HLRS

Institut für
Höchstleistungsrechnen

FORSCHUNGS- UND ENTWICKLUNGSBERICHTE

MODEL-CENTRIC TASK DEBUGGING AT SCALE

Mathias Nachtmann

ISSN 0941 - 4665 2017 HLRS-17

Höchstleistungsrechenzentrum
Universität Stuttgart
Prof. Dr.-Ing. Dr. h.c. Dr. h.c. Prof. E.h. M. Resch
Nobelstrasse 19 - 70569 Stuttgart
Institut für Höchstleistungsrechnen

MODEL-CENTRIC TASK DEBUGGING AT SCALE

von der Fakultät Energie-, Verfahrens- und Biotechnik
der Universität Stuttgart zur Erlangung der Würde eines
Doktor-Ingenieurs (Dr.-Ing.) genehmigte Abhandlung

vorgelegt von

Mathias Nachtmann
aus Backnang

Hauptberichter:	Prof. Dr.- Ing. Dr. h.c. Dr. h.c. Prof. E.h. Michael M. Resch
Mitberichter:	Prof. Jesùs Labarta
Tag der Einreichung:	19.06.2017
Tag der mündlichen Prüfung:	14.05.2018
CR-Klassifikation:	I.3.2, I.6.6

D93

ISSN 0941 - 4665 2017 HLRS-17

Abstract

Chapter 1, Introduction, presents state of the art debugging techniques in high-performance computing. The lack of information out of the programming model, these traditional debugging tools suffer, motivated the model-centric debugging approach. Chapter 2, Technical Background: Parallel Programming Models & Tools, exemplifies the programming models used in the scope of my work. The differences between those models are illustrated, and for the most popular programming models in HPC, examples are attached in this chapter. The chapter also describes TEMANEJO, the toolchain's front-end, which supports the application developer during his actions. In the following chapter (Chapter 4), Design: Events & Requests in AYUDAME, the theory of "task" and "dependency" representation is stated. The chapter includes the design of different information types, which are later on used for the communication between a programming model and the model-centric debugging approach. In chapter 5, Design: Communication Back-end AYUDAME, the design of the back-end tool infrastructure is described in detail. This also includes the problems occurring during the design process and their specific solutions. The concept of a multi- process environment and the usage of different programming models at the same time is also part of this chapter. The following chapter (Chapter 6), Instrumentation of Runtime Systems, briefly describes the information exchange between a programming model and the model-centric debugging approach. The different ways of monitoring and controlling an application through its programming model are illustrated. In chapter 7, Case Study: Performance Debugging, the model-centric debugging approach is used for optimising an application. All necessary optimisation steps are described in detail, with the help of mock-ups. Additionally a description of the different optimised versions is included in this chapter. The evaluation, done on different hardware architectures, is presented and discussed. This includes not only the behaviour of the versions on different platform, but also architecture specific issues.

Zusammenfassung

Kapitel 1, Introduction, behandelt aktuelle Debugging Technologien im HPC-Umfeld. Der Mangel an Information aus dem Programmiermodell, den diese Werkzeuge aufweisen, motiviert den modell-zentrischen Debugging-Ansatz. Kapitel 2, Technical Background: Parallel Programming Models & Tools, erläutert die verschiedenen Programmiermodelle, welche im Rahmen dieser Arbeit eingesetzt wurden. Die Unterschiede zwischen den verschiedenen Programmiermodell-ansätzen werden aufgezeigt und Beispiele für die gängigsten Programmiermodelle werden aufgeführt. Desweiterem wird TEMANEJO dargelegt, das Front-End des entworfenen Tools, welches den Anwender während der Softwareentwicklung unterstützt. Im folgenden Kapitel (Kapitel 4), Design: Events & Requests in AYUDAME, werden die Konzepte von “Tasks” und “Abhängigkeiten” spezifiziert. Das Kapitel zeigt zusätzlich das Design der verschiedenen Informationstypen, welche später für die Kommunikation zwischen einem beliebigen Programmiermodell und dem modell-zentrischen Debugging-Ansatz benötigt werden auf. In Kapitel 5, Design: Communication Back-end AYUDAME, wird das Design der Back-End Infrastruktur detailliert beschrieben. Zusätzlich beinhaltet dieses Kapitel Analysen der Probleme, die während des Designprozesses aufgetreten sind. Hierfür wurden problemspezifischen Lösungen entwickelt und detailliert beschrieben. Außerdem behandelt das Kapitel den Einsatz von modell-zentrischem Debugging in einer Multi-Prozess Umgebung, bei zeitgleicher Benutzung von verschiedenen Programmiermodellen. Das folgende Kapitel (Kapitel 6), Instrumentation of Runtime Systems, beschreibt in kürze den Informationsaustausch zwischen einem Programmiermodell und dem modell-zentrischen Debugging-Ansatz. Die verschieden Möglichkeiten eine Anwendung durch ein Programmiermodell zu überwachen und zu kontrollieren werden ebenfalls aufgezeigt. In Kapitel 7, Case Study: Performance Debugging, wird der modell-zentrische Ansatz zur Optimierung einer Anwendung eingesetzt. Alle notwendigen Optimierungsschritte werden unter Zuhilfenahme von Mock-ups detailliert aufgezeigt. Die drei optimierte Versionen der Anwendung werden in Detail beschrieben. Die Auswertung der drei Versionen wird auf verschiedenen Hardware-Architekturen ausgeführt und anschließend diskutiert. Dies beinhaltet nicht nur das Verhalten der Versionen auf verschiedenen Plattformen, sondern auch Architekturen-spezifische Probleme.

Acknowledgements/Danksagung

An erster Stelle danke ich meinen Eltern, die mich während meiner schulischen und universitären Ausbildung stets unterstützten und somit diese Promotion überhaupt erst ermöglicht haben.

An zweiter Stelle geht besonderen Dank an Dr. José Gracia, der mir vor fünf Jahren am Bundeshöchstleistungsrechenzentrum Stuttgart (HLRS) als Wiss. Mitarbeiter die Chance zur Promotion gab. Seine Ratschläge und Hilfe jeglicher Art, aber auch die Freiheit die er mir für meine wissenschaftlichen Tätigkeiten einräumte, ermöglichte die in dieser Arbeit präsentierten Ergebnisse.

Ebenso danke ich Herrn Prof. Dr.-Ing. Dr. h.c. Dr. h.c. Hon.-Prof. Michael M. Resch, für die Möglichkeit als Wissenschaftlicher Mitarbeiter am HLRS arbeiten zu dürfen.

Auch möchte ich mich bei allen Projektpartern des Mont-Blanc Projekts bedanken. Viele Ideen und Konzepte sind erst durch Diskussionen innerhalb des Projektes entstanden. Besonderer Dank gilt hier Prof. Jesús Labarta der durch seine Expertise im Bereich Performanceoptimierung stets die richtigen Fragen stellte.

Ebenso möchte ich mich an dieser Stelle auch bei B.Sc. Jutta Wieland, die mich tatkräftig unterstützte, und bei Dipl.-Phys. Claudius Weimann, ohne den ich sicherlich nicht diesen Weg der Promotion eingeschlagen hätte, bedanken.

Herzlichen Dank gilt natürlich auch allen Freunden, Bekannten und Kollegen.

Mathias Nachtmann Stuttgart Juni 2017

Contents

Abstract iii

Zusammenfassung iv

Acknowledgements v

Contents vii

List of Figures xi

Abbreviations xv

1 Introduction 1
1.1 Motivation and Goal . . . 2
1.2 Programming Model . . . 3
1.3 Traditional versus Model-Centric Debugging . . . 5
1.4 Model-Centric Debugging for Task-Based Programming Models . . . 7
1.5 Performance Relevance for Model-Centric Debugging . . . 8
1.6 Performance Debugging as Case Study . . . 9
1.7 Outline of the Thesis . . . 12

2 Technical Background: Parallel Programming Models & Tools 13
2.1 Programming Model Overview . . . 13
2.2 OpenMP: Open Multi-Processing . . . 14
2.3 OmpSs: OpenMP SuperScalar . . . 16
2.4 StarPU . . . 19
2.5 MPI: Message Passing Interface . . . 20
2.6 Hybrid Programming Models . . . 23
2.7 Temanejo . . . 23

3 Related Work 25
3.1 Ayudame . . . 25

3.2 Traditional Debugging tools . . . 26
3.2.1 DDT . . . 26
3.2.2 Totalview . . . 26
3.3 Programming-Model-Centric Debugging for multicore embedded systems . . . 26

4 Design: Events & Requests in Ayudame 29
4.1 Introduction . . . 29
4.2 Events in AYUDAME . . . 29
4.2.1 Mandatory Information . . . 30
4.2.2 Additional Information . . . 31
4.2.3 Interfaces for C and C++ . . . 33
4.3 Requests in AYUDAME . . . 35

5 Design: Communication Back-end Ayudame 37
5.1 Introduction . . . 37
5.2 Motivation . . . 39
5.3 Runtime Monitoring and Controlling . . . 40
5.4 Ayudame internals . . . 42
5.4.1 Event Handler for specific programming models . . . 44
5.4.2 Connect Handler . . . 45
5.4.3 Ayu Socket . . . 45
5.4.4 Intern Event . . . 46
5.5 Multi process node environment . . . 46
5.5.1 Connection tree . . . 47
5.6 Identification and Information routing . . . 50
5.6.1 Routing events and requests . . . 50
5.6.2 Unique ID . . . 52
5.7 MPI Matcher . . . 53

6 Instrumentation of Runtime Systems 55
6.1 Introduction . . . 55
6.2 Monitoring . . . 56
6.2.1 OmpSs instrumentation . . . 56
6.2.2 OpenMP Tools Application Programming Interfaces for Performance Analysis and Debugging - OMPT . . . 61
6.3 Controlling . . . 62
6.3.1 Tasking Control API - TCA . . . 62
6.4 Simple Stepper . . . 70

7 Case Study: Performance Debugging 71
7.1 Introduction . . . 71
7.2 Optimization . . . 73
7.2.1 Fork Join . . . 75
7.2.2 Communication hiding . . . 77

7.2.3 Communication Overlap 80
7.2.4 Iteration Overlap 82
7.3 Performance Evaluation 83
7.3.1 Evaluation Platform 84
7.3.2 Tuning of number of OmpSs workers and process placement 84
7.3.3 Benchmark on Hazel Hen 86
7.3.4 Benchmark on the Mont-Blanc prototype 86
7.3.5 Comparing the Cavium ThunderX with the Cray XC40 . . . 87
7.4 Use Case Conclusion 89

8 Conclusions & Open research topics **91**
8.1 Discussion 91
8.1.1 Evaluation of the design 92
8.2 Open research topics 93

A Appendix **95**
A.1 Hardware Technology 95
A.1.1 Non-uniform memory access 95
A.1.2 Memory layers 98
A.1.3 Distributed memory 99
A.1.4 Heterogeneous systems 99
A.1.5 Knights Landing 99

Bibliography **101**

List of Figures

1.1 HPC systems placed in the TOP500 (November 2016) including their node and core number. 2
1.2 shows different abstraction layers. The *level of understanding* is placed between the *User* and the *Programming Model*. 4
1.3 shows the different abstraction layers, in addition the *Tools* are placed at the same abstraction level as the *low-level Software*. 6
1.4 shows the different abstraction layers, including the *Model-Centric Debugging approach* (located at the same abstraction level as the *Programming Model*). 7
1.5 shows a two level domain decomposition. The coarse grained decomposition can be used for distributed memory programming models and the fine grained decomposition can be used for taskification. 10
1.6 shows a domain decomposition with marked neighbour tasks. For example: The blue task(4,3) has its neighbours marked in light blue(tasks: (3,3), (5,3), (4,2), (4,4)). 11

2.1 shows the most relevant programming models for HPC 14
2.2 shows a listing of an “OpenMP parallel for” example. Every iteration can be executed by a different thread independently. 15
2.3 shows a listing of an “OpenMP section” example. Every section can be executed by a different thread. 15
2.4 shows a listing of an “OpenMP task” example. Both foo tasks can be executed in parallel. The bar task depends on the foo tasks. . . . 15
2.5 shows a listing of an “OmpSs task” example including dependencies between the tasks. According to the dependencies the tasks can be executed. 17
2.6 shows the task-dependency graph of the listing in Figure 2.5. According to the dependencies the tasks 4,5 and 6,7 can be executed in parallel. 18
2.7 shows a listing of an StarPU example code. 20
2.8 shows a listing of an MPI ring example. 21

2.9 displays a MPI ring application. The dependency graph is generated with TEMANEJO. Each colour in the dependency graph represents a different MPI rank. The different MPI operations are represented with different shapes (square: *MPI_Init*, trapezoid: *MPI_Send*, turned trapezoid: *MPI_Recv*, turned square: *MPI_Finalize*). The red lines represent the data transfer between MPI ranks, the blue marked dependencies arises from the program order of the MPI calls within a process. 22

4.1 The runtime informs AYUDAME about any relevant changes. The events are forwarded to TEMANEJO 30
4.2 shows examples for task and dependency properties. 32
4.3 C interface of AYUDAME 33
4.4 C++ interface of AYUDAME 34
4.5 C interface of the AYUDAME userdata event. 34
4.6 TEMANEJO controls the runtime/application via AYUDAME by sending request. Requests are forwarded to the runtime. 35

5.1 Modules and their functionality used in AYUDAME. The data exchange inside an AYUDAMEis accomplished by the internal “Buffers”. The inter process data exchange is done by socket communication. . 38
5.2 shows the overall control and request flow inside an AYUDAME instance. 41
5.3 shows an example of attaching AYUDAME to a programming model. 41
5.4 shows multiple runtimes (in one process) attached to AYUDAME. In addition, the possible outputs of AYUDAMEare also illustrated. . . . 43
5.5 One Process at a node is connected to TEMANEJO. 47
5.6 Multiple Process at a node are connected to TEMANEJO 48
5.7 Multiple Process at multiple nodes are connected to TEMANEJO. The communication is spanned up in a “All to one” fashion. 49
5.8 Multiple Process at multiple nodes are connected to TEMANEJO. The communication is spanned up as a “Communication Tree”, e.g. the messages are routed through AYUDAME instances. 50
5.9 shows the algorithm spanning up the“Communication Tree”. 51
5.10 shows the “Communication Tree” used for multi-process debugging. 51
5.11 The function set_master_id shifts the master_id into the m_id, the second function returns the master_id out of the m_id 53
5.12 The function set_proc_id shifts the procc_id into the m_id, the second function returns the procc_id out of the m_id 53
5.13 The function set_client_id shifts the client_id into the m_id, the second function returns the client_id out of the m_id 54

6.1 shows the different possible AYUDAME instrumentations. Some programming models need an InCode instrumentation, others can be accessed through an API. 56

6.2 shows the “addEventList” code part, used for instrumenting OmpSs. The function splits the different runtime events into “addTask”, “addDependency”, etc. . . . 57
6.3 shows the “addResumeTask” code part, used for instrumenting OmpSs. The function detects the tasks states and generates an AYUDAME event. . . . 58
6.4 shows the “addTask” code part, used for instrumenting OmpSs. The function generates an AYUDAME task creation event. . . . 59
6.5 shows the “addDependency” code part, used for instrumenting OmpSs. The function generates an AYUDAME dependency creation event. . . . 60
6.6 Stage one of the OMPT initialization . . . 61
6.7 Stage two of the OMPT initialization . . . 62
6.8 Interaction of TEMANEJO with a runtime system through TCA . . . 63
6.9 Interaction between the runtime and the tool, through TCA . . . 66
6.10 shows the tca_initialize implementation inside the tool. . . . 68
6.11 shows the lookup implementation inside the runtime. . . . 68
6.12 shows tca_initialize process inside the runtime. . . . 69

7.1 *default* code structure . . . 72
7.2 Task colouring and their meaning. . . . 74
7.3 *Fork/Join* code structure. . . . 76
7.4 *Fork/Join* dependency graph . . . 76
7.5 *Fork/Join* timeline without compute task differentiation . . . 77
7.6 *Fork/Join* timeline with compute task differentiation . . . 77
7.7 *Comm hiding* code structure. . . . 78
7.8 *Comm hiding* dependency graph with differentiation in inner and outer tasks. . . . 78
7.9 *Comm hiding* timeline with normal behaviour. . . . 79
7.10 *Comm hiding* timeline with delay in the MPI communication. . . . 79
7.11 *Comm overlap* timeline with runtime aware MPI task behaviour. . . . 81
7.12 *Iteration overlap* dependency graph. The necessary dependencies are getting complex. . . . 82
7.13 *Iteration overlap* timeline. . . . 83
7.14 timeline of combied *Iteration overlap* and *Comm hiding* . . . 83
7.15 shows the performance as a function of MPI processes and OmpSs worker threads per node. . . . 85
7.16 show the performance comparison of the different LBC versions (strong scaling experiment). These results were generated on the Hazel Hen. . . . 86
7.17 shows the comparison of two OmpSs versions with the pure-MPI implementation of LBC(weak scaling experiment). These results were generated on the Mont-Blanc prototype . . . 87
7.18 shows a weak scaling experiment running on the target systems. . . . 88
7.19 sets the MLUPs in relation to the peak performance of the system (Cavium: 345 GFLOPs, XC40: 960 GFLOPs). . . . 88

7.20 sets the MLUPs in relation to the measurable memory bandwidth (Cavium: 70 GBs, XC40: 113 GBs). 88
7.21 shows the fraction of peak performance the application reaches on the target platform. 89

A.1 Ring 95
A.2 Cross link 95
A.3 Memory bandwidth 4 socket system with 4 cores per socket. In this system no cross link available and therefore a second stage performance drop is measurable 96
A.4 Memory bandwidth 4 socket system with 6 cores per socket. In this system a cross link available and therefore no second stage performance drop is measurable 96
A.5 Non-Uniform Memory Systems @ HLRS 97
A.6 Two socket architecture. Each socket has its own Memory 97
A.7 Interlagos architecture. 98
A.8 Nehalem EX architecture. 98

Abbreviations

HPC	**H**igh **P**erformance **C**omputing
LBC	**L**attice-**B**oltzmann **C**ode
MPI	**M**essage **P**arsing **I**nterface
BEST	**B**oltzmann **E**quation **S**olver **T**ool
SIMD	**S**ingle **I**nstruction, **M**ultiple **D**ata
TCA	**T**asking **C**ontrol **A**PI
OMPT	**O**pen**MP** **T**ools Application Programming
PGAS	**P**artitioned **G**lobal **A**ddress **S**pace
DAG	**D**irected **A**cyclic **G**raph
DAC	**U**nified **P**arallel **C**
NUMA	**N**on-**U**niform **M**emmory **A**ccess
UMA	**U**niform **M**emmory **A**ccess
HBM	**H**igh **B**andwidth **M**emory
HMC	**H**ybrid **M**emory **C**ube
NVM	**N**on **V**olotile **M**emory
POSIX	**P**ortable **O**perating **S**ystem **I**nterface
PGAS	**P**artitioned **G**lobal **A**ddress **S**pace
GDB	The **G**NU Project **D**e**b**ugger

This thesis is dedicated to my parents and friends

Chapter 1

Introduction

This thesis addresses the problems during the application developing process, especially the issues occurring due to parallelisation. In complex applications, debugging tools and supporting frameworks are essential for the high-performance computing (HPC) area. Information out of the parallelisation strategy can be used to give the application developer necessary hints. TEMANEJO, a task-based debugger, converts this information and supports the developer in his activities. This work is highly motivated by the problems appearing during the parallelisation process of a traditional engineering application. Therefore, this thesis is located as an interdisciplinary field of engineering and informatics (software development).

