6.3: The duration of the `taskbench` for $n = 10^6$ for a disabled interface and both `libpextrae` plugin versions on the y-axis for an incremented number of task classes m on the x-axis. If the tool interface is disabled, i.e. no instrumentation is done, the number of task classes does not have an impact on the duration of the benchmark. Both `libpextrae` plugins require significantly more time to finish the benchmark. However the alternative version requires significantly less time then the standard version.

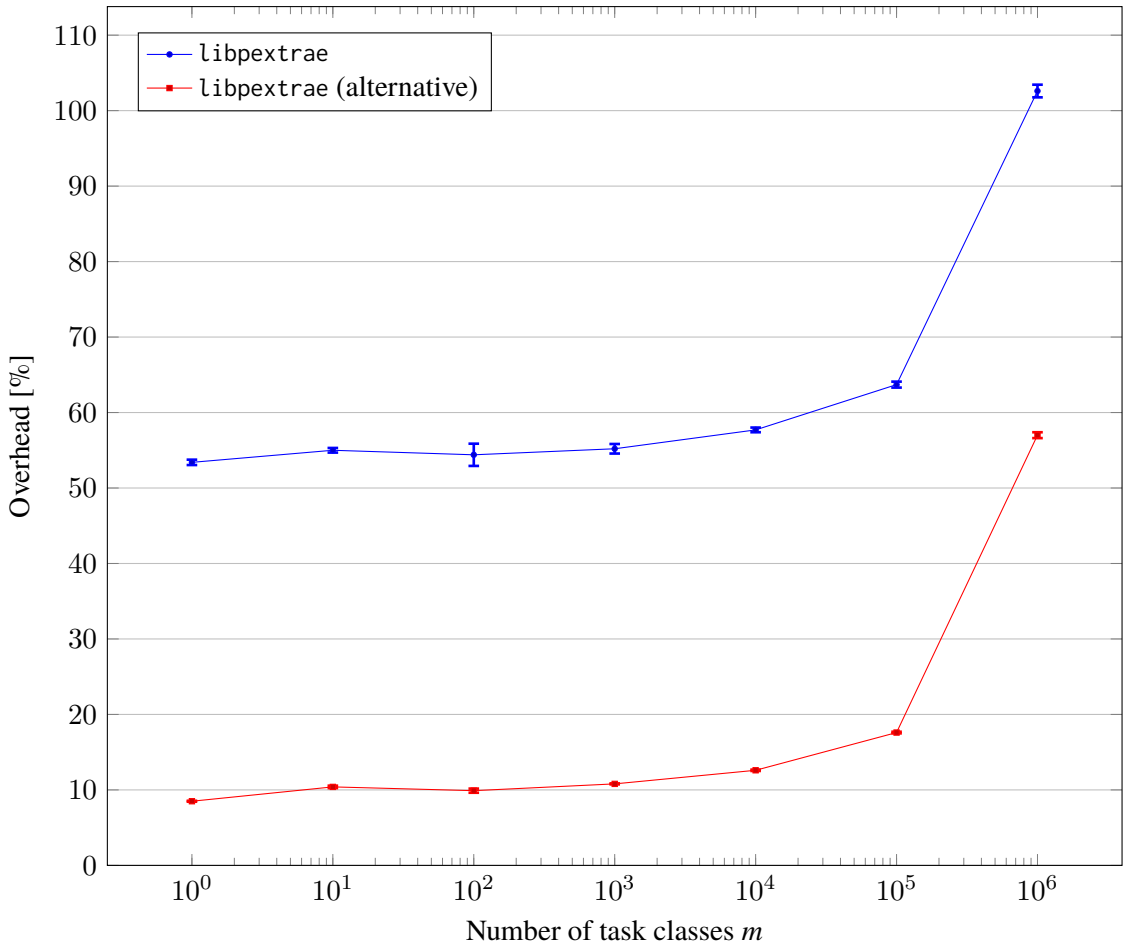

Figure 6.4: The average overhead of both `libpextrae` versions in percent compared to the disabled tools interface shown in Figure 6.3.

Impact of increasing thread count

To gather insight into the scalability of the `libpextrae` plugin, the duration of the `taskbench` for $n = 10^6$ was measured with an increasing number of threads per run. The number of threads was increased by the power of two in every step. In addition to that, the number of task classes m was varied twice. The best case is represented by setting $m = 1$, whereas $m = 10^6$ represents the worst case in terms of performance. At $m = 10^6$, every task has a different task class, which is more computationally expensive, compared to $m = 1$.