Chapter Introduction briefly describes the context the work is placed in. The section Motivation and Goal issues the increasing complexity of hardware and the resulting software complexity. This technological progress leads to a need for tools supporting computer scientists and engineers. On the one hand, there are state of the art debugging tools. On the other hand, there is a lack of tools trying to include the means out of the parallelisation technique into the debugging process. This lack of tools is targeted by the work presented in my thesis. Therefore, TEMANEJO, which is a framework capable of extracting the available *programming model* specific information, is introduced. Furthermore in this chapter the term *Model-Centric Debugging* is defined, by trying to distinguish it from the *traditional debugging approach*. The *Model-Centric Debugging* approach is inevitable due to the increasing hardware complexity. The penultimate section Performance

Debugging as Case Study gives a brief example for an application optimisation use-case. The thesis outline finalises this chapter.

1.1 Motivation and Goal

In the past decade, the ecosystem complexity of parallel computing has increased massively. Coming from single core architectures, the system design has evolved dramatically. The present-day's computer systems are usually multi-core based. The cores in these multi-core systems are located in disjointed physical processors (central processing unit/CPU). Therefore, technologies like *non-uniform memory access*, *hyper threading*, *caching* and much more, are necessary to operate these systems efficiently. The systems used in high-performance computing today, scale up to thousands of nodes comprising millions of independent cores.

System	Cores	Nodes
Sunway TaihuLight	10.649.600	40.960
Tianhe-2 (MilkyWay-2)	3.120.000	16.000
Titan - Cray XK7	560.640	18.688
Hazel Hen - Cray XC40	185,088	7712

FIGURE 1.1: HPC systems placed in the TOP500 (November 2016) including their node and core number.

The table in Figure 1.1 shows the first three systems on the TOP500[1] list in November 2016. Also listed is the supercomputer Hazel Hen located in Stuttgart. The huge amount of nodes and cores has the need for a well-designed network infrastructure, e.g. network bottlenecks. The most important factors which limit the enlargement of these systems to exascale are cooling and immense power consumption. The work of my thesis is integrated into the hard-and software ecosystem, by introducing several abstraction layers. On the hardware side, there are technology concepts like 1) *non-uniform memory access*, 2) *memory layers*, 3) *distributed memory* and 4) *heterogeneous systems*. The detailed information about the listed technologies can be found in Appendix A. On the software side, the hardware trend caused an evolving software environment. This includes different *parallel programming concepts* and their parallelization strategies for shared and distributed memory (Chapter 2.1) systems. These *concepts* (*programming models* chapter 1.2) attempt to simplify the application development process, by giving the user a framework for application parallelisation. The outcome are highly complex applications, which use different *programming models* and techniques to achieve the best performance on a given hardware. The application complexity is hidden from the application developer by the *programming model*. Developing and especially debugging such an application requires fundamental knowledge about hard- and software. Therefore, tools have to be designed and developed to support the application developer, by simplifying the development or debugging process.

The objective of this thesis was to design a debugging tool for *task-based programming models*. The tool has to be capable of extracting and processing programming model relevant information and assist the application developer in his actions. During the execution of a parallel application, the most pertinent information, supporting the developer, is generated by the *programming model* itself. This information can be extracted by a debugging tool and could be visualised in a *programming model specific language*. Due to the specific environment in HPC systems, the debugging tool has to deal with Hybrid Programming Models (Chapter 2.6), which are normally realised using shared and distributed programming models simultaneously. Therefore, the attention of the presented work is paid to the design of the debugging tool's back-end. The quality of the designed tool is evaluated by the performance improvement of a given application; e.g. Case Study: Performance Debugging (Chapter 7).

1.2 Programming Model

This chapter briefly introduces the term *programming model*. In the chapter Technical Background: Parallel Programming Models & Tools (Chapter 2) the parallel programming models used in the scope of my thesis are described in detail. Additionally, chapter 2 classifies the different models and gives examples for the most common ones.

In the literature, there are a few clear definitions for programming model, but one is done by Skillicorn and Talia '98:

> Models that abstract from parallelism completely. Such models describe only the purpose of a program and not how it is to achieve this Parallel Computation • 135 ACM Computing Surveys, Vol. 30, No. 2, June 1998 purpose. Software developers do not even need to know

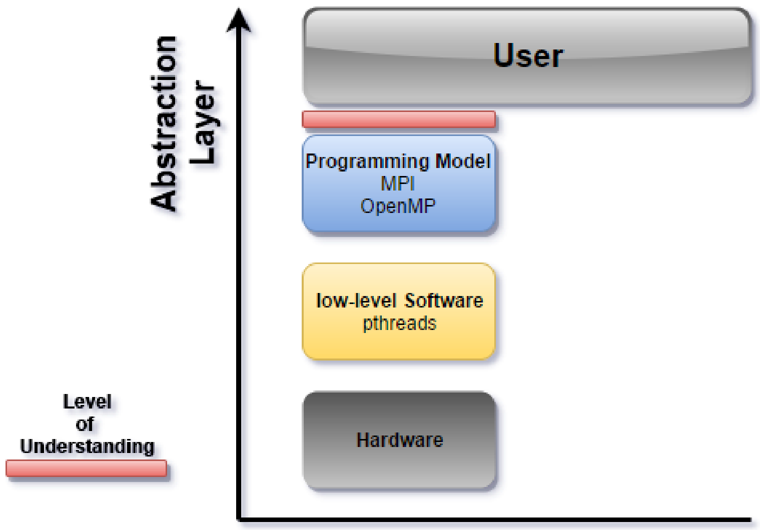


FIGURE 1.2: shows different abstraction layers. The *level of understanding* is placed between the *User* and the *Programming Model*.

> if the program they build will execute in parallel. Such models are necessarily abstract and relatively simple, since programs need be no more complex than sequential ones. [2]

A *programming model* is a model of an abstract machine, which provides certain operations to the programming 1) *level above* and requires for each of these operations an implementation on all 2) *architectures below*. As previously mentioned, the increasing complexity in hardware results in a growing complexity of the software used for programming these systems. *Programming models* try to hide the hardware architectures from the application developer. Therefore, they introduce an abstraction layer called *level of understanding*, shown in Figure 1.2, between the hardware and the application developer.
1) In Figure 1.2, the *level above* is represented by the *level of understanding* and is used by the application developer. This *level of understanding* is used for parallelising an application. The parallelising technique can be API-based calls like MPI: Message Passing Interface (Chapter 2.5) or pragma-based like OpenMP: Open Multi-Processing (Chapter 2.2).
2) The *architecture below* is represented by the low-level software layer, e.g. POSIX threads (PThreads [3]). For MPI, this low-level software layer could be a specific network architecture.

OpenMP and OmpSs: OpenMP SuperScalar (Chapter 2.3), for example, use the low-level Software PThreads to enable multithreading support in applications.

MPI is used for data exchange between different processes or nodes, hiding the communication structure (InfiniBand, extol, ethernet, Cray XC40 aries).

1.3 Traditional versus Model-Centric Debugging

Debugging is a significant part of any software development process. However, what does debugger or debugging mean? Most practical definitions of debugging would probably include the following three aspects: 1) the ability to control the program execution, in particular, the ability to suspend and resume the program execution, 2) the possibility to inspect the program state e.g. print a variable's current value and 3) optional, the potential to change the program's state, e.g. set a variable's value or set the point where the execution will be resumed. The traditional debugging process is specific for a programming language. In contrast, the *Model-Centric* [4] *approach* is specific for a programming model. For instance, pydb is a debugger for Python, while GDB is a debugger for C (C++, Fortran, etc.). All these debuggers are reasoning about the application at the lowest level, which is accessible by the programming language. In case of a single threaded C application, a debugger is reasoning at instruction level or statement level (static variable, pointers, function). The value of every single variable or pointer can be accessed, and the whole debugging process is related to the behaviour und execution on the hardware level. In case of a shared or distributed environment, the debugger has to reason across different thread or process states. In such a multi-threaded environment every PThread can have locally allocated variables, but can also access memory areas simultaneously with other threads. In a multi-threaded environment the application developer has often to deal with concurrency and their undefined side effects.

Starting a debugging process, the application developer has to switch from his high-level programming model *abstraction layer* to the low-level software layer or even to the hardware layer below, which is used in the traditional debugging process. In Figure 1.3 the "level of understanding" is added between the low-level software layer and the tools, presenting traditional debuggers like TotalView (Chapter 3.2.2), DTT (Chapter 3.2.1) or GDB. At the low-level software or the hardware layer the application developer has to care about: instruction breakpoints, memory watchpoints, event catch points, step-by-step (function, line or instruction granularity) execution and memory or processor inspection. All

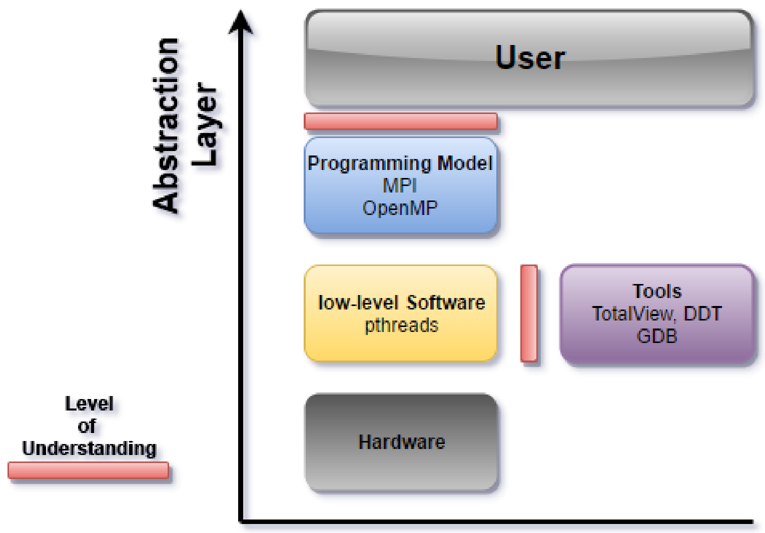


Figure 1.3: shows the different abstraction layers, in addition the *Tools* are placed at the same abstraction level as the *low-level Software*.

these traditional techniques are not taking the overall programming model related information into account. Debuggers, such as GDB, still perceive an OpenMP application as a collection of low-level threads but without any further semantics.

Other debuggers, in particular, those used in HPC such as DDT and Totalview, have some awareness of the parallel programming model and allow for instance to step all threads in a parallel region at the same time. However, Temanejo (Chapter 2.7), including its back-end Ayudame (Chapter 5), follows a strong *Model-Centric approach* and abstracts the application regarding a task dependency graph, i.e. a directed acyclic graph consisting of tasks and data-dependencies as node and edges respectively. The user may inspect task inputs and outputs or control the runtime system to step through the application task-wise or even modify task dependencies. Temanejo uses for visualisation the same *abstraction level* and vocabulary as during application development. This kind of debugging or application development (*Model-Centric approach*) is a supplement to the traditional debugging process and is using the programming model specific representation.

Today, however, the programming language is just a small part of the programmer's development environment: an application developer relies on third-party libraries, frameworks or whole programming models. In the ideal case, debuggers are *aware* of the development environment as a whole and operate on suitable *abstraction layers*, use the appropriate *vocabulary*, and *interact* with the runtime system. This debugging process is called *Model-Centric* debugging and is opposed

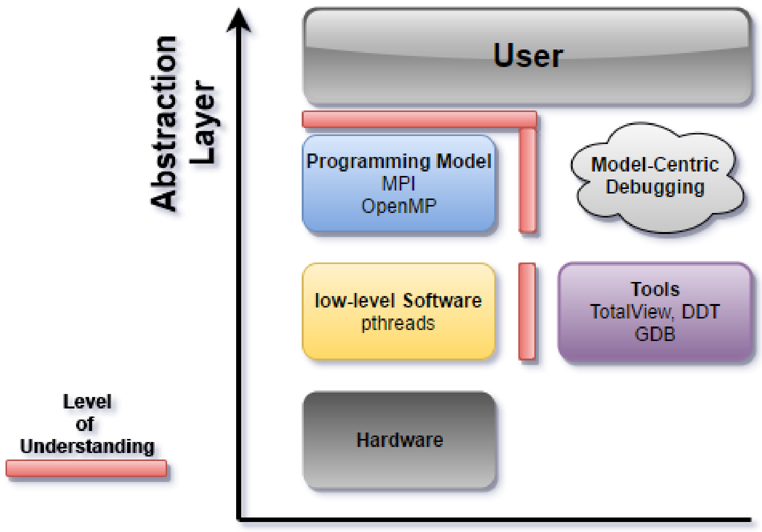


FIGURE 1.4: shows the different abstraction layers, including the *Model-Centric Debugging approach* (located at the same abstraction level as the *Programming Model*).

to the traditional debugging process. The application developers' mental representation differs from the actual execution on the underlying hardware. In Figure 1.4 the *Model-Centric Debugging* cloud is placed at the same *abstraction layer* as the programming model is located.

The objective of my thesis was to provide the application developers with means for better understanding the state and behaviour of their high-level applications and let them control the application execution from a *Model-Centric* perspective. The enhancement of the hardware and the used software and programming model also needs a strengthening of tools, e.g. including semantic information into the debugging process.

1.4 Model-Centric Debugging for Task-Based Programming Models

Integrating the *programming model* concepts into the debugging process gives the programmer additional information and hints of the overall application execution. This extracted information could be a specific task, a function execution, a parallel loop region/iteration, relationships between entities, code blocks or the data transfer between two nodes. Monitoring their different states gives a detailed view of the actual application execution, e.g. which tasks are executed, is the scheduling algorithm (longest path, depths first) working correctly? etc. It is also possible to interact with the application during execution and influence the application execution behaviour, e.g. blocking of specific tasks or step-by-step (task-by-task) execution.

Model-Centric Debugging allows illustrating communication between tasks for synchronous and asynchronous communication and their different communication patterns *one-to-one*, *one-to-many*, *many-to-one* and *many-to-many*. This allows to detect deadlocks and bottlenecks in *task-based programming models* and message-passing models. Most of these *programming models* rely on a non-static programming model. That means, the execution of tasks is not necessarily always in the same order, but through relationships between tasks, the result remains the same. Interacting and monitoring dynamic application behaviour, depending on the used *programming model*, is challenging and needs a structured concept. In most cases, the high-level *programming model* concept can be mapped into a graph structure, representing the application execution. Drawing such a graph illustrates the natural execution order or behaviour of a given application and gives the application developer a general view of the application execution.

1.5 Performance Relevance for Model-Centric Debugging

Looking at application performance as a functional debugging requirement, *Model-Centric Debugging* means to check, verify and debug the correctness of an application and additionally to analyse performance relevant issues. It is possible, that the *programming model* serialises the application execution, but however the application produces correct results. Not exploiting today's multi-core or multi-node systems, can also be seen as a bug in the application execution. The necessary information for finding bottlenecks and performance relevant issues can be extracted and illustrated in a dependency graph. This technique is used in the visualisation framework TEMANEJO (Chapter 2.7). The task placement and its memory access pattern are influencing the application performance. It is more efficient to execute tasks on cores located close to each other, because these cores are sharing the same cache or memory. The necessary tasks data should be placed as close as possible to the thread executing the task. Temanejo is capable visualising the mapping of tasks to the cores, the tasks are executed on. It is also possible to analyse the mapping of task to the corresponding NUMA domain.

Model-Centric Debugging comes to a point where not only the process of finding and resolving bugs or detecting incorrect operations, in traditional debugging techniques, but also the correct usage of the *programming model* has to be taken into account. The correct usage includes the accordance of the dependency graph, the application developer has in mind, with the dependency graph generated by the *programming model*, but also the correctness in case of locality and other performance-relevant facts.

The gathered information allows the user to access the *programming model* internal representation. This concept extends the debugging process not only with semantic information but also with sources of error, having a performance relevant aspect. This can, for example, be the width of a given graph representing the parallelism of the application. Some possible examples solving these questionings are 1) Are there too few tasks for a 24 core system? Does the application reach the maximum parallelism? 2) How many cores are necessary to get the maximum performance in case of parallelism out of the application? 3) Are there bottlenecks inside the MPI communication? Is MPI communication serialised? 3) Are tasks placed on a wrong NUMA domain and is, therefore, data locality not given? Most of these problems can be visualised in a graph-based format, which is the most suitable representation for the user.

1.6 Performance Debugging as Case Study

For any application using multiple *programming models* to express the best-optimised parallelisation, the underlying dependency graph gets extremely complex. In chapter Case Study: Performance Debugging (Chapter 7) the complete task-based optimisation workflow is shown with all intermediate steps. However, even there, the sequence and dependency diagrams have to be replaced with mock-ups. This section only explains the workflow on a fictive, well-optimised problem, which can be used as a concept for a future application developing process. The same techniques and domain decomposition have been applied to the application explained in chapter 7. For simplification, the concept is only explained in a two-dimensional

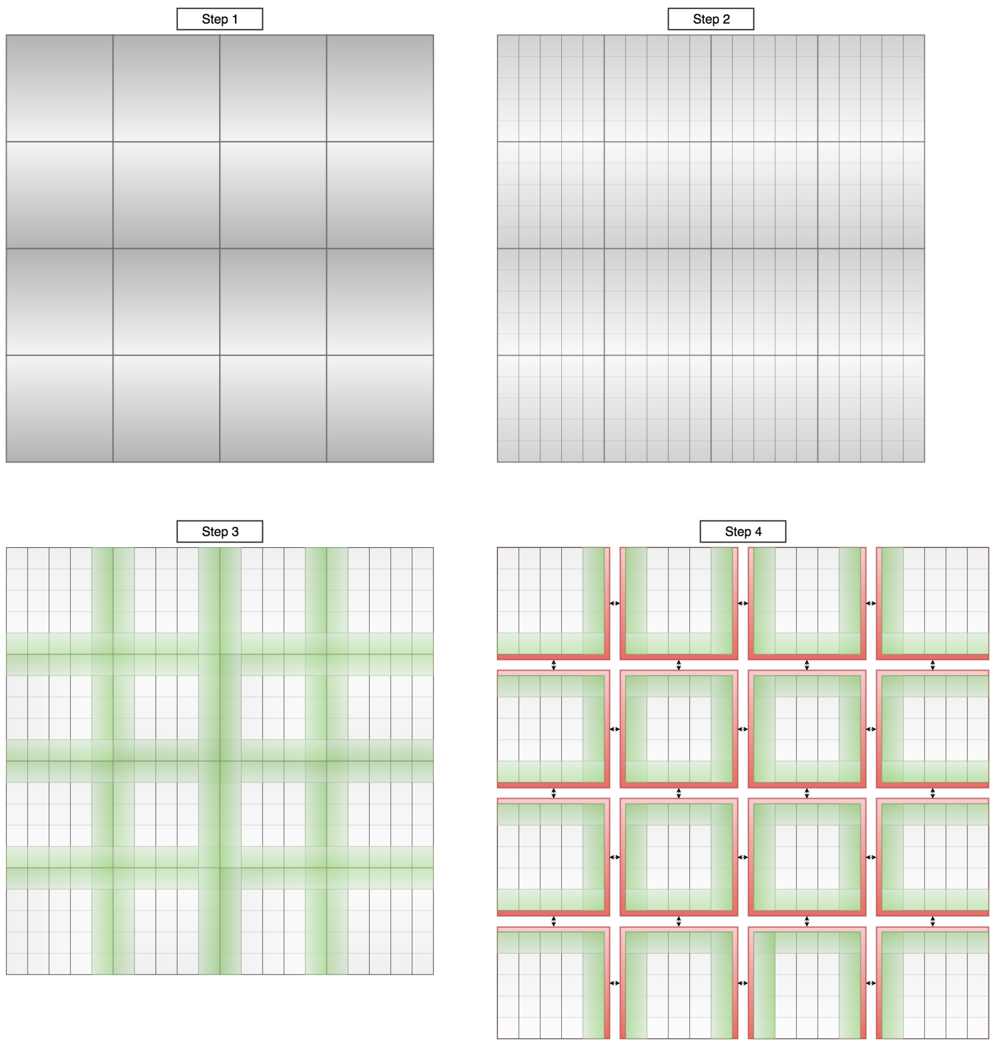


FIGURE 1.5: shows a two level domain decomposition. The coarse grained decomposition can be used for distributed memory programming models and the fine grained decomposition can be used for taskification.

domain. The following example is based on an arbitrary stencil code, which requires values for next-neighbours only. The value introduced into the domain at a given spot, diffuses over time through the domain. Solving such a problem in a hybrid application approach using MPI and a task-based *programming model* (OmpSs) has serval design, implementation and parallelisation steps.

- Figure 1.5 Step 1: Divide the domain into a subdomain for each MPI process.
- Figure 1.5 Step 2: Divide the subdomain into tasks for OpenMP/Ompss.
- Figure 1.5 Step 3: Split the tasks into tasks needed for communication(green) and tasks without communication (grey).

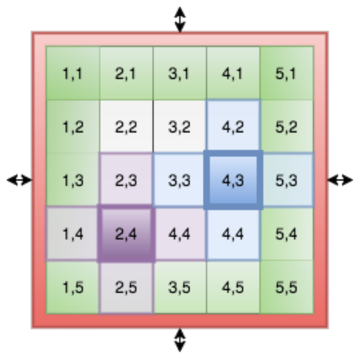


FIGURE 1.6: shows a domain decomposition with marked neighbour tasks. For example: The blue task(4,3) has its neighbours marked in light blue(tasks: (3,3), (5,3), (4,2), (4,4)).

- Figure 1.5 Step 4: Generate one task (red) per process handling the MPI communication with all neighbouring domains. This task can be executed as soon as all communication relevant tasks (green) of the same iteration have been finished.

Implementing these steps and defining the correct dependencies between the tasks will allow the runtime to schedule tasks in an efficient way. Using this technique and giving the runtime the hint of prioritising the execution of the green and red tasks automatically overlaps computation and communication. With double buffering or multi-buffering (keeping several copies of the domain, one for every iteration) it is also possible to overlap multiple iterations. In the example the domain is only divided into inner and outer tasks, this separation can also be seen as a coarse-grained stencil. A finer-grained definition of the stencil is also possible. Therefore the dependencies for every task can be built up only on the relevant neighbouring tasks. This leads to a complex dependency graph expressing the real data dependencies. Figure 1.6 shows a simple example of a fine-grained stencil. Task (4,3) for example depends on the previous execution iteration (iteration n-1) of the task (4,3) itself and the neighbour tasks ((3,3), (5,3), (4,2), (4,4)). Every green task also depends on the MPI communication task (red) of the previous iteration.

Even without this overlapping of multiple iterations and without fine-grained stencils the generated dependency graph is getting huge and complex. Without any tool supporting the application developer in the design, implementation, and parallelisation workflow it is a difficult job.

1.7 Outline of the Thesis

Concluding this chapter, I briefly summarise: The thesis addresses the lack of information flow between the *programming model* (Chapter Technical Background: Parallel Programming Models & Tools) and the application developer. The work shown in my thesis tackles this gap by providing a tool for *Model-Centric Debugging*. The infrastructure is implemented in the in AYUDAME & TEMANEJO toolchain. Chapter Design: Events & Requests in AYUDAME contains the design of events and requests, which are necessary for monitoring and controlling a runtime, respectively. The design and the issues during the tool development are present in the chapter Design: Communication Back-end AYUDAME. The interface between the *programming model* and AYUDAME is discussed in the chapter Instrumentation of Runtime Systems. Chapter Case Study: Performance Debugging demonstrates the tool usage at an application parallelisation process and gives, also, a performance evaluation of different optimised versions.

Chapter 2

Technical Background: Parallel Programming Models & Tools

This chapter gives a brief overview of today's parallel programming models used in high-performance computing (HPC). In addition, the chapter contains a section about TEMANEJO the front-end of the designed toolchain. The programming models are classified into shared and distributed memory programming models (Chapter 2.1). The most commonly used programming models are MPI for distributed applications and OpenMP for shared memory applications. Besides these two traditional models, there is a growing trend and usage of different parallelization concepts. The programming models named in the listing of Figure 2.1 are commonly used in today's HPC and engineering applications and separated into shared and distributed programming models. Therefore, I selected them as the foundation for the evaluation and design process. In the sections below there are three examples for shared memory programming models (OpenMP chapter 2.2, the StarSs family Chapter 2.3 and StarPU chapter 2.4). The chapter 2.5 Message Passing Interface (MPI) includes an example of a distributed memory programming model.

2.1 Programming Model Overview

In today's computer architecture there is a differentiation between shared and distributed memory systems and their respective programming models. In distributed memory systems, each process has its private memory and can only operate on its local data. Multiple processes communicate and exchange data through some kind of network or process interconnect. In shared memory systems, all threads share the same memory, and they have to care about concurrent data access and race conditions. A shared memory application can be seen as a single process application. As soon as the application uses more than one process, the application uses a distributed memory architecture. The processes in a distributed memory could be placed on the same node, but the operation system separates the virtual memory. The shared memory concept can be extended across the process border; a framework (PGAS) is handling the data transfer between processes or nodes. This results in an easy to use unified memory space for the application developers.

- Shared Memory
  - OpenMP [5] [6]
  - StarSs family (OmpSs, SmpSs) [7] [8] [9] [10]
  - StarPU [11] [12] [13]
  - fastflow [14]
  - Cilk
  - Threading Building Blocks
  - CUDA
- Distributed Memory
  - Message Passing Interface (MPI)
  - Partitioned Global Address Space (PGAS)
    - Global Address Space Programming Interface (GASPI)
    - Dash

FIGURE 2.1: shows the most relevant programming models for HPC

Traditionally shared memory programming models are used for on node level parallelisation. Distributed memory models are used for the interconnection between different nodes.

2.2 OpenMP: Open Multi-Processing

In high-performance computing, the standard programming model for shared-memory systems is OpenMP[5, 6]. Until recently, the programming model was

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
  a[i]= 2 * i;
}
```

FIGURE 2.2: shows a listing of an "OpenMP parallel for" example. Every iteration can be executed by a different thread independently.

```
#pragma omp parallel sections
{
#pragma omp section
  {
    printf ("id = %d, \n", omp_get_thread_num());
  }
#pragma omp section
  {
    printf ("id = %d, \n", omp_get_thread_num());
  }
}
```

FIGURE 2.3: shows a listing of an "OpenMP section" example. Every section can be executed by a different thread.

```
  int x,y,z,k;
#pragma omp task depend(in:x) depend(out:y)
  foo(x,y);
#pragma omp task depend(in:z) depend(out:k)
  foo(z,k);
#pragma omp task depend(in:y) depend(in:k)
  bar(y,k);
#pragma omp taskwait
```

FIGURE 2.4: shows a listing of an "OpenMP task" example. Both foo tasks can be executed in parallel. The bar task depends on the foo tasks.

a relatively simple flavour of the fork-join model: independent tasks were grouped in so-called parallel regions. All tasks within a region could be executed concurrently on OpenMP threads, while different regions were synchronised according to the program order. In fact, the term task did not play a major role in the OpenMP specification. Starting with version 3.0, however, concepts such as explicit and untied task, data-dependency between asynchronous tasks, and execution target for offloading of tasks have successively enriched the OpenMP programming model. With all these changes and the widely usage also the need for an OpenMP Tools Application Programming (OMPT 6.2.2) came along.

OpenMP is a pragma-based shared memory programming model for C/C++ and Fortran. The parallel programming model has various parallelisation approaches

1) loop-level parallelism, 2) parallel sections and 3) task parallelism. The loop-level (listing in Figure 2.2) approach is often called fine-grained parallelism. This technique parallelises individual loops. Each thread is working on a unique range of loop indexes. The second approach (listing in Figure 2.3) is often used for coarse-grained parallelisation: a code sections can be parallelised, not just individual loops. OpenMP 3.0 introduces the concept of tasks; e.g. a task is a self-contained unit of work. With the OpenMP 4.0 standard, tasks also can have dependencies between each other (listing in Figure 2.4), this concept is heading in the direction of the SmpSs family.

In the first and second approaches, the application developer has to care about the synchronisation between the different parallel regions. Additionally, the data used in each region can be declared as *shared* or *private*. Depending on this declaration the data is copied in (*private*) or used as a reference (*shared*), meaning other threads can access the data at the same time. If the data is declared as *shared*, the application developer has to care about race conditions and may have to access the variables in an atomic way. In the third approach, the synchronisation between tasks is given through the dependency graph. The data scoping between tasks is by default *shared*, and the access to the parameter is regulated through the dependency graph. Two tasks, both having the same parameter as input dependency, are not allowed to run in parallel. The one instantiated first, will be executed first. The task and dependency concept in OpenMP is similar to the OmpSs dependency concept and explained in chapter 2.3.

2.3 OmpSs: OpenMP SuperScalar

OmpSs, belongs to the StarSs family and is developed at BSC in Barcelona. In the StarSs family (but also in OpenMP), the programmer needs to identify suitable units of work. In general these are functions designated as tasks. This is identification, done through pragma-based code annotations. In addition to OpenMP the programming model has a stronger concept of tasks and dependencies. In contrast to OpenMP, the StarSs programming model infers the synchronisation from the data dependencies in the program. In an OpenMP or PThread application the synchronisation has to be specified by the programmer explicitly. Dependencies between tasks are generated automatically from the pragma directives, given by the user, to distinguish between input, output and input-output arguments.

Tracking the memory addresses of these task parameters allows the programming model to synchronise data dependencies. At runtime, this information is used to generate a dynamic task dependency graph. In the simplest case, tasks are executed sequentially in the same order as they have been added to the graph. This is equal to a serial code execution. In most cases, the task graph can expose concurrency and independent tasks or tasks with fulfilled dependencies can be executed in parallel on the available compute cores. The data dependencies ensure that no task is scheduled before any tasks that modifies the task's input parameters has finished its execution.

```
 1 int main ()
 2 {
 3 size_t SIZE=8;
 4 double  *a1 = new double [SIZE];
 5 double  *a2 = new double [SIZE];
 6
 7 #pragma omp task out (a1, a2) label(fill)
 8          fill (a1, a2, SIZE);
 9 #pragma omp task inout (a1) label(add)
10          add      (a1, 13, SIZE);
11 #pragma omp task inout (a2) label(add)
12          add      (a2, 5, SIZE);
13 #pragma omp task inout (a1) label(multiply)
14          multiply (a1, 3, SIZE);
15 #pragma omp task inout (a2) label(multiply)
16          multiply (a2, 7, SIZE);
17 #pragma omp task inout (a1, a2) label(add)
18          add      (a1, a2, SIZE);
19 #pragma omp task in (a1) label(dump)
20          dump (a1, SIZE);
21
22 #pragma omp taskwait
23 free(a1);
24 free(a2);
25 }
```

FIGURE 2.5: shows a listing of an "OmpSs task" example including dependencies between the tasks. According to the dependencies the tasks can be executed.

The example (listing in Figure 2.5) shows the above-explained code concept, and Figure 2.6 shows the associated dependency graph. The dependency graph shown was generated by using AYUDAME & TEMANEJO.

In the line four and five in the listing of Figure 2.5, two arrays get allocated. These arrays are filled inside the *fill function*. This task exhibits as output dependency

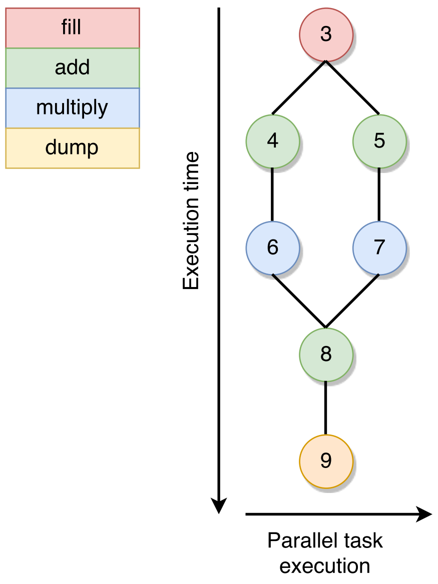


FIGURE 2.6: shows the task-dependency graph of the listing in Figure 2.5. According to the dependencies the tasks 4,5 and 6,7 can be executed in parallel.

a1 and a2. Each of these arguments is used at the *add function* as *inout dependency*. This allows both *add functions* being executed independently from each other. Because the *add function* is using *a1* or *a2* as an *inout dependency*, the following *multiply function* can only be performed after the previous *add function* in the dependency graph has been finished. The third *add function* consumes both dependencies a1 and a2, which were produced by the *multiply functions*. The last function inside the dependency graph consumes the *inout dependency* from the third *add function*. Before freeing the memory, the application needs to wait for all tasks to be finished; this is related to the synchronisation between tasks and the master thread accessing memory, which is touched inside a task.

This programming model allows writing code without any explicit synchronisation. Synchronisation is only needed if the master thread is touching memory which is used inside a task; e.g synchronisation is only necessary between the sequential code and code executed inside tasks, but not between tasks. This behaviour can be avoided by taskifiying all relevant parts of the application.

The StarSs programming model has several advantages compared to other shared memory programming models, which either use parallel loops, static dependencies or explicit synchronisation. A dynamic scheduling process allows an efficient parallelisation, even for different input data sets. With a bunch of tasks and automatic load balancing inside the application, such a concept helps to overcome scalability problems.

Some key features give the developer high flexibility:

- The programming models takes care of synchronisation between tasks; the synchronisation is given through the task-dependency graph.
- The whole process is a dynamic process. It allows efficient parallelisation even for different input data sets. Every execution has its dynamically generated graph.
- As the entire process is dynamic, the application can easily handle load imbalances and adapt its execution.
- Taskifying MPI communication allows the overlapping of communication and computation in a more simple and efficient way than a hand-written code, which uses non-blocking MPI communications. Furthermore, the runtime can be aware or can detect whether the MPI communication has finished, before rescheduling the task.

2.4 StarPU

The StarPU programming model was developed by the French national research institute INRIA. The StarPU programming model is a task-based library for hybrid architectures. The concept is similar to the OmpSs concept, and they also have an OpenMP 4 compatibility interface. StarPU uses a combination of pragma-based annotation and library calls. A task has to be declared with *__attribute__((task))*.

The key features of the StarPU programming model are: 1) Portability 2) Dependencies; Dependencies provide the programmer with a flexible way of programming and designing applications. 3) Heterogeneous Scheduling; Clusters migrate tasks to different nodes. The communication is done through MPI and the communication is automatically combined and overlapped with the intra-node data transfer and the task execution. 4) Out of Order execution 5) Extension to the C language with an OpenMP 4 compatible interface. The listing in Figure 2.7 shows an StarPU example published in the “StarPU Handbook” [15]. In line 16 the StarPu runtime is initialised. After this a task is asynchronously executed. Before the runtime is shutting down (line 22) the application has to wait for all tasks to finish (line 20).