Results The results are depicted in Figure 6.5. The x-axis shows the number of threads the benchmark was running on, whereas the y-axis shows the duration of the `taskbench` in seconds. It can be observed that all six curves have a parabolic form. First, only the blue and red dashed curves are considered. Both represent the case when the tools interface is disabled. Hence, both curves represent the baseline.

As expected, both curves have an identical path within the measurement accuracy. This is due DART does not have computationally expensive processing associated to the task name. Hence, the number of task classes is not relevant for the `taskbench` duration.

The duration of the `taskbench` decreases when using more than 2 threads up to 8 threads. With 16 threads, the duration increases by $\approx 10\%$ compared to 8 threads. By using more threads, the slope of both curves gets steeper, resulting in a $\approx 75\%$ (32 threads) and a $\approx 262\%$ (64 threads) increase of duration compared to 8 threads. The `taskbench` has the shortest duration when using 8 threads.

By investigating this phenomenon with the Linux performance analysis tool *perf*, multiple possible reasons were found. First the function inside DART responsible for taking the tasks from the queue has an overhead of $\approx 3\%$ when using 8 threads. This function checks if the task queue is empty, otherwise a mutex is locked and the task from the top of the queue is popped. After that, the mutex is released. With 16 threads, the overhead caused by this function increases ($\approx 20\%$), increasing further for 32 ($\approx 70\%$) and 64 threads ($\approx 79\%$) and causing congestion in DART. In particular, when using 64 threads, this function requires $\approx 79\%$ of the computation time.

To give possible explanations for this increase in synchronization overhead, the underlying hardware has to be looked closer at. A single Skylake node of the *vulcan* cluster, consists of of a dual-socket mainboard with two 20 core CPUs installed. Since SMT is enabled, in total 80 threads are available. The threads are pinned in a way, such that the second CPU is only used, when an application uses more than 20 threads. Similarly, the threads offered by the SMT are used only if an application used more than 40 threads. Assuming this, using more than 20 threads results in additional overhead caused by inter-CPU communication. When using more than 40 threads, CPU internal resources of a single core are shared between two threads on this particular system. Hence, two threads may have to resolve additional data conflicts caused by SMT. Additionally, the number of tasks created by every thread decreases with an increasing number of threads used. Therefore the synchronization overhead is larger than the actual computational overhead caused by creating empty tasks. This has to be taken into account.

If a `libpextrae` plugin is used, the `taskbench` duration is higher than without the plugin. The blue and the brown curves represent the unmodified `libpextrae` plugin, whereas the red and black curves represent the modified `libpextrae` plugin. As expected, the modified `libpextrae` plugin results in a smaller duration of the `taskbench`. The best case ($m = 1$) of the modified `libpextrae` plugin shows only a minimal overhead compared to the disabled interface. If more task classes are created ($m = 10^6$), the overhead caused by the plugin is significantly higher.

This is due to the increasing synchronization costs, since every new task class introduces a writer-lock for writing data into data structures.

Both curves representing the unmodified `libpextrae` plugin show a significantly higher overhead. This can be explained by the additional write-lock if a specific task class was not already processed combined with saving the mapping between the task ID and the hash of the task name.