```c
//http://starpu.gforge.inria.fr/doc/starpu.pdf
//Chapter 3
//Basic Examples
#include <stdio.h>

// Task declaration.
static void my_task (int x) __attribute__ ((task));

// Definition of the CPU implementation of "my_task".
static void my_task (int x){
  printf ("Hello, world! With x = %d\n", x);
}

int main (){
  // Initialize StarPU.
#pragma starpu initialize
  // Do an asynchronous call to "my_task".
  my_task (42);
  // Wait for the call to complete.
#pragma starpu wait
  // Terminate.
#pragma starpu shutdown
  return 0;
}

}
```

Figure 2.7: shows a listing of an StarPU example code.

2.5 MPI: Message Passing Interface

The Message Passing Interface (MPI) is a standard, which describes the data exchange between nodes in a parallel distributed memory application. The API defines different communication and exchange patterns, which are necessary for sharing information inside a distributed application. Normally an MPI application consists of multiple processes which communicate with each other. Usually, these communication patterns can be classified into 1) *one-to-one*, 2) *one-to-many*, 3) *many-to-one* and 4) *many-to-many*.
1) *One-to-one* is used, for example, to exchange ghost cell information between neighbours (*MPI_Send*, *MPI_Recv*). 2) *One-to-many* can be used, for example, to transfer a global or local density, calculated by one process, to all or a bunch of other MPI ranks (*MPI_Scatter*, *MPI_Bcast*) 3) *Many-to-one* operation can be seen as a reduction. One process has to calculate the global or local density. Therefore the process needs the density of all or a bunch of processes (*MPI_Gather*). 4) The last communication pattern is the *many-to-many* operation. In this operation each process is sending and receiving values from all other processes (*MPI_Alltoall*).

The *one-to-one* communications are also known as *point-to-point* communication; the other three (*one-to-many*, *many-to-one*, *many-to-many*) communication patterns are named *collective communication*. In addition there is the *MPI one-sided* operation, which was introduce in the MPI-2 standard. The *MPI one-sided* operations can directly access remote memory. (*MPI_Put*, *MPI_Get*).

The implementation of this standard is done by different vendors (OpenMPI, MPICH, Cray-MPICH) and is adapted to their underlying network (InfiniBand, Ethernet, Cray-Aries). The MPI standard is giving a defined interface abstracting the communication from the application developer.

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

int main(int argc, char** argv) {
  int rank, size, token;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  if (world_rank != 0){
    MPI_Recv(&token, 1, MPI_INT, world_rank - 1, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
  }else{
    token = -1;
  }
  MPI_Send(&token, 1, MPI_INT, (world_rank + 1)% size, 0,
              MPI_COMM_WORLD);
  if (world_rank == 0){
    MPI_Recv(&token, 1, MPI_INT, size - 1, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
  }
  MPI_Finalize();
}
```

FIGURE 2.8: shows a listing of an MPI ring example.

In course of my thesis, MPI is handled as a task-based programming model. All the patterns described above can be composed of simple send-recv events. For example, in case of a broadcast, one node is executing a *send* to every other node. Accordingly all other nodes are receiving this message. Looking at these *sends* and *receives* in a task based-view, a visualisation framework (TEMANEJO) can generate a dependency graph out of the extracted information. The dependencies between the nodes in the dependency graph are the data transfers between the different MPI ranks. Figure 2.9 shows an MPI ring application (listing of Figure 2.8), which

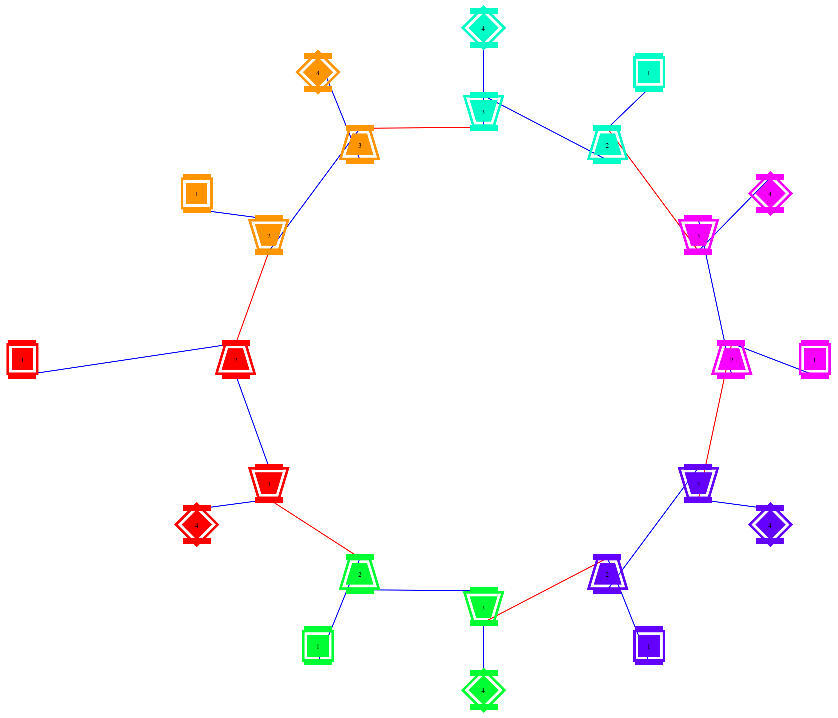


FIGURE 2.9: displays a MPI ring application. The dependency graph is generated with TEMANEJO. Each colour in the dependency graph represents a different MPI rank. The different MPI operations are represented with different shapes (square: *MPI_Init*, trapezoid: *MPI_Send*, turned trapezoid: *MPI_Recv*, turned square: *MPI_Finalize*). The red lines represent the data transfer between MPI ranks, the blue marked dependencies arises from the program order of the MPI calls within a process.

uses TEMANEJO for visualisation. Each colour in the dependency graph represents a different MPI rank. The shapes represents the *MPI_Init*, *MPI_Send*, *MPI_Recv*, and *MPI_Finalize*, respectively. The red lines represent the data transfer between two MPI ranks. The data transfer dependency is generated between the *MPI_Send* and *MPI_Recv*. At this point, the real data transfer is happening. In addition, there are blue marked dependencies between the MPI calls on each rank; these dependencies arise from the sequential MPI call order. The order of the appearance of the MPI operation inside the application, e.g. *MPI_Init* → *MPI_Recv* → *MPI_Send* → *MPI_Finalize*. Rank 0 (coloured orange) in the ring example calls first *MPI_Send* and then *MPI_Recv*.

2.6 Hybrid Programming Models

In today's high-performance compution applications, MPI is widely used and could also be seen as a standard. Actually, MPI is a distributed memory parallelisation scheme, but it is also commonly used on shared memory systems, as for example today's multi-core CPUs. On such multi-core systems, the usage of a shared memory parallelisation schemes as for instance OpenMP can lead to a more efficient utilisation of the hardware. In particular, the communication overhead is reduced, and computation and communication can be overlapped. The successful usage of hybrid parallelisation models, consisting of a distributed memory part (communication through messages between the nodes) and a shared memory part exploiting all available cores on a node, has been shown. Nevertheless, most applications are still based on pure MPI implementations and are not benefitting from a second parallelisation scheme. The classic MPI approach is challenged by today's prevalence of multicore and many-core systems. Usually pure MPI applications can't efficiently scale with the increasing amount of cores and nodes. Hybrid applications, however, only have to scale with the number of nodes. Assuming the MPI-scalability is caused by the growing amount of MPI-ranks, the application developer has the opportunity by annotating his application with pragmas to go hybrid. Going hybrid means, in general, to redesign and rewrite parts of the application. For some well-structured applications, having a second domain decomposition besides the MPI domain decomposition, dividing the subdomain of each MPI-rank into smaller work packages pragmas can be used to enable shared memory parallelisation.

The scaling effect mostly happens for strong scaling experiments, due to changing the ratio between computation and communication. The domain stays the same but is decomposed into several sub domains, one for every MPI process. The amount of computation for every process decreases, but the total amount of needed MPI communication increases.

2.7 Temanejo

Temanejo is a graphical debugger for task-based programming models. Strictly speaking, Temanejo is the front-end of the debugger. It is drawing and analysing the dependency graph and handles user's interactions. Temanejo is connected to its back-end library Ayudame by socket communication.

Temanejo is written in Python, using QT as graphical user interface. The socket communication and the *marshalling unmarshaling* library is written in C++ and swigged (Swig [16]) into Temanejo and was formerly developed for Ayudame. The same library is used for inter-node inter-process communication. Marshalling means to transform an object or structured data into an easily transportable format. The counterpart is unmarshaling rebuilding the object or structured data out of the format used for transportation. Every object has the capability to transform itself into a byte stream/sequence. Furthermore, every object can be constructed out of a correctly structured byte stream. These objects can either be events or requests. In practice, the objects are transformed into an XML-based format and back into objects.

Chapter 3

Related Work

3.1 Ayudame

Referring to Ayudame in my thesis I always mean Ayudame 2.0. Ayudame 1.0 [17] [18] [19] is not developed by me and is, therefore, part of the related work. The concept used in both libraries is the same. They are both gathering and exchanging information between a programming model and Temanejo. But that is almost everything they have in common. With the design and implementation of Ayudame 2.0, there was also the need for a new design and implementation of Temanejo (called Temanjeo 2.0). Coming from a very static and fixed event & request system in Ayudame 1.0, Ayudame 2.0 is using a flexible and generic event & request system to allow different programming models to be easily integrated. Ayudame 1.0 was strictly linked and implemented to support SmpSs and its successor OmpSs. Ayudame 2.0 supports the usage of hybrid applications using multiple programming models (MPI+OmpSs). From a single node and a single process supporting Ayudame 1.0, Ayudame 2.0 moved to a library running on several processes/nodes and is also capable of scaling with the application.

3.2 Traditional Debugging tools

3.2.1 DDT

Allinea DDT [20] (based on GNU [21])is one of the leading parallel debuggers used in HPC. The debugger supports a wide range of parallel architectures even besides the today's HPC systems (ARM32, ARM64) and parallel programming models, including MPI, UPC [22], CUDA and OpenMP. The newest version Allinea Forge, combines Allinea DDT and Allinea MAP. Allinea MAP is a low-overhead and line-based profiler for MPI, OpenMP and scalar programs. Allinea DDT is capable of debugging: 1) single process and multithreaded software, 2) OpenMP parallel applications, 3) applications using MPI, 4) heterogeneous applications using GPUs, 5) hybrid applications combining different programming models and 6) Allinea DDT is capable of debugging multi-process applications. The tool supports the mainstream languages (C, C++, Fortran90, CUDA) used in HPC and also the most common programming models (MPI, OpenMP, UPC, Co-array Fortran, etc.). As back-end the debugger uses a modified gdb version.

3.2.2 Totalview

Totalview [23] is the sophisticated software debugger from Rogue Wave Software, Inc. The debugger is used for debugging and analysing both, serial and parallel programs. Like DDT the software is also designed for the usage with complex multi-process and multi-threaded applications. Totalview supports the major HPC platforms in the U.S., in addition, there are also parts for NEC, etc. available from 3rd-party sources. The tool supports the mainstream languages (C, C++, Fortran90, Assembler) used in HPC and the most common programming models (MPI, OpenMP) as well.

3.3 Programming-Model-Centric Debugging for multicore embedded systems

This paragraph is related to the work Kevin Pouget did during his PhD thesis [24] and explains shortly the solution he has chosen. For a short remark, the work I did is directly connected to the programming model. AYUDAME monitors and interacts with the runtime. Kevin Pouget has chosen another way for monitoring and interacting. He developed a Python-based library, interacting with the gdb debugger, called mcgdb. Therefore, he has to detect the low-level instructions of the runtime and translate this information into programming model behaviour, e.g. 1) instruction x means task creation, 2) instruction y means task execution, etc. Looking again at the Figure 1.4 his approach is vertical translating from a lower abstraction layer to a higher layer, while the approach I chose is horizontal, translating directly from the programming model into the *Model-Centric Debugging* approach. Kevin Pouget is using the TEMANEJO & AYUDAME tool infrastructure to transfer and display the information extracted by mcgdb. He also introduced a new graphical representation in TEMANEJO, using sequence diagrams in addition to the dependency graph representation.

Chapter 4

Design: Events & Requests in Ayudame

4.1 Introduction

Communicating messages between an application (Chapter 1.6) and a visualisation front-end (TEMANEJO chapter 2.7) require a generic interface. This interface is designed to exchange defined message types in a standardised way. Two major goals have to be fulfilled: The support of 1) hybrid programming models and 2) distributed applications. The hybrid programming models are located in the same process, e.g. MPI+OpenMP, but in a distributed application there could be multiple of these processes, located on different nodes. Information monitored by the instrumentation of a given programming model are called events (Section 4.2). Events are always passed from the application towards TEMANEJO or any user interface. Messages coming from the visualisation and transferred towards the programming model are called requests (Section 4.3). These requests can control the runtime behaviour.

4.2 Events in Ayudame

Events in AYUDAME and TEMANEJO are messages originating from the different instances of the application. They need to be passed towards the visualisation front-end. The box in Figure 4.1 shows a brief example of the different event

Events **Monitoring**

- Task
  Parameters: task ID, scope ID, task label
- Dependencies (synchronisation between tasks)
  Parameters: dependency ID, from task ID, to task ID, dependency label
- Properties (tasks or dependency characteristic)
  Parameters: property ID, property key, property value
  - task state
  - priority
  - function name
  - ...

FIGURE 4.1: The runtime informs AYUDAME about any relevant changes. The events are forwarded to TEMANEJO

types, and also the necessary parameters for every event are listed. Events can be split up into mandatory information (Section 4.2.1), like tasks and dependencies, and optional or additional information (Section 4.2.2), containing properties and payload. For building a dependency graph only the *mandatory information* is necessary. The *additional information* is optional and can be used to add task or dependency characteristics to the graph.

4.2.1 Mandatory Information

Mandatory information is needed to build up the dependency graph in TEMANEJO. The two events needed are tasks (Section 4.2.1), representing the nodes in the dependency graph, and dependencies (Section 4.2.1), the edges between the nodes. In a dependency graph each task can have several predecessors and only if all of the predecessors have been executed the task is allowed to run. After the execution of the task, all successor dependencies are resolved. Tasks with resolved dependencies can be executed. The dependency graph is represented through a directed acyclic graph (DAG).

Tasks are the nodes in the dependency graph. With this graph representation one requirement comes up: Every task can only appear once in the DAG, and has a *unique identifier* (Section 5.6.2) and a *scope identifier.* This means one task in the parallel programming model will only be executed once. There can be multiple task instances from a given code region, but their parameters may be different. The *scope identifier* tells the tool if the task is generated from inside another task.

Optional: The node characteristic depends in Temanejo on the properties, which can be added as additional information through the property event, explained in section 4.2.2. Node characteristics in Temanejo are for example the node colour, node shape, margin colour and node label. For any of these characteristics, a filter can be applied in Temanejo. Through the UI an attached property for a characteristic can be selected, e.g. the node colour can be set as a representation for the function name. All nodes with the appropriate property will be coloured according to their value, e.g. function name "foo" will be coloured blue, function name "bar" will be coloured red, etc.

Dependencies represent the edges in the DAG. Dependencies also have requirements, a *unique identifier* (Section 5.6.2), a *from task identifier* and a *to task identifier.* Equivalent to the task characteristic, further dependency characteristic can also be set through a property.

Optional: The Edge characteristics in Temanejo are for example the edge colour and edge label. Like tasks, the dependency characteristic can be adjusted according to a property.

4.2.2 Additional Information

Additional Information is not needed to generate a dependency graph. In Temanejo there are two different types of additional information: *properties* and *payload.* Properties extend the graph with task or dependency characteristics. The Payload is information needed by the tool developer, it is either used in Ayudame or it can be used in Temanejo.

Properties are used for characterising the nodes and edges. The information is transferred as a key-value pair and belongs to exactly one node or edge (unique identifier). The following table shows some examples of properties, but they can vary between different programming models. In some programming models, for example OmpSs, there is information about function names available. In contrast MPI has no function name concept. Figure 4.2 lists some examples for task and dependency properties.

key	value
function name	foo, bar
MPI rank	1, 2
priority	99, 1
thread ID	1, 99
numa node	1, 2
cpuid	11, 12
memoryaddress	0x00000000006031c9, 0x000000000040157f

Figure 4.2: shows examples for task and dependency properties.

Payloads are messages with runtime specific information. These messages are used by modules inside Ayudame or Temanejo. The structure of these messages contains an identifier followed by '#' and then the payload depends on the tool developers needs. The implementation is shown in the listing of Figure 4.5.

In case of MPI, a matcher module (Chapter 5.7) is implemented inside Ayudame, this module generates out of two datasets, (for example, *MPI_Send* & *MPI_Recv*) the correct dependency between two tasks. This is necessary, because on each site only incomplete information about the dependency is available. The *MPI_Send* site just knows the from (sender) *task_ID*, but doesn't have any information where the dependency is going to, or the event's destination. The *MPI_Recv* site only has the information about the to (receiver) *task_ID*, but doesn't know where the dependency is coming from. Both have incomplete information about the dependency between those two tasks. Therefore, partial information is sent as payload towards Ayudame, where a matcher module generates the dependency out of the payload. The matcher module is simular to the MPI runtime error detection tool MUST [25].

There is also another case where the payload is used by INRIA (Chapter 3.3) to transfer additional information to Temanejo. They use the payload structure to transfer data needed for sequence diagrams generation in Temanejo. Therefore, Temanejo is extended with additional functionality.

4.2.3 Interfaces for C and C++

Two interfaces are provided, one interface for C and one for C++. For both interfaces AYUDAME takes care of making the *client ID* unique among all instrumentation instances. The *client ID* identifies a single programming model inside a distributed environment. Therefore, AYUDAME adds additional information (Section 5.6.2) to the *client ID*.

```

/**
get_client_id could be deleted
*/
ayu_client_id_t myclient_id = get_client_id(AYU_CLIENT_RESERVED);

ayu_event_data_t data;

data.common.client_id = myclient_id;
data.add_task.task_id = 1;
data.add_task.scope_id = 0;
data.add_task.task_label = "I'm task number one";
ayu_event(AYU_ADDTASK, data);
ayu_wipe_data(&data);

data.common.client_id = myclient_id;
data.add_dependency.dependency_id= 101;
data.add_dependency.from_id=1;
data.add_dependency.to_id=2;
data.add_dependency.dependency_label="dependency from 1 to 2";
ayu_event(AYU_ADDDEPENDENCY, data);
ayu_wipe_data(&data);

char wd_description_buffer[1024];
sprintf(wd_description_buffer,"%s", wd_description);
data.common.client_id = myclient_id;
data.set_property.property_owner_id= 1;
data.set_property.key="function_name";
data.set_property.value=wd_description_buffer;
ayu_event(AYU_SETPROPERTY, data);
ayu_wipe_data(&data);
```

FIGURE 4.3: C interface of AYUDAME

The C interface consists of the *ayu_event(event_type, data)* call containing the event type (*AYU_ADDTASK*, *AYU_ADDDEPENDENCY*, *AYU_SETPROPERTY*, *AYU_USERDATA*), and the event argument. The listing in Figure 4.3 shows a usage example of the AYUDAME C interface. All events are passed by the *ayu_event* function call, which is defined as an extern C function inside AYUDAME. The *get_client_id* function returns a *client ID* representing an AYUDAME *Client Handler*. With this ID all events sent by the runtime, can later on be identified. With the *ayu_event_data_t* structure, the different event types are generated and passed into AYUDAME calling the *ayu_event* function. The *ayu_event_data_t* structure has to be wiped after an event is dispatched. The *ayu_event_data_t* is designed as a union of different structures; this means the structure is capable of containing different event data sets (add_task, add_dependency, etc.).

```
 1 auto ayu = get_ayudame_ptr(AYU_CLIENT_RESERVED);
 2 
 3 //ayu->add_task(task_id, scope_id, task_label);
 4 ayu->add_task(1, 0, "I'm task number one");
 5 
 6 //ayu->set_property(property_owner_id, key, value);
 7 ayu->set_property(1, "function_name", "foo");
 8 
 9 //ayu->add_dependency(dependency_id, from_id, to_id, dep_label);
10 ayu->add_dependency(101, 1, 2, "dependency from task 1 to task 2");
```

FIGURE 4.4: C++ interface of AYUDAME

```
ayu_client_id_t myclient_id = get_client_id(AYU_CLIENT_RESERVED);

ayu_event_data_t data;
data.common.client_id = myclient_id;
char str[50];

sprintf(str, "MPI#%d#%d#%d#%d#%d#%d",   m_envelope[0], m_envelope[1],
                                        m_envelope[2], m_envelope[3],
                                        m_envelope[4], m_envelope[5]);
data.userdata.data = str;
ayu_event(AYU_USERDATA, data);
ayu_wipe_data(&data);
```

FIGURE 4.5: C interface of the AYUDAME userdata event.

The C++ interface, consists of separate function calls for every event type: a) *add_task*, b) *add_dependency*, c) *set_property*, d) *add_userdata*. The listing in Figure 4.4 shows the AYUDAME C++ interface. A *Client Handler* object pointer is returned by the *get_ayudame_ptr* function. As functions parameter the type of the runtime is passed. All member functions of the object can then be directly used. The *client ID* has not to be passed explicitly, because the returned *Client Handler* object contains this information.

a) The function *add_task* has three parameters. The first one adds the *task ID*. This *task ID* ("1"), line 4 in the example, has to be unique among all tasks and dependencies in the instrumentation instance. The second one is the *scope ID* ("0") of the task and the third is the task label ("I'm task number one").

b) The function *add_dependency* has 4 parameters. The first one adds the *dependency ID*. This *dependency ID* ("101"), line 10 in the example, has to be unique among all tasks and dependencies in the current instrumentation instance. The second one is the *from task ID* ("1") and the third is the *to task ID* ("2"). The last one is the dependency label ("dependency from task 1 to task 2"). Only dependencies can be added between two task in your local instrumentation, where all needed information is available. If there is the need for a dependency between two tasks in different instrumentation instances, this can be done by using a matcher 5.7 and a *add_userdata* event.

c) The function *set_property*, line 7 in the example, has 3 parameters. First one is the *property owner ID* ("1"), this links the property either to a task or to a dependency, the second one is the key ("function name") and the third is the value ("foo").

d) There is one additional event, adding support for tool developers. The function *add_userdata*, shown in the listing of Figure 4.5, has just one parameter. This parameter has to contain an leading identifier followed by '#'. The following junk of data can be set by the tool developer and can provide any information the tool developer needs inside AYUDAME or TEMANEJO. If the information can't be used inside AYUDAME the userdata is forwarded towards TEMANEJO.

4.3 Requests in Ayudame

Request **Controlling**

- Parameters: request ID, key , value
  - set/delete a breakpoints
  - step specific task
  - set progress break/continue/step
  - break/unbreak a task
  - break/unbreak a thread
  - add/remove dependencies between tasks

FIGURE 4.6: TEMANEJO controls the runtime/application via AYUDAME by sending request. Requests are forwarded to the runtime.

Requests in AYUDAME and TEMANEJO are messages, transferred from TEMANEJO to a specific *Event Handler* inside an AYUDAME instance. The different request types are encoded as key value pair. The box in Figure 4.6 shows some of the most common requests, which are also supported by the Tasking Control API (TCA, Chapter 6.3.1). The request contains an *unique ID* (Section 5.6.2) to identify the destination of the request and a key value pair. The key is the action of the request and could be for example:
1) *block*, 2) *add dependency*, 3) *break task*, 4) *step*, 5) *unblock thread*, etc.
The value is, for instance 1) a *task ID* the action has to be executed on, 2) the two tasks a dependency has to be added or 3) an action that can be relevant for the programming model itself, like step, break or continue.

Chapter 5

Design: Communication Back-end Ayudame

5.1 Introduction

This chapter shows the internal concepts necessary for the communication back-end. For the goal of the thesis, the communication back-end is represented through AYUDAME and the front-end is represented through TEMANEJO. Every programming model is directly connected to a module, handling the specific needs of the given programming model. These modules and their model specific implementation are called handlers. Through this handler, AYUDAME communicates events for monitoring purposes, and also process requests for controlling purposes. The instrumentation framework has to be implemented for different programming models and is explained in chapter 6. The instrumentation is used for translating events and requests into runtime specific behaviour. Events & requests are the internal data representation of the TEMANEJO & AYUDAME toolchain.

Figure 5.1 shows the interaction of the different models inside AYUDAME. The central point inside the AYUDAME library are two storage units, called buffers. They buffer objects of the type *Intern Event* (Section5.4.4); these objects can either be events or requests. On the left side, a *Client Manager* stores the *Event Handlers* (Section 5.4.1) for the connected programming models. The *MPI Handler* in this scenario has two additional modules included, the *MPI Matcher* (Section 5.7) module and the *Ayu Socket* (Section 5.4.3) module. In case of the *MPI Handler*,

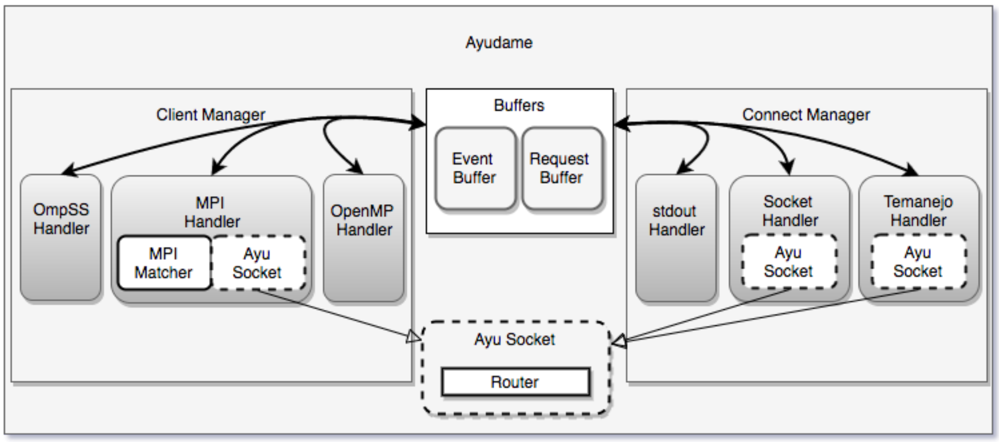


FIGURE 5.1: Modules and their functionality used in AYUDAME. The data exchange inside an AYUDAMEis accomplished by the internal "Buffers". The inter process data exchange is done by socket communication.

the *Ayu Socket* module is configured in server mode, allowing other AYUDAME instances to connect. On the right side, a *Connect Manager* stores the *Connect Handlers* (Section 5.4.2). They can be seen as different outputs of AYUDAME. This output can also be remote and therefore the *Ayu Socket* module is used. In case of such a remote communication, the *Ayu Socket* module inside the *Connect Handlers* is configured in client mode. The *Ayu Socket* is capable of connecting an AYUDAME instance either to TEMANEJO or another AYUDAME instance. Also, this module can route events & requests depending on their *Unique ID* (Section 5.6.2) to its destination. Every internal communication between the right and left side is done through the central buffers.

As a conclusion, the communication library attaches to the programming model through an instrumentation framework, translating the information into AYUDAME conform data. Furthermore, this information is extended with necessary additional information and then transferred towards TEMANEJO. In case of multiple processes, AYUDAME has to build up a communication structure between the different AYUDAME instances. These instances can run in different processes or different nodes. In addition, the communication towards TEMANEJO is done with socket communication and outlined in section 5.5.1. All necessary programming model depending implementations are done inside the Event Handler.

5.2 Motivation

Why is a communication back-end necessary? In the simplest case, the AYUDAME library is directly sitting between the visualisation front-end and the runtime of a given programming model inside the application. As long as the goal is to monitor and control an application, running in different processes, a middleware communication library, fetching and distributing the needed information, is necessary. If it is only necessary to do post analysis of the application or to do performance analysis, such a library like AYUDAME is not required, because the information can be written to disk and the analysis can be done on the stored data. For all interactive activities, communication between the applications developer, the visualisation, and the application is essential. During the design and implementation of the back-end library, AYUDAME, three major aspects where focused and problem specific solutions were developed:

Generic & Adaptable Interface — Chapter 5.4

Having multiple different programming models based on different concepts derives into different AYUDAME implementations supporting these programming models. For some applications, the library has to support multiple runtimes at the same time. Therefore, the library needs to identify from which runtime the API call is coming from and has to behave in a programming model specific way. Also, TEMANEJO supports a generic and adaptable way for displaying the information.

Scalability — Chapter 5.5

Zooming out of a very basic workflow like: programming model, instrumentation, AYUDAME, TEMANEJO; to a more complex one, there is something like a distributed memory application, with multiple nodes and processes connected. There, the application will have several instances of a used programming model and its runtime; all these instances have to be connected to the visualisation front-end. Scaling the debugging workflow large enough, there is no possibility to connect all AYUDAME instances to one visualisation front-end at the same time. Therefore, the solution is to connect communication libraries to each other, spanning up a *communication tree*. Looking at this concept in detail we will find the communication library acting in different ways. As before, each of the communication libraries is linked to one process. The leaf nodes are connected through sockets to other communication libraries, called *master* or *super* AYUDAME. Only the *super* communication library has a socket communication opened towards the visualisation front-end.

Unique Identification & Information routing Chapter 5.6

In a distributed memory environment, there has to be a mechanism to identify every single event & request. Therefore, AYUDAME generates *Unique IDs* for every single event or request in the shared or distributed memory environment. In most cases, events have only to be transferred towards TEMANEJO. For the requests the concept is more complex. There, the library uses the *Unique ID* to route the request to its destination. In a simple example, one runtime is sending the event about a task creation to TEMANEJO, through the AYUDAME *communication tree* and TEMANEJO displays the information. The user of the debugger now wants to block this specific task from being executed. Through the *Unique ID* this block task request is sent back to the runtime, which generated the task and is now able to block this specific task.

5.3 Runtime Monitoring and Controlling

This section is about the monitoring and control flow inside the communication library. Monitoring means the information is going from the runtime to the visualisation front-end, this information is called events (Section 4.2). Controlling means the information is going from TEMANEJO to the runtime, this information is called requests (Section 4.3). There are several different concepts necessary to handle the different communication library layers. Starting from a single process application debugging process and the overall control and request flow, shown in Figure 5.2, the debugging process can be split into three parts: 1) the executable, 2) an aggregation layer and 3) a visualisation layer.

1) The execution layer contains the user code; the application has some architectural relevant behaviours, is maybe using a communication library like MPI or another programming model and can be written in different languages (or is also mixing different languages). The juncture between the aggregation layer and the execution layer is designed inside the programming model, because the necessary information for *Model-Centric Debugging* is available inside this code part.

2) AYUDAME, in our case, represents the aggregation layer/communication library. This library has a defined interface to interact with different information extraction

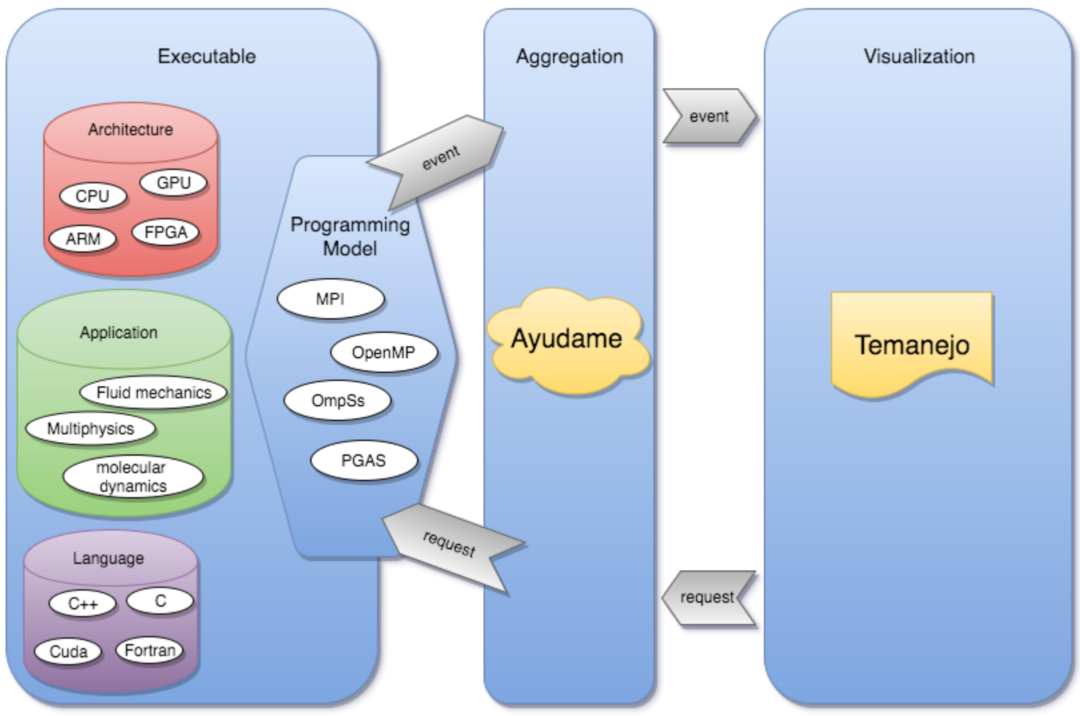
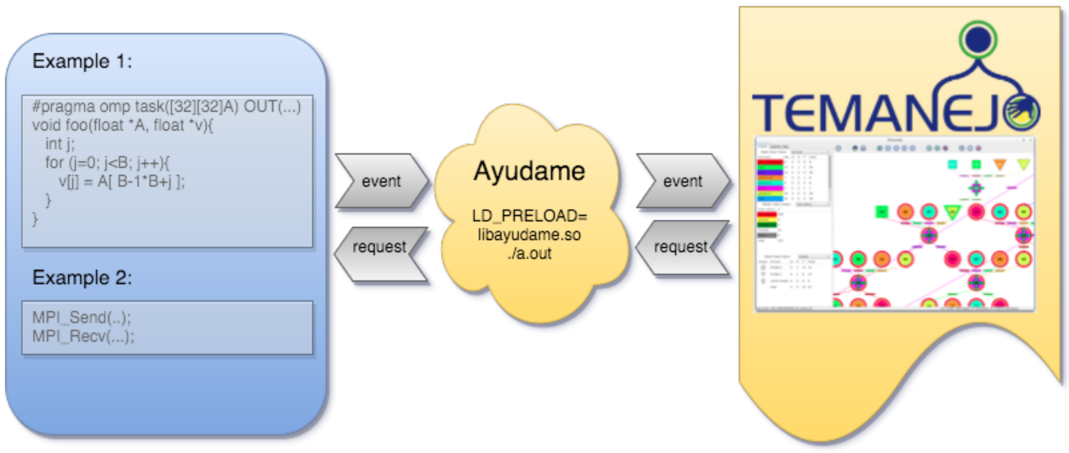


FIGURE 5.2: shows the overall control and request flow inside an AYUDAME instance.

points of the programming model and can also pass instructions back into the programming model.