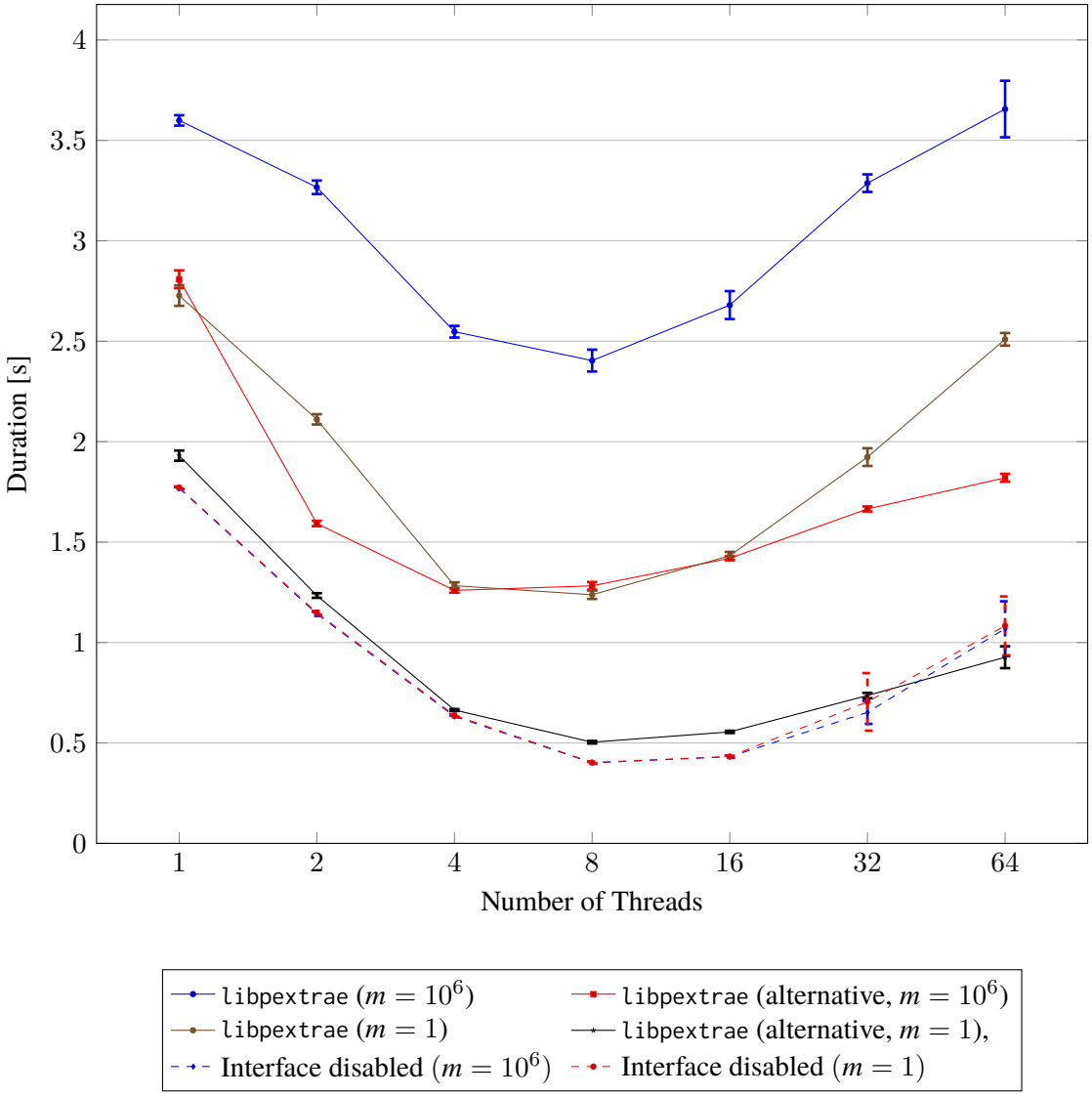

Figure 6.5: The duration of the `taskbench` for multiple versions of the `libpextrae` plugin for an increasing number of threads on a single node. The alternative version of the `libpextrae` plugin requires significantly less time than the unmodified version of the same plugin. However, the alternative version requires a slightly modified tools interface.

Impact of increasing the number of nodes

To gather insight into the scalability of the `libpextrae` plugin when using more than one node, the problem size of the Cholesky decomposition is increased proportional to the number of nodes used. The block size (320×320) is constant over all runs. For a single node, the size of the matrix is $(20 \cdot 1024 \times 20 \cdot 1024) = (20480 \times 20480)$. For more than one node, the size of the matrix is increased by multiplying each dimension by the square root of the number of nodes used. Thus, the problem size is constant per node and increasing proportional to the number of nodes used. The benchmark is run ten times for each number of nodes.

Results The results are depicted in Figure 6.6. The x-axis shows the number of nodes the benchmark was running on, whereas the y-axis shows the duration of the Cholesky decomposition benchmark in seconds. No significant overhead of the `libpextrae` plugin can be obtained, since all results only vary within the measurement accuracy. However, when using more than 16 nodes, the variance of the individual runs increases, resulting in larger error bars. All measurements shown in Figure 6.6 were obtained on the same day, with the same configuration on the cluster. To ensure that no external load on the cluster distorted the measurements, all benchmarks were repeated twice. Nevertheless, the results are reproducible.

Unfortunately, no time was left to inspect the behavior of the benchmark on more than 16 nodes in detail. A possible explanation may be the topology of the network. The cluster has 24 nodes connected to a single switch. Ideally, using less than 24 nodes should result in using a single switch only. However, this is not guaranteed. When using more than 24 nodes, at least two switches are used to connect all nodes. This results in a higher communication overhead, since other jobs running on the cluster may use additional bandwidth. However, this particular explanation has to be analyzed further to draw a final conclusion.