3) The visualisation layer (TEMANEJO) is the user end-point and is responsible for structuring and filtering the extracted information. TEMANEJO also allows the user to interact with the application. For visualisation of the data a directed acyclic graph (DAG) is used. Events & requests are used for passing information between the different layers.



```
#pragma omp task([32][32]A) OUT(...)
void foo(float *A, float *v){
   int j;
   for (j=0; j<B; j++){
      v[j] = A[ B-1*B+j ];
   }
}
```

```
MPI_Send(..);
MPI_Recv(...);
```

```
LD_PRELOAD=
libayudame.so
./a.out
```

FIGURE 5.3: shows an example of attaching AYUDAME to a programming model.

The concept introduced in the section above is shown in Figure 5.3 for the debugging process between an OmpSs & MPI application and AYUDAME & TEMANEJO. The application uses the #PRAGMA OMP TASK for parallelizing a given code block. For communication between processes *MPI_Send* & *MPI_Recv* is used. With a preloaded AYUDAME library, which includes the necessary instrumentation for the two given programming models, the tool can extract information about the task states, the dependencies between the tasks, etc. and also the MPI communication between the *MPI_Send* & *MPI_Recv*. All the extracted information is transferred and represented in TEMANEJO. Additionally, the figure shows the previously discussed case of multiple programming models inside one process. It is also indicated, that AYUDAME is used for data aggregation, but can be used for visualisation or text based output (stdout or dot), as well.

5.4 Ayudame internals

This section explains the internal modules used in AYUDAME. The internal modules are split up into *Event Handlers* for different programming models. *Connect Handlers* for the user interaction or interaction with other AYUDAME instances. The *Ayu Socket* is able to connect to remote instances (including a router for events & requests) and the *Intern Event*, the internal representation of events & requests. As stated previously, it is necessary to have different implementations for each programming model. In Figure 5.4, there are three parts of the debugging process illustrated. On the left side, there is the executable part , containing the programming model with its specific instrumentation. In the middle, there is the aggregation layer, AYUDAME and on the right side, there is the visualisation layer.

The runtime instrumentation talks directly to one specific instance of the *Event Handler*, illustrated in Figure 5.4 as grey boxes MPI or OmpSs. The *Connect Handler*, TEMANEJO and stdout, are shown as yellow boxes. All *Event Handler* and *Connect Handler* are derived from a base class, which contains basic functionality. This could for example send messages towards TEMANEJO or receive messages from TEMANEJO. Both handlers are defined in a generic way, but can be adapted and extended with programming model specific behaviour.

A *Client Manager* controls these *Event Handlers*, sitting between the different *Event Handlers* and a central event buffer inside AYUDAME. All messages coming

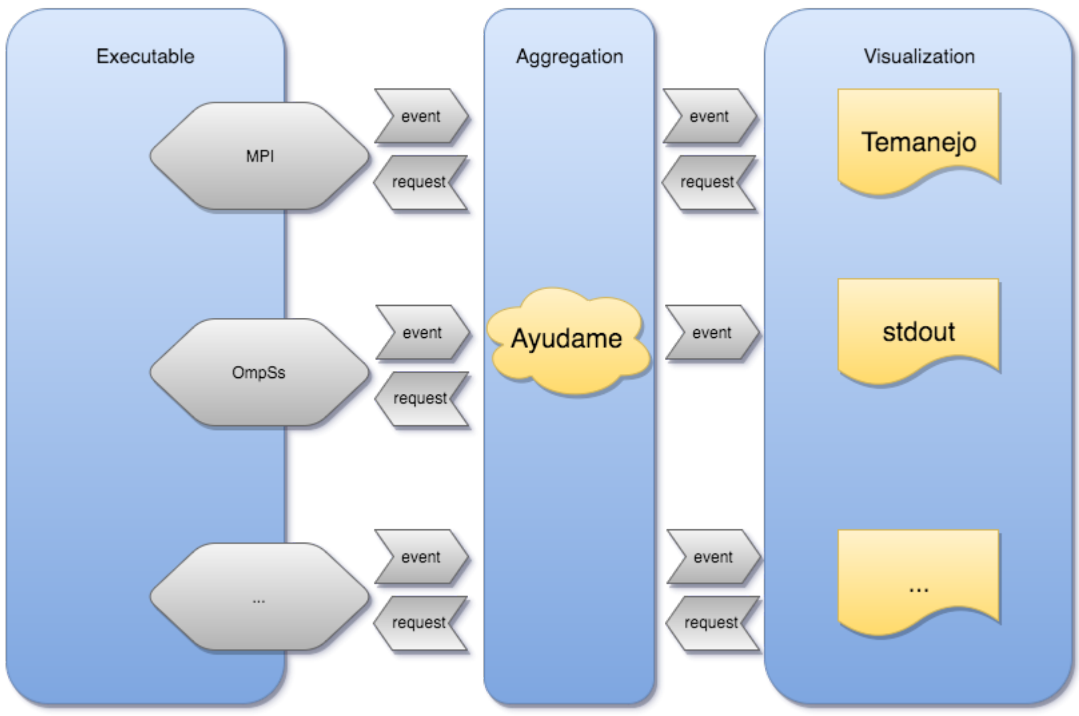


FIGURE 5.4: shows multiple runtimes (in one process) attached to AYUDAME. In addition, the possible outputs of AYUDAMEare also illustrated.

from an *Event Handler* are passing this buffer. From the event buffer an AYUDAME thread is reading the messages and sending it out either to the front-end or another AYUDAME instance. In case of a multi-process environment, a *Socket Connect Handler* is used. It is also possible to activate multiple of these *Connect Handlers* at the same time. The thread reading the information out of the event buffer is also responsible for receiving the requests from another AYUDAME instance or TEMANEJO. These requests are stored inside the request buffer. A second thread inside AYUDAME is checking the request buffer for requests, and either passing the message into the according runtime through the runtime specific *Event Handler* or, if the *Unique ID* does not belong to this AYUDAME instance, the message is forwarded through the *MPI Event Handler* to another process. The logic about the path the message has to travel is encoded in the *Unique ID* and can be calculated. This concept is outlined in Section 5.6

The *Event Manager* and *Connect Manager*, are responsible for the shutdown process. The *Event Manager* is tracking the shutdown of every runtime connected to AYUDAME. If all runtimes have sent their shutdown event, the *Event Manager* initialises the overall shutdown process. The *Connect Manager* takes care of the buffers and the messages on the wire. As long as there are messages, not transferred to their destination, the *Connect Manager* postpones the shutdown process.

5.4.1 Event Handler for specific programming models

Different programming models have the need for different *Event Handlers*. These handlers have to to take care of programming model related behaviours. The StarPU programming model is using a very basic *Event Handler* implementation, without any extensions. For MPI, OmpSs and OMPT, specialised *Event Handler* are needed:

OmpSs Event Handler

The *OmpSs Handler* is connected through an instrumentation to the OmpSs runtime. This *Event Handler* is also capable of providing the necessary information of task nesting level to the matcher. This information can then be used by any other *Event Handler* to nest a task into a task of this runtime. In case of an MPI operation inside and OmpSs task, AYUDAME is capable of providing this information to TEMANEJO. Also, the *Ompss Event Handler* uses TCA (Chapter 6.3.1) for forwarding the requests into the runtime.

OMPT Event Handler

In case of the OMPT (Chapter 6.2.2) the OMPT API is implemented inside the *OMPT Event Handler*. This handler is capable of any programming model using the OMPT instrumentation framework.

MPI Event Handler

For distributed memory applications, using MPI (Chapter 2.5), the *MPI Handler* provides three programming model related features: 1) Capability of building up the underlying connection tree (Section 5.5.1), and the routing of events and requests to their destination. 2) Implementation of a module matching the MPI operation (Chapter 5.7). This is necessary because only distributed information about the dependency is available and TEMANEJO has to combine the information. 3) For nesting MPI tasks inside other tasks, the *MPI Event Handler* can request this information through the *Client Manager*. In addition to the regular shutdown process in case of MPI, the *Event Handler* takes care of a top down shutdown. That means the connection tree is shut down from its leaves to the root.

5.4.2 Connect Handler

For the *Connect Handler* side there are following implementations available:

TEMANEJO Connect Handler

The connect handler TEMANEJO is used to connect AYUDAME with TEMANEJO. The module uses an *Ayu Socket* instance for communication.

Socket Connect Handler

The *Socket Connect Handler* is used for inter-process communications and sends or receives messages from other AYUDAME instances. Below the *Connect Handler* is using an instance of *Ayu Socket*.

XML Connect Handler

The *Connect Handler XML* is writing out the events to disc in an XML-based (Extensible Markup Language) format.

Dot Connect Handler

The *Connect Handler DOT* (text based graph description format) is writing out the events to disc in a DOT based format.

Stdout Connect Handler

The *Connect Handler std* is writing out the events directly to the terminal.

In case of a multi-process environment, only the root AYUDAME instance is configured to write to XML, Dot and Std_out. All other instances are configured to use the *Socket Connect Handler* to pass the information to the root instance.

5.4.3 Ayu Socket

Ayu Socket is a module based on libevent [26] and is used inside AYUDAME& TEMANEJO for inter process communication. The module is configurable and can act either as 1) *Server Socket* or as 2) *Client Socket*. 1) Acting as *Server Socket*, the module allows a defined number of incoming connections on a specified port. The amount is configured during the initialising process of the module. In addition, the module is bookkeeping the hostname and process ID for every opened file descriptor. Every connection has its own separate file descriptor. Using the *Unique ID* of the events & requests and the inside encoded information about hostname and process ID, the *Ayu Socket* module is capable of calculating the destination for every package. With the stored correlation between host name, process ID and file descriptor, it is possible to calculate the correct path for every package. 2) In the *Client Socket* configuration the module has to be initialised with hostname and port and sends the information about the process ID to the AYUDAME instance it is connected to. This is necessary to distinguish the file descriptor within the same host name. For both configurations the *Ayu Socket* module processes *Intern Event* (Section 5.4.4) objects.

5.4.4 Intern Event

Intern Event represents the internal data structure of AYUDAME. Every event coming from an instrumented runtime system is transformed into an object of the type *Intern Event*. In most cases, the events sent by the instrumented runtime system are AYUDAME events, and can be directly transformed into *Intern Events*. For an OMPT instrumented runtime system AYUDAME has to translate the extracted information. The same happens for requests; they are transformed from *Intern Events* into runtime requests, which are for example defined in the TCA. The following objects inherit from the *Intern Event* object: *Intern Event Dependency*, *Intern Event Property*, *Intern Event Task*, *Intern Event Userdata* and *Intern Request*. *Intern Event* also inherits from *Intern Printable*. All *Intern Event* can convert their data into XML, DOT or stdout. Also, all of the objects have the abillity to transform their information into a byte stream and to regenerate themselves out of a byte stream. This marshalling / unmarshalling process is used during the transmission between different processes.

5.5 Multi process node environment

This chapter illustrates the complexity of multi-process debugging, with several AYUDAME instances. Figure 5.5 shows the behaviour of a single node single process debugging process. The Node 0 is directly connected to the front-end. Inside this node, one process is running the executable including the application (user code), the runtime and a preloaded AYUDAME library. The runtime is representing one or multiple programming models and their instrumentations. The API calls or callback functions are marked with yellow arrows. The black arrows are socket communication which represent messages sent between two processes. Figure 5.6

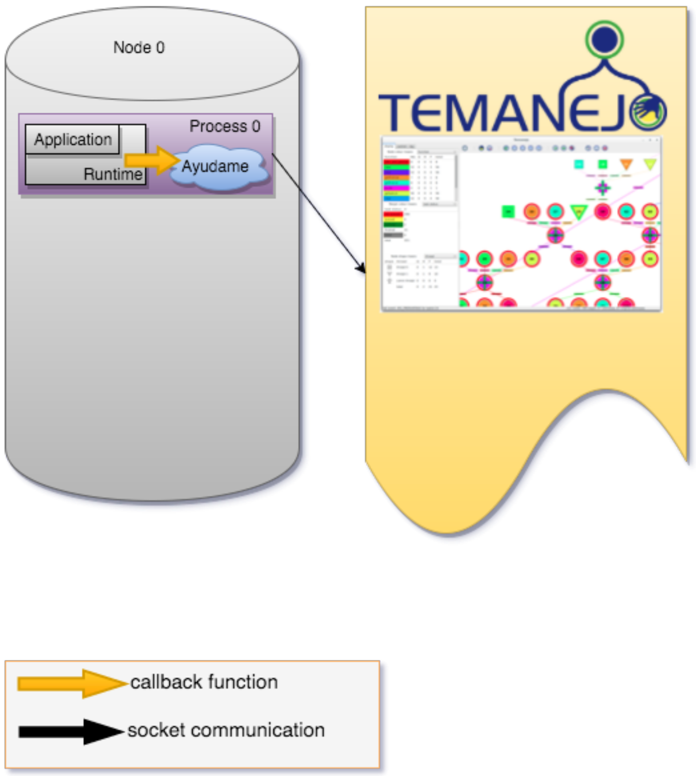


FIGURE 5.5: One Process at a node is connected to TEMANEJO.

shows the concept of multiple processes at one node and the connection between the different AYUDAME instances and TEMANEJO. All processes on the node are directly connected to TEMANEJO. Scaling it up on multiple nodes is shown in Figure 5.7 and will cause a bottleneck in TEMANEJO.

To get rid of this bottleneck, I introduced a multi-level *communication tree*, explained in detail in the section 5.5.1.

5.5.1 Connection tree

For inter-node and inter-process communication, sockets are used for the information exchange. Figure 5.8 shows the design of the *communication tree* and the connections between the different instances of AYUDAME. On every node, one process, allows all other processes on the same nodes to connect via socket communication to themselves, this process is called master AYUDAME (marked red in the Figure). The master also allows incoming connection from other nodes. Every master is then calculating the process on another node he has to connect to. The root node in the connection tree, is called super AYUDAME (marked orange

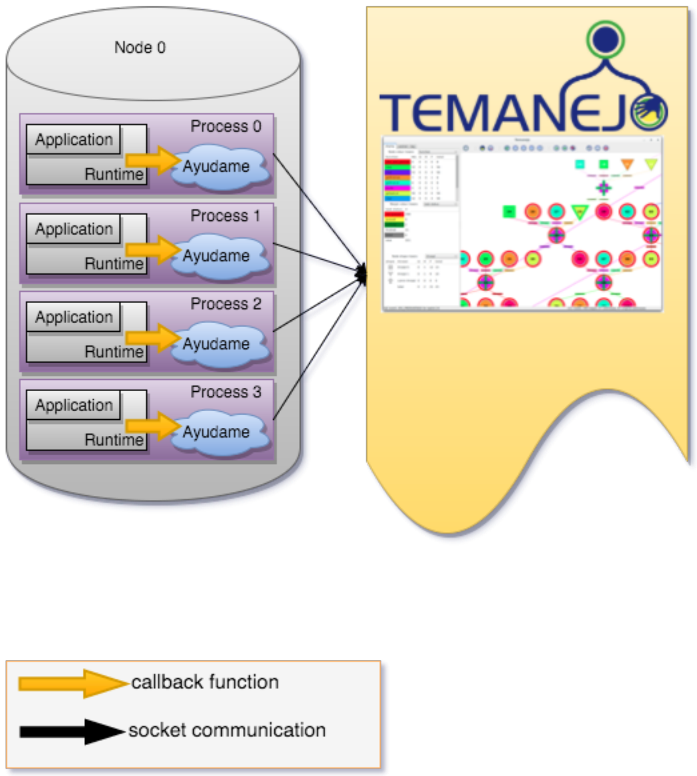


FIGURE 5.6: Multiple Process at a node are connected to TEMANEJO

in the Figure). The super AYUDAME also connects to any tool interacting with AYUDAME, in this case TEMANEJO.

Building up the *communication tree* requires several steps. The design I did for AYUDAME relays on the usage of MPI for the job submission in a distributed environment. But as long as it is possible to extract the information about the hostnames, the application runs on, the design can easily be adapted to other environments. During the *MPI Handler* initialisation, AYUDAME exchanges information about ranks and hostnames using MPI_Alltoall. After this step, every MPI rank holds the MPI hostfile (a list of all MPI ranks, hostnames) belonging to the application execution. For the MPI-specific implementation AYUDAME reduces this *MPI hostfile* to a single hostname list. This means, if multiple processes run on the same node, only one entry will be kept in the reduced list. Every MPI rank not listed in the reduced list will connect to the remaining hostname (localhost) in the reduced list. The port is specified during the configuration process of the module.