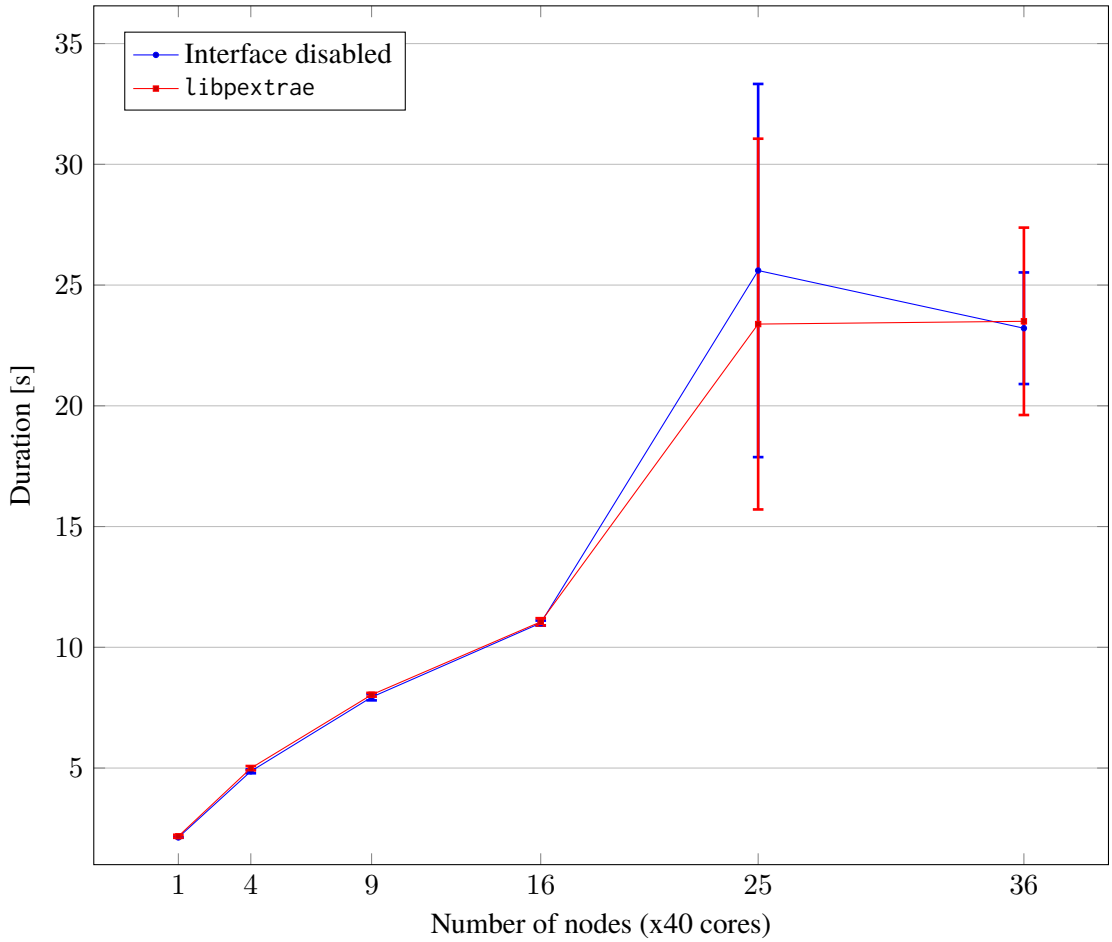

Figure 6.6: The duration of the Cholesky decomposition benchmark when using a fixed problem size per node.

6.2 Discussion

The evaluation does only contain measurements concerning the overhead of the interface infrastructure and the `libpextrae` plugin for Extrae, in order to enable performance analysis of DASH applications. The `libpayudame` plugin for Temanejo and Ayudame is not measured, since the performance overhead is negligible for debugging applications as long as the tools interface sends the data instantly to the debugger, which it does in this case.

No significant overhead for the interface infrastructure could be measured. Although, more instructions need to be processed if the interface is enabled, though the overhead caused by the additional instructions cannot be measured effectively, since the results vary within the measurement accuracy. This is true for both, real-world and microbenchmarks.

However the total number of instructions used for an application run can be obtained via the Linux performance analysis tool *perf* using the `instructions` event. For the `taskbench` with $n = 10^6$ and $m = 1$, the overhead caused by the interface infrastructure amounts to $\approx 1.3\%$ more instructions compared to a disabled tools interface. For the Cholesky decomposition benchmark using a 20000×20000 matrix with a block size of 500×500, it is even lower, amounting to $\approx 0.03\%$ more instructions compared to a disabled tools interface. Therefore the overhead caused by the interface is minimal.