The remaining MPI ranks (reduced list) now built up a connection tree across the different nodes. AYUDAME uses the the algorithm shown in the listing of

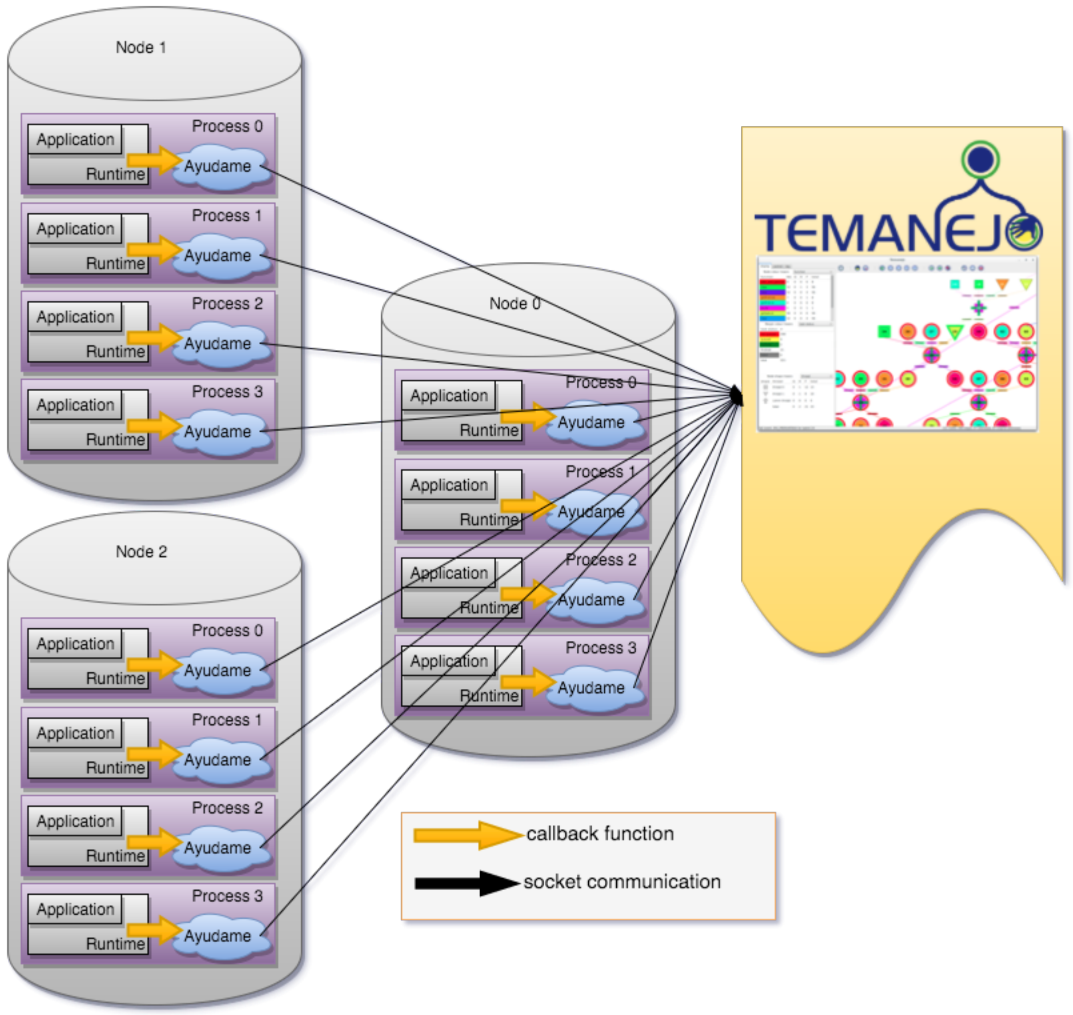


FIGURE 5.7: Multiple Process at multiple nodes are connected to TEMANEJO. The communication is spanned up in a "All to one" fashion.

Figure 5.9 to generate a tree out of the reduced list. According to this algorithm the remaining MPI ranks connect to the calculated destination.

Figure 5.10 shows the generated tree. All local connections, (intra node) are marked black. The inter node tree is marked with red lines between the nodes. The nodes spanning up this tree are called master AYUDAME nodes. At the bottom, the visualisation front-end is connected to rank 0; this node is also known as super AYUDAME. In this example, every node in the red tree has at most two incoming remote connections. This variable is adjustable and by increasing the allowed connections per node the tree's height is reduced, and the width is increased.

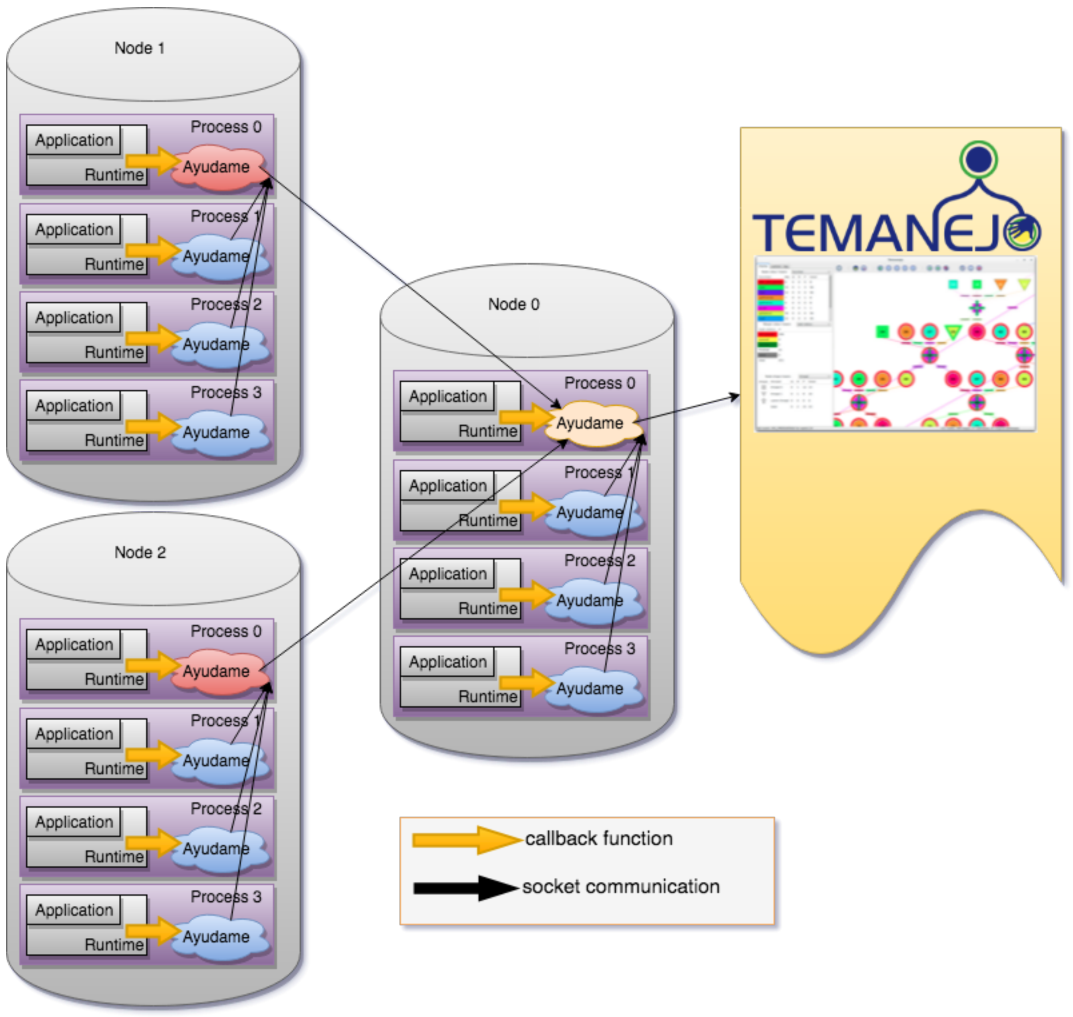


FIGURE 5.8: Multiple Process at multiple nodes are connected to TEMANEJO. The communication is spanned up as a "Communication Tree", e.g. the messages are routed through AYUDAME instances.

5.6 Identification and Information routing

5.6.1 Routing events and requests

In a shared memory environment, AYUDAME has to handle multi-process applications. Therefore, the library has to take care of the event and request routing. In case of events the routing, the message flow, is clearly defined. The direction is always from the child node towards the parent node. In cases of requests, the routing is more challenging. AYUDAME needs to be calculated, the destination for every message. This is done through recursively calling the calculate destination function. The function used the algorithm explained in section 5.5.1, spanning up the *connection tree* and is therefore, capable of calculating the parent of a given

```
#define MAX_CORES_PER_NODE 48
#define MAX_CHANNELS_PER_NODE 64
#define MAX_REMOTE_CHANNELS_PER_NODE (MAX_CHANNELS_PER_NODE
                                        - MAX_CORES_PER_NODE)

 //search for my_hostname in the reduced_list
 it = std::find(reduced_list.begin(), reduced_list.end(), my_hostname);

 //get the position of my_hostname
 index = std::distance(reduced_list.begin(), it);

 //calculate the index of the destination considering the
 //configuration variable MAX_REMOTE_CHANNELS_PER_NODE
 index = index - MAX_REMOTE_CHANNELS_PER_NODE + 1;
 destination_index = (MAX_REMOTE_CHANNELS_PER_NODE + index - 2)
                     / MAX_REMOTE_CHANNELS_PER_NODE;

 //get the destination_hostname
 std::string destination_hostname = reduced_list[destination_index];
```

FIGURE 5.9: shows the algorithm spanning up the“Communication Tree”.

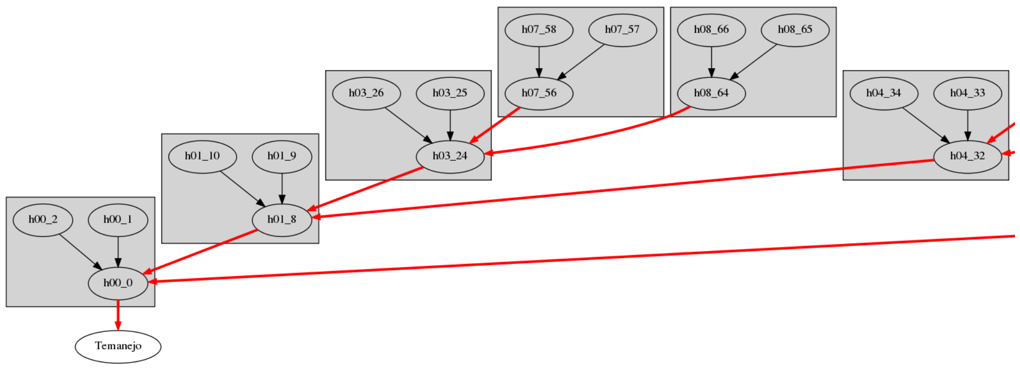


FIGURE 5.10: shows the “Communication Tree” used for multi-process debugging.

node. Outgoing from the request’s message destination, stored in the *Unique ID* (Section 5.6.2), AYUDAME calculates recursively the parent of the node. The recursive calculation is interrupted as soon as the calculated parent is the actual instance of AYUDAME. At this point AYUDAME knows to which child the message has to be forwarded to. Every further instance of AYUDAME will do the message forwarding until the message reaches its destination. In the current design all communications are either generated from an AYUDAME instance and will be consumed by TEMANEJO or they are generated by TEMANEJO and consumed by AYUDAME.

In case of inter-child communication, the algorithm can be adapted for supporting such a data flow. At the moment all messages without a destination in the current subtree are dropped. One possible adaptation could be forwarding the message to the parent node until the destination can be calculated. For the currently supported features of AYUDAME there is no need for such an design, but future features like local storage of events or graph analysis across AYUDAME instances may require inter-child communication.

5.6.2 Unique ID

Every instrumentation instance of AYUDAME is getting an *Unique ID*. Therefore, internally AYUDAME generates the *m_id*, also known as *Unique ID*, by shifting the *master_id*, *proc_id* and *client_id* into the *m_id*. To do so, three functions are used for moving the information in and three for getting the information back out. The *master_id* represents the MPI rank in a distributed environment. The *proc_id* represents the process identifier on a node. The *client_id* identifies the different programming models in a process.

The first function pair, shown in the listing of Figure 5.11, will set the first three bytes of *m_id* and gets the stored information back out. These three bytes are called *master_id* and hold the information about node/hostname the AYUDAME instance is running on. This information is needed to distinguish between the different communication back-ends in a distributed memory environment. The current implementation uses the MPI rank as master_id.

The second function pair, shown in the listing of Figure 5.12, sets the forth to the sixth byte of *m_id* and gets the stored information back out. These bytes are called *proc_id* and they hold the information about the process the AYUDAME instance is running in. This information is needed to distinguish between the different communication back-ends in a shared memory environment running on the same node.

The third function pair, shown in the listing of Figure 5.13, sets the seventh to the eighth byte of *m_id* and gets the stored information back out. These bytes are called *client_id* and they contain the information about the *Event Handler*. This is necessary to distinguish the different handlers in a hybrid environment. Every programming model in such a hybrid environment has its handler, with its associated implementation.

```
/**
 * Sets the first three bytes of m_id
 * @param m_id
 * @param master_id
 */
static void set_master_id(uint64_t *m_id, uint64_t master_id) {
    *m_id |= master_id << 0;
}

/**
 * Returns the first three bytes of m_id
 * 0000000000000000000000000000000000000000111111111111111111111111
 * @param m_id
 * @return
 */
static uint64_t get_master_id(uint64_t m_id) {
    return (m_id & 0xFFFFFF) >> 0;
}
```

FIGURE 5.11: The function set_master_id shifts the master_id into the m_id, the second function returns the master_id out of the m_id

```
/**
 * Sets the 4. to the 6. byte of m_id
 * @param m_id
 * @param proc_id
 */
static void set_proc_id(uint64_t *m_id, uint64_t proc_id) {
    *m_id |= proc_id << 24;
}

/**
 * Returns the 4. to the 6. byte of m_id
 * 0000000000000000111111111111111111111111000000000000000000000000
 * @param m_id
 * @return
 */
static uint64_t get_proc_id(uint64_t m_id) {
    return (m_id & 0xFFFFFF000000) >> 24;
}
```

FIGURE 5.12: The function set_proc_id shifts the procc_id into the m_id, the second function returns the procc_id out of the m_id

5.7 MPI Matcher

The *MPI Matcher* is a module located inside the super AYUDAME. It generates the communication dependencies between the MPI tasks. The necessary information (generated by the programming model) for generating such dependencies is transferred as payload (Section 4.2.2) into the module. The *MPI payload* contains

```
/**
 * Sets the 7. to the 8. byte of m_id
 * @param m_id
 * @param client_id
 */
static void set_client_id(uint64_t *m_id, uint64_t client_id) {
    *m_id |= client_id << 48;
}

/**
 * Returns the 7. to the 8. byte of m_id
 * 1111111111111111000000000000000000000000000000000000000000000000
 * @param m_id
 * @return
 */
static uint64_t get_client_id(uint64_t m_id) {
    return (m_id & 0xFFFF000000000000) >> 48;
}
```

FIGURE 5.13: The function set_client_id shifts the client_id into the m_id, the second function returns the client_id out of the m_id

the information about the *MPI type*, *MPI rank*, *MPI partner*, *MPI tag*, *MPI comm* and the *task ID*. For every payload the module tries to find the matching payload (stored in a list) inside, if the there is no, the payload will be added to the list; e.g. for a *MPI_Send*, the module tries to find the corresponding *MPI_Recv*. For a *MPI_Recv*, the matcher searchers for the *MPI_Send*, where the *MPI tag* and the *MPI_Recv rank* is the same as the *MPI_Send* partner. By using the *task ID* and *client ID*, which is also stored in the payload (because of the inheritance), AYUDAME is now capable of producing the data dependency between the *MPI_Send* and *MPI_Recv* running on different nodes or processes. The same design is used for *MPI_Isend* and *MPI_Irecv* considering the *MPI_Wait*.

Chapter 6

Instrumentation of Runtime Systems

6.1 Introduction

This chapter briefly describes the usage of the instrumentation framework of Ayudame. An instrumentation is able to monitor and control an application. This is in the case of my thesis done by instrumenting the source code. There are also other tools which use binary instrumentation. In general an instrumentation is necessary for generating addition information out of an application execution; e.g. debugging, code tracing, profiling, measuring the performance counters, etc..

The instrumentation is the part of Ayudame, which is connected closest to the programming model and its runtime is the instrumentation. The instrumentation translates runtime information into Ayudame events for monitoring (Section 6.2). Additionally, the instrumentation converts Ayudame requests into runtime behaviour for controlling (Section 6.3). Figure 6.1 illustrates the four different ways (Event+InCode, Event+API, Request+InCode and Request+API) of instrumentation. Some programming models implement a defined API to have a general programming model independent interface for monitoring and controlling the runtime. The API based implementation is represented by OMPT (Section 6.2.2) and TCA (Section 6.3.1) in the Figure 6.1. Other programming models have to be directly instrumented with the Ayudame API (InCode Instrumentation). InCode means that the monitoring events and the controlling request are directly implemented in the runtime.

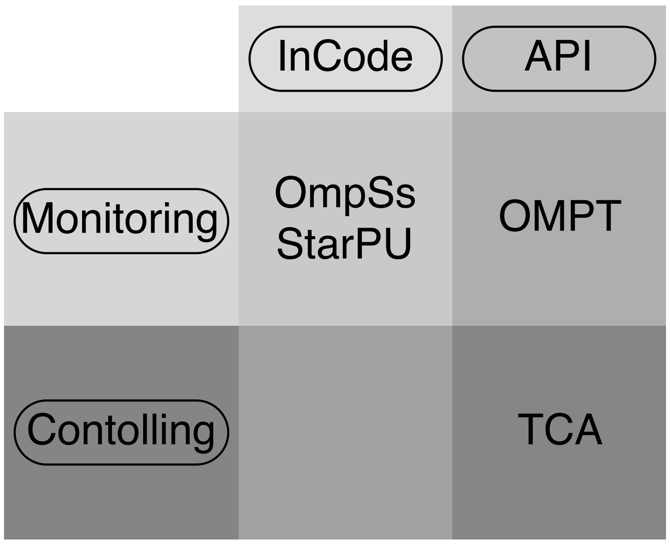


FIGURE 6.1: shows the different possible AYUDAME instrumentations. Some programming models need an InCode instrumentation, others can be accessed through an API.

6.2 Monitoring

For the monitoring purpose the C or C++ events API, explained in section 4.2, is used inside the runtime. Section 6.2.1 shows, as an example, the usage of the AYUDAME C interface in OmpSs. In addition, AYUDAME alternatively interact with the OMPT Interface, which is also offered by OmpSs (the InCode instrumentation supports more functionality than the OMPT OmpSs instrumentation) and OpenMP. Therefore, AYUDAME has the OMPT *Event Handler*.

6.2.1 OmpSs instrumentation

This section gives a brief introduction to the InCode OmpSs monitoring (event) instrumentation. The instrumentation is done inside the OmpSs runtime and can be loaded by setting an environment variable. If this variable is set, the OmpSs runtime (nanox) opens the correct AYUDAME instrumentation library. Now, every event generated by the nanox runtime reaches the library. This library has to implement, amongst others, the following functions: 1) *addEventList*, 2) *addResumeTask*, 3) *addSuspendTask*, 4) *initialize* and 5) the constructor *InstrumentationAyudame*. The runtime now calls for every nanox event (generated by the runtime), one of the above listed functions. Inside these functions the nanox events are filtered, analysed and transformed/ translated into AYUDAME events. To do

```
void addEventList ( unsigned int count, Event *events ) {
  for (unsigned int i = 0; i < count; i++) {
    switch ( events[i].getType() ) {
      case NANOS_POINT:
      if ( events[i].getKey() == ``create_wd_ptr'' ) {
        /* It is a create task event */

        WorkDescriptor = events[i].getValue();
        addTask(wd);
      }
      if ( events[i].getKey() == ``dependence'' ) {
        /* It is a dependence event */

        dependence_value = events[i].getValue();
        int sender_id = ( dependence_value >> 32 );
        int receiver_id = ( dependence_value & 0xFFFFFFFFFF );
        dep_direction_value = events[i+1].getValue();
        dep_address_value = events[i+2].getValue();
        addDependency(sender_id, receiver_id, dep_direction_value,
                                              dep_address_value);
      }
      break;

      case NANOS_BURST_START:
      if ( events[i].getKey() == ``wd_id_event'' ) {
        /* It is a entering WorkDescriptor event */

        enteringWD(events[i].getValue());
      }
      if ( events[i].getKey() == ``user_code'' ) {
        /* It is a entering User code event */

        enteringUserCode(events[i].getValue());
      }
      break;
    } // switch end
  } // for end
}
```

FIGURE 6.2: shows the "addEventList" code part, used for instrumenting OmpSs. The function splits the different runtime events into "addTask", "addDependency", etc.

so the library has to store and process information. The events reach the instrumentation library either as an event chain (*addEventList*) or as a direct function call. The following sections show code snippets of the above listed function and the most relevant functions for generating the dependency graph, are explained.

1) The function *addEventList* (listing in Figure 6.2) processes the event chains. The function parameters are the number of events and a pointer to the first event. Inside the *addEventList* functions, the instrumentation checks the event chain for

```
void addResumeTask( WorkDescriptor &w ){

  WorkDescriptor *parent = w.getParent();

  if (parent == NULL) {
    /* Task is a thread */

    // add the thread ID to a vector
    threadid_vec.push_back(thread_ID);

  }else{
    /* Task is a an actual task */

    // Thread ID is added as task property
    ayu_event(AYU_SETPROPERTY, thread_ID);

    // check if the task can be executed
    stepper::stepper_request_progress(task_ID , thread_ID) ;

    // Task state running is added as task property
    ayu_event(AYU_SETPROPERTY, ''running'');
  }
}
```

FIGURE 6.3: shows the "addResumeTask" code part, used for instrumenting OmpSs. The function detects the tasks states and generates an AYUDAME event.

specific events. The most important events are of the event type *NANOS_POINT*. Important are also the events of the event type *NANOS_BURST_START* and the corresponding type *NANOS_BURST_END*.

The event type *NANOS_POINT* detects the events for task creation and dependency creation. For task creation, the *addTask* (line 9 listing in Figure 6.2) function is called with the according *WorkDescriptor*, which is returned by getValue. The *addTask* helper function (listing in Figure 6.4) collects all required information for the AYU_ADDTASK event, line 27, and sends the event to AYUDAME. Therefore, it is, necessary to calculate the the scope ID, by checking if the task has a parent task. Additionally, the *addTask* function also sends AYU_SETPROPERTY events to AYUDAMEwhich contains information like the function names or priorities. Most of the information can get extracted out of a *WorkDescriptor* object provided by the runtime.

If a dependency generation is detected the *addDependency* helper function is called. Therefore, the *sender ID* (*from ID*) and *receiver ID* (*to ID*) have to be shifted

```
void addTask( WorkDescriptor *wd) {

  // Task ID
  int64_t wd_id= wd->getId();

  // Function ID
  int64_t funct_id =  wd->getActiveDevice().getWorkFct();

  // Function name
  char *wd_description = wd->getDescription();

  // Parent task
  WorkDescriptor *parent = wd->getParent();

  // Priority
  int64_t priority = wd->getPriority();

  if (parent->getParent() == NULL) {
    /* not nested scope ID 0 */
    scope_id = 0;
  } else {
    /* nested scope ID = parent task ID*/
    scope_id = parent->getId();
  }

  // Task with task ID = wd_id, scope ID= scope_id gets added
  ayu_event(AYU_ADDTASK, data);

  // Function name is added as task property
  ayu_event(AYU_SETPROPERTY, wd_description);

  // Priority is added as task property
  ayu_event(AYU_SETPROPERTY, priority);
}
```

FIGURE 6.4: shows the “addTask” code part, used for instrumenting OmpSs. The function generates an AYUDAME task creation event.

out of the *dependence value*, which is returned by getValue. The function *addDependency* (listing in Figure 6.5) collects all required information for the AYU_ADDDEPENDENCY event, line 21, and sends the event to AYUDAME. This function is also the most complex one; which is related to the different possible dependency types (dep_direction_value). This chapter will only handle the creation of a regular dependency (true dependency) type. Besides; dependencies coming from a thread ID are filtered out. The dependencies from a thread are synchronisation dependencies (#pragma OMP TASK WAITON). There are some other dependency types known as *Output dependence wd* → *concurrent*, *wd* → *commutative* or *common* → *wd*, these types are relevant for concurrent or commutative dependencies. For a usual case, a dependency can be created and added to the task

```
void addDependency(int sender_id, int receiver_id,
                   nanos_event_value_t dep_direction_value,
                   nanos_event_value_t dep_address_value) {

  switch (dep_direction_value) {

    case 1:
    /* True dependence */

    // Dependency memory address
    int64_t address_id = dep_address_value;

    // Dependency ID generation
    uint64_t dep_id = 0;
    set_sender_id(&dep_id, (uint64_t)sender_id);
    set_receiver_id(&dep_id,(uint64_t)receiver_id);
    set_dep_adrress_id(&dep_id,dep_address_value);

    // Dependency with dependency ID = dep_id, from_id= sender_id
    // and to_id=receiver_id gets added
    ayu_event(AYU_ADDDEPENDENCY, data);
    ayu_event(AYU_SETPROPERTY, ``addresse'');

    break;
  }
}
```

FIGURE 6.5: shows the "addDependency" code part, used for instrumenting OmpSs. The function generates an AYUDAME dependency creation event.

graph by just using the sender ID and receiver ID. The unique dependency ID is generated out of the sender ID, receiver ID and the underlying memory address of the dependency. All additional dependency information is added as a dependency property.

The *NANOS_BURST_START* event type calls the function *enteringWD* for entering a WorkDescriptor or *enteringUserCode* for an User Code area. The event type *NANOS_BURST_END* calls the corresponding functions for leaving a *WorkDescriptor* or User Code area. The function *leavingUserCode* is important for AYUDAME. Inside this function the instrumentation can detect if the runtime leaves the main function of the application. If this is true, the instrumentation will inform AYUDAME about the upcoming shutdown process.

2) The function *addResumeTask* (listing in Figure 6.3) first checks if the *WorkDescriptor* (task) is an actual task or if the *WorkDescriptor* is, in reality, a thread.

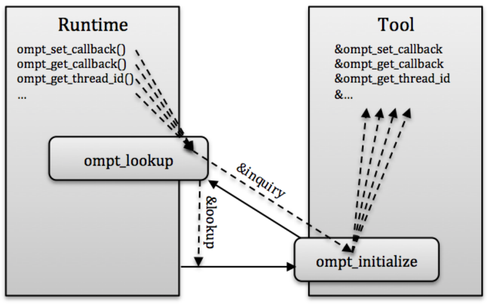


FIGURE 6.6: Stage one of the OMPT initialization

This is done by calling the *getParent* method, if the value returned is *NULL*, the *WorkDescriptor* is, in reality, a thread, and is added to a vector. This information is, later on, necessary for dependency creation. If the return value of *getParent* is not *NULL*, an AYUDAME event will be emitted, with the information, that a task with a given task ID is now running on a specific thread. At this point, the instrumentation library can also extract the information about the executing core, NUMA domain, etc..

3) Corresponding to the function *addResumeTask*, the *addSuspendTask* emits an event about the finalisation of a specific task.

4) The function *initialize* adds meta information about the application to the dependency graph. This information is for example necessary to start a DDT debugging session and contains the hostname, the process ID, the name of the executable and the path to the executable.

5) The constructor *InstrumentationAyudame* initialises the storage and locking units inside AYUDAME.

6.2.2 OpenMP Tools Application Programming Interfaces for Performance Analysis and Debugging - OMPT

OMPT([27]) is an interface for performance analysis and debugging for the OpenMP programming model. For OmpSs, which also supports OMPT and AYUDAME, the OMPT runtime-side implementation is done inside the AYUDAME *OMPT Event Handler* and is based on the technical report (tr2). This *Event Handler* implements the tool side of OMPT (Section 6.3.1). The initialisation of OMPT can

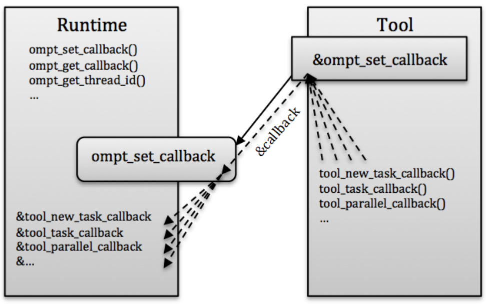


FIGURE 6.7: Stage two of the OMPT initialization

be split into two stages. In the first stage, the runtime calls an extern public initialisation function, implemented inside the tool. The lookup function pointer is passed as parameter to the initialisation function. Now the tool can ask for the different control request callbacks through the lookup function. As parameter for the lookup function a char pointer is passed. Depending on this string the lookup functions return the requested inquiry function. This concept is illustrated in Figure 6.6. Figure 6.7 shows the workflow of the second stage. There the tool instantiates callbacks for the different events in the runtime. This is done through the formerly requested inquiry function (ompt_set_callback). The tool can now repeat this procedure to register all the events it wants to track.

6.3 Controlling

For controlling, the programming model has to implement the AYUDAME request callbacks, explained in section 6.2. Such a callback function has a defined signature and can handle a given AYUDAME request, e.g. block a particular task. For controlling, AYUDAME also includes an implementation of TCA. The TCA interface is developed in addition to the OMPT interface and shown as an example for controlling runtimes in section 6.3.1.

6.3.1 Tasking Control API - TCA

The specification of Tasking Control API (TCA) is published in [28]. TCA is a generic interface to interact with the runtime system of task-based programming models. The interface defines a small set of control requests, which need to be implemented by a conforming runtime system. These requests allow a tool, for instance a debugger, to instruct the runtime system to suspend or resume task execution and to manipulate task dependencies. TCA has been designed to inter-operate OMPT and is currently used by the back-end of the debugger TEMANEJO to interact with the OmpSs runtime system. The interface, however, is sufficiently generic and can be supported by other similar task-based programming models. As TCA is designed as an extension of OMPT and is, at the moment, implemented and used inside AYUDAME. But the idea is to let TCA become a part of the OMPT instrumentation.

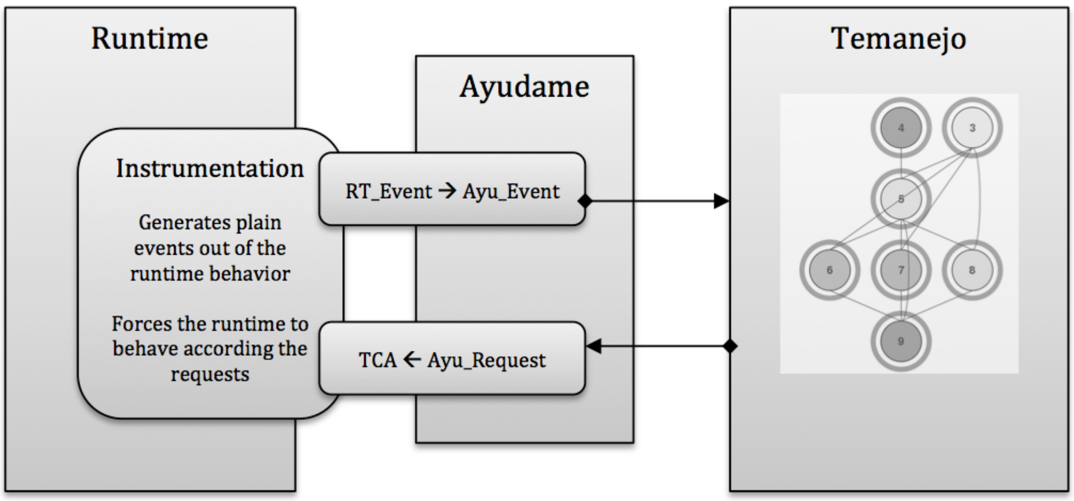

Runtime
Ayudame
Temanejo
Instrumentation
Generates plain events out of the runtime behavior
Forces the runtime to behave according the requests
RT_Event → Ayu_Event
TCA ← Ayu_Request

FIGURE 6.8: Interaction of TEMANEJO with a runtime system through TCA

TCA is used as described to connect TEMANEJO, or more specifically its back-end AYUDAME, with the runtime of the task-based parallel programming model OmpSs. Figure 6.8 shows the general structure of information flow. In the current prototype, the OmpSs runtime has been instrumented to emit plain (InCode) AYUDAME events (Section 4.1). In future it is planned to use the OMPT interface (API) of OmpSs to generate AYUDAME events. AYUDAME events are processed and ultimately forwarded to the front-end to visualise the task dependency graph. Part of the information transmitted with the events is identifiers for dependency and task instances. These identifiers are later on used in TCA.

In the other direction, user interaction in TEMANEJO results in AYUDAME control request (Section 4.6). So far, these control requests have been used to drive a *Simple Stepper* (Section 6.4) inside AYUDAME, which either suspends or resumes worker threads of the OmpSs runtime. Instead, the AYUDAME library will now generate TCA control requests from AYUDAME. I have developed a plugin module for OmpSs, which initialises TCA. The module also provides all mandatory and optional TCA control request functions and delegates all possible AYUDAME control actions to the OmpSs runtime system.

The services inside the OmpSs runtime do not allow to implement all TCA requests. For instance, there is no interface to prevent the scheduler from executing a specific task as required by `tca_request_block_task`. As a work around until the functionally is added to the OmpSs runtime, the *Simple Stepper* was moved out of the AYUDAME library and into the OmpSs plugin. This prototypical implementation is able to fully control the execution of an OmpSs application.

Functionality

The purpose of TCA is to allow a tool to request certain actions from a conforming runtime system including:

- suspend and resume execution of a specific, labelled tasks
- insert and delete a dependency between two specific, labelled tasks

Some requests are mandatory, thus providing a minimal functionality to the tool, while others are optional.

Examples for TCA requests

The below listed examples show two mandatory requests (tca_request_continue, tca_request_break) and one optional request (tca_request_insert_dependency). These requests are defined in TCA.

- **Continue control request** **mandatory**

  ```
  tca_success_t tca_request_continue();
  ```

  Instructs the runtime to continue normal execution of tasks until a breakpoint is reached or until a *break* control request is issued. This request has no arguments.

- **Break control request** **mandatory**

  ```
  tca_success_t tca_request_break();
  ```

  Instructs the runtime system to stop executing any task until a *continue* or *step* control request is issued. This request has no arguments.

- **Insert dependency control request** **optional**

```
tca_success_t tca_request_insert_dependency(
                        tca_task_id_t from_task_id,
                        tca_task_id_t to_task_id);
```

Instruct the runtime system to insert a dependency from task with identifier `from_task_id` to task with identifier `to_task_id`. The runtime system should not execute task with identifier `to_task_id` before task with identifier `from_task_id` has finished.

Usage

The foremost design objective is providing sufficient functionality to allow the debugger Temanejo to step through an application task-wise. A second major design objective is interoperability with OMPT. In particular, the design should meet two requirements:

- the procedure to initialise TCA should be the same as for OMPT
- do not introduce any concepts beyond those already present in OMPT, i.e. tool callbacks, runtime inquiry functions, tool data structures, etc.

Any runtime supporting OMPT should be able to support TCA, as long as it can provide the minimal set of TCA request. The design of TCA, however, is sufficiently generic to be applied with any runtime system for asynchronous task parallelisation.

It is assumed, that the programming model meaningfully defines the concept of task and the concept of dependency. A program is composed of task instances and dependency instances between them, thus forming a task-dependency graph. A unique identifier can label task and dependency instances. Note, that the same dependency instance, and thus identifier, may appear more than once at different positions in the task-dependency-graph.[1] This might be the case when dependencies arise through data-dependencies; e.g. the same datum can be an input dependency for several tasks. In addition, it is assumed that the runtime system executes tasks on the same kind of uniquely labelled execution resource. TCA

[1]The rest of this section will use task and dependency instead of task instance and dependency instance, respectively. Similarly, a task-graph or graph is referred to task-dependency graph.

uses the name thread for this concept; however, this does not imply that actual OS threads need to be used by the concrete runtime system. As with tasks and dependencies, threads have a unique identifier that can be used to identify it. The purpose of TCA is to send control requests from a tool to a runtime system, only. It does not provide means to monitor the runtime system or inquire its state. This needs to be done through a separate channel – for instance through event callbacks in OMPT, or any other monitoring interface. In particular, the monitoring system needs to report task, dependency and thread identifiers, respectively.