A significant overhead can be measured for the `libpextrae` plugin with the `taskbench` microbenchmark. Both versions of the plugin produce an additional overhead of approx. 8.5% up to approx. 102% per task on the particular system. However, the alternative version of the `libpextrae` plugin is significantly faster. This can be seen in Figure 6.3.

In practice, this is not relevant as the results depicted in Figure 6.2 show. The duration of the Cholesky decomposition benchmark is not affected by the `libpextrae` plugin within measurement accuracy.

7 Conclusion and Outlook

This thesis explored and evaluated the design of a tools interface for the DASH C++ PGAS Framework. DASH combines the tasking approach with distributed data structures by using the Partitioned Global Address Space (PGAS) model.

The tools interface enables users to connect various external analysis tools to the DASH Runtime (DART) in order to extract data that is needed for correctness checking, debugging, and performance analysis of DASH applications at run time. This data consists of events, which model task-state changes as well as local and remote dependencies.

A prototypical implementation is provided based on the proposed design that fulfills the presented design goals. In particular, the interface provides information about the task graph of a DASH application during run time. The provided interface infrastructure allows the user to connect external analysis tools. Two external analysis tools, namely Temanejo and Extrae, were integrated into the interface infrastructure. The prototypical implementation offers plugins that provide the infrastructure to connect both tools. From a user's perspective, Temanejo can be used for correctness checking and debugging of DASH applications. Extrae offers the possibility to create traces of DASH applications. These traces give an in-depth view of the application's behavior during run time, allowing performance analysis of DASH applications.

Benchmarks were conducted to measure the overhead of the interface infrastructure and the plugin that provides the infrastructure to connect Extrae. The interface infrastructure does not produce a significant overhead compared to a disabled tools interface in both real-world and microbenchmarks. However, with microbencmarks, a significant overhead can be measured for the Extrae plugin. As the real-world benchmark reveals, the overhead is not relevant in practice, since no significant overhead can be measured here.

Outlook

Since the interface still only offers basic functionality for DASH applications, which are using the tasking functions that DASH provides, it can be extended in the future by offering more information, e.g., information about the current memory usage.

Besides that, the interface currently only supports two external analysis tools. Support for more tools can be added in the future, e.g., a native profiler.

In addition to that, the proposed changes for Temanejo and Ayudame discussed in Section 5.5 should be implemented in order to improve the support of DASH for Temanejo. This includes the distinction between local and remote dependencies together with displaying user-defined strings as edge labels, e.g., to display the data hazard case for a particular dependency. Additionally, Temanejo should display the DASH unit on which a particular task run on the GUI.

Bibliography

[AKH03] J. H. Anderson, Y.-J. Kim, T. Herman. "Shared-memory mutual exclusion: major research trends since 1986". In: *Distributed Computing* 16.2 (Sept. 2003), pp. 75–110. ISSN: 1432-0452. DOI: 10.1007/s00446-003-0088-6. URL: https://doi.org/10.1007/s00446-003-0088-6 (cit. on p. 16).

[Alm11] G. Almasi. "PGAS (Partitioned Global Address Space) Languages". In: *Encyclopedia of Parallel Computing*. Ed. by D. Padua. Boston, MA: Springer US, 2011, pp. 1539–1545. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_210. URL: https://doi.org/10.1007/978-0-387-09766-4_210 (cit. on p. 18).

[Bar10] B. Barney. *Introduction to parallel computing*. Vol. 6. 13. 2010, p. 10 (cit. on pp. 15–17).

[BDF92] B. K. Bershad, R. P. Draves, A. Forin. "Using microbenchmarks to evaluate system performance". In: *[1992] Proceedings Third Workshop on Workstation Operating Systems*. Apr. 1992, pp. 148–153. DOI: 10.1109/WWOS.1992.275671 (cit. on p. 57).

[BGN13] S. Brinkmann, J. Gracia, C. Niethammer. "Task Debugging with TEMANEJO". In: *Tools for High Performance Computing 2012*. Ed. by A. Cheptsov, S. Brinkmann, J. Gracia, M. M. Resch, W. E. Nagel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 13–21. ISBN: 978-3-642-37349-7 (cit. on pp. 25, 49, 54).

[BSC19a] BSC Performance Tools. *Extrae documentation - Extrae 3.7.1 documentation*. 2019. URL: https://tools.bsc.es/sites/default/files/documentation/html/extrae-3.7.1/ (visited on 11/28/2019) (cit. on p. 26).

[BSC19b] BSC Performance Tools. *Paraver: a flexible performance analysis tool | BSC-Tools*. 2019. URL: https://tools.bsc.es/paraver (visited on 11/28/2019) (cit. on p. 26).

[CCC07] K. M. Chandy, M. Charpentier, A. Capponi. "Towards a Theory of Events". In: *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*. DEBS '07. Toronto, Ontario, Canada: ACM, 2007, pp. 180–187. ISBN: 978-1-59593-665-3. DOI: 10.1145/1266894.1266929. URL: http://doi.acm.org/10.1145/1266894.1266929 (cit. on p. 38).

[CET+11] M. Cotton, L. Eggert, J. Touch, M. Westerlund, S. Cheshire. *Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry*. Ed. by Internet Engineering Task Force (IETF). RFC 6335. 2011. URL: https://tools.ietf.org/html/rfc6335 (cit. on p. 50).

[CGH94] L. Clarke, I. Glendinning, R. Hempel. "The MPI Message Passing Interface Standard". In: *Programming Environments for Massively Parallel Distributed Systems*. Ed. by K. M. Decker, R. M. Rehmann. Basel: Birkhäuser Basel, 1994, pp. 213–218. ISBN: 978-3-0348-8534-8 (cit. on pp. 17, 30).

[Cra14] Cray Inc. *Task Parallelism By Example*. 2014. URL: https://chapel-lang.org/tutorials/SC14/SC14-4-Chapel-TaskPar.pdf (visited on 11/07/2019) (cit. on p. 18).

[DM98] L. Dagum, R. Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE Computational Science and Engineering* 5.1 (Jan. 1998), pp. 46–55. DOI: 10.1109/99.660313 (cit. on p. 16).

[EMS+13] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, D. Lorenz. “OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis”. In: *OpenMP in the Era of Low Power Devices and Accelerators*. Ed. by A. P. Rendell, B. M. Chapman, M. S. Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 171–185. ISBN: 978-3-642-40698-0 (cit. on p. 36).

[EMS+14] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, D. Lorenz. *OpenMP Technical Report 2 on the OMPT interface*. Tech. rep. OpenMP Architecture Review Board, Mar. 21, 2014. URL: https://www.openmp.org/wp-content/uploads/ompt-tr2.pdf (cit. on pp. 13, 27).

[EYS19] B. Elis, D. Yang, M. Schulz. “QMPI: A Next Generation MPI Profiling Interface for Modern HPC Platforms”. In: *Proceedings of the 26th European MPI Users’ Group Meeting*. EuroMPI ’19. Zürich, Switzerland: ACM, 2019, 4:1–4:10. ISBN: 978-1-4503-7175-9. DOI: 10.1145/3343211.3343215. URL: http://doi.acm.org/10.1145/3343211.3343215 (cit. on p. 33).

[FFK16] K. Fuerlinger, T. Fuchs, R. Kowalewski. “DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms”. In: *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. Dec. 2016, pp. 983–990. DOI: 10.1109/HPCC-SmartCity-DSS.2016.0140 (cit. on pp. 18, 21).

[Fre19] Free Software Foundation, Inc. *GCC online documentation*. Ed. by GCC team. 2019. URL: https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html (cit. on p. 47).

[GBH18] R. Gerstenberger, M. Besta, T. Hoefler. “Enabling Highly Scalable Remote Memory Access Programming with MPI-3 One Sided”. In: *Commun. ACM* 61.10 (Sept. 2018), pp. 106–113. ISSN: 0001-0782. DOI: 10.1145/3264413. URL: http://doi.acm.org/10.1145/3264413 (cit. on p. 18).

[GL02] W. Gropp, E. Lusk. “Goals guiding design: PVM and MPI”. In: *Proceedings. IEEE International Conference on Cluster Computing*. Sept. 2002, pp. 257–265. DOI: 10.1109/CLUSTR.2002.1137754 (cit. on p. 17).

[GPH+15] A. Gómez-Iglesias, D. Pekurovsky, K. Hamidouche, J. Zhang, J. Vienne. “Porting Scientific Libraries to PGAS in XSEDE Resources: Practice and Experience”. In: *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*. XSEDE ’15. St. Louis, Missouri: ACM, 2015, 40:1–40:7. ISBN: 978-1-4503-3720-5. DOI: 10.1145/2792745.2792785. URL: http://doi.acm.org/10.1145/2792745.2792785 (cit. on p. 18).

[Gro12] W. Gropp. “MPI 3 and Beyond: Why MPI Is Successful and What Challenges It Faces”. In: *Recent Advances in the Message Passing Interface*. Ed. by J. L. Träff, S. Benkner, J. J. Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–9. ISBN: 978-3-642-33518-1 (cit. on p. 17).

[HDT+15] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, K. Underwood. "Remote Memory Access Programming in MPI-3". In: *ACM Trans. Parallel Comput.* 2.2 (June 2015), 9:1–9:26. ISSN: 2329-4949. DOI: 10.1145/2780584. URL: http://doi.acm.org/10.1145/2780584 (cit. on p. 16).

[Int17] Intel Corporation. *5-Level Paging and 5-Level EPT*. Tech. rep. Intel Corporation, 2017. URL: https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf (cit. on p. 51).

[ISSH15] T. Ilsche, J. Schuchart, R. Schöne, D. Hackenberg. "Combining Instrumentation and Sampling for Trace-Based Application Performance Analysis". In: *Tools for High Performance Computing 2014*. Ed. by C. Niethammer, J. Gracia, A. Knüpfer, M. M. Resch, W. E. Nagel. Cham: Springer International Publishing, 2015, pp. 123–136. ISBN: 978-3-319-16012-2 (cit. on pp. 23, 24).

[Ker19] Kernel.org. *Kernel.org documentation on Memory mapping*. 2019. URL: https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt (cit. on p. 51).

[KLL15] S. J. Kang, S. Y. Lee, K. M. Lee. "Performance Comparison of OpenMP, MPI, and Mapreduce in Practical Problems". In: *Adv. MultiMedia* 2015 (Jan. 2015), 7:7–7:7. ISSN: 1687-5680. DOI: 10.1155/2015/575687. URL: https://doi.org/10.1155/2015/575687 (cit. on p. 17).

[KRM+12] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, F. Wolf. "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir". In: *Tools for High Performance Computing 2011*. Ed. by H. Brunst, M. S. Müller, W. E. Nagel, M. M. Resch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91. ISBN: 978-3-642-31476-6 (cit. on p. 26).

[Lin19] Linux man-pages project. *pthreads(7) - Linux manual page*. 2019. URL: http://man7.org/linux/man-pages/man7/pthreads.7.html (visited on 10/13/2019) (cit. on pp. 15, 16).

[Mes15] Message Passing Interface Forum. *MPI: a Message-Passing Interface Standard. Version 3.1*. Tech. rep. University of Tennesse, June 4, 2015 (cit. on pp. 13, 17, 30, 31, 33, 36, 40).

[Mic19] Microsoft Corp. *About Processes and Threads - Windows applications | Microsoft Docs*. 2019. URL: https://docs.microsoft.com/en-us/windows/win32/procthread/about-processes-and-threads?redirectedfrom=MSDN (visited on 10/13/2019) (cit. on p. 15).

[Ope18] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*. Nov. 2018. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf (cit. on pp. 18, 29).

[Pro17] J. Protze. *OpenMP Tool Interfaces in OpenMP 5.0*. Nov. 15, 2017. URL: https://www.openmp.org/wp-content/uploads/SC17-Protze-2017-11-15_OpenMP-Tools.pdf (cit. on p. 28).

[PTM+17] J. Protze, C. Terboven, M. S. Müller, S. Petiton, N. Emad, H. Murai, T. Boku. "Runtime Correctness Checking for Emerging Programming Paradigms". In: *Proceedings of the First International Workshop on Software Correctness for HPC Applications*. Correctness'17. Denver, CO, USA: ACM, 2017, pp. 21–27. ISBN: 978-1-4503-5127-0. DOI: 10.1145/3145344.3145490. URL: http://doi.acm.org/10.1145/3145344.3145490 (cit. on pp. 13, 33).

[SD06] M. Schulz, B. R. De Supinski. "A Flexible and Dynamic Infrastructure for MPI Tool Interoperability". In: *2006 International Conference on Parallel Processing (ICPP'06)*. Aug. 2006, pp. 193–202. DOI: 10.1109/ICPP.2006.6 (cit. on p. 31).

[SD07] M. Schulz, B. R. De Supinski. "PNMPI Tools: A Whole Lot Greater Than the Sum of Their Parts". In: *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. SC '07. Reno, Nevada: ACM, 2007, 30:1–30:10. ISBN: 978-1-59593-764-3. DOI: 10.1145/1362622.1362663. URL: http://doi.acm.org/10.1145/1362622.1362663 (cit. on p. 32).

[SG19] J. Schuchart, J. Gracia. "Global Task Data-Dependencies in PGAS Applications". In: *High Performance Computing*. Ed. by M. Weiland, G. Juckeland, C. Trinitis, P. Sadayappan. Cham: Springer International Publishing, 2019, pp. 312–329. ISBN: 978-3-030-20656-7 (cit. on pp. 13, 18, 21, 44).

[SGC93] R. H. Saavedra, R. S. Gains, M. J. Carlton. "Micro benchmark analysis of the KSR1". In: *Supercomputing '93:Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. Nov. 1993, pp. 202–213. DOI: 10.1109/SUPERC.1993.1263469 (cit. on p. 57).

[ST98] D. B. Skillicorn, D. Talia. "Models and Languages for Parallel Computation". In: *ACM Comput. Surv.* 30.2 (June 1998), pp. 123–169. ISSN: 0360-0300. DOI: 10.1145/280277.280278. URL: http://doi.acm.org/10.1145/280277.280278 (cit. on p. 17).

[Sta19] Standard C++ Foundation. *New adopted paper: N3659, Shared Locking in C++ – Howard Hinnant, Detlef Vollmann, Hans Boehm*. Nov. 11, 2019. URL: https://isocpp.org/blog/2013/04/n3659-shared-locking (cit. on p. 54).

[Xca18] XcalableMP Specification Working Group. *XcalableMP ex-scalable-em-p Language Specification*. Tech. rep. XcalableMP Specification Working Group, 2018. URL: https://xcalablemp.org/download/spec/xmp-spec-1.4.pdf (cit. on p. 33).

[YBC+07] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, T. Wen. "Productivity and Performance Using Partitioned Global Address Space Languages". In: *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*. PASCO '07. London, Ontario, Canada: ACM, 2007, pp. 24–32. ISBN: 978-1-59593-741-4. DOI: 10.1145/1278177.1278183. URL: http://doi.acm.org/10.1145/1278177.1278183 (cit. on p. 18).

All links were last followed on December 16, 2019.

A Appendix

```
typedef void (*dart_tool_task_create_cb_t) (
  uint64_t task,
  dart_task_prio_t prio,
  const char *name,
  void *userdata
);
typedef void (*dart_tool_task_add_to_queue_cb_t) (
  uint64_t task,
  uint64_t thread,
  void *userdata
);
typedef void (*dart_tool_task_begin_cb_t) (
  uint64_t task,
  uint64_t thread,
  void *userdata
);
typedef void (*dart_tool_task_end_cb_t) (
  uint64_t task,
  uint64_t thread,
  void *userdata
);
typedef void (*dart_tool_task_cancel_cb_t) (
  uint64_t task,
  uint64_t thread,
  void *userdata
);
typedef void (*dart_tool_task_yield_leave_cb_t) (
  uint64_t task,
  uint64_t thread,
  void *userdata
);
typedef void (*dart_tool_task_yield_resume_cb_t) (
  uint64_t task,
  uint64_t thread,
  void *userdata
);
typedef void (*dart_tool_task_finalize_cb_t) (
  void *userdata
);
```

Listing A.1: The template definitions of the task state change notification functions available in the DASH Tools interface.

```c
typedef void (*dart_tool_local_dep_cb_t) (
  uint64_t task1,
  uint64_t task2,
  uint64_t memaddr,
  int32_t task1_unitid,
  int32_t task2_unitid,
  int edge_type,
  void *userdata
);
typedef void (*dart_tool_local_dep_cb_t) (
  uint64_t task1,
  uint64_t task2,
  uint64_t memaddr,
  int32_t task1_unitid,
  int32_t task2_unitid,
  int edge_type,
  void *userdata
);
```

Listing A.2: The template definitions of the dependency notification functions available in the DASH Tools interface.

```c
void ayu_event_preinit(uint64_t rt);

void ayu_event_init(uint64_t nthreads);

void ayu_event_addtask(uint64_t task_id, uint64_t func_id, uint64_t priority, uint64_t scope_id);

void ayu_event_registerfunction(uint64_t func_id, const char *name);

void ayu_event_adddependency(uint64_t to_id, uint64_t from_id, uint64_t memaddr, uint64_t orig_memaddr);

void ayu_event_addtasktoqueue(uint64_t task_id, uint64_t thread_id);

void ayu_event_preruntask(uint64_t task_id, uint64_t thread_id);

void ayu_event_runtask(uint64_t task_id);

void ayu_event_postruntask(uint64_t task_id);

void ayu_event_removetask(uint64_t task_id);

void ayu_event_barrier();

void ayu_event_waiton(uint64_t task_id);

void ayu_event_finish();
```

Listing A.3: The original Ayudame interface.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature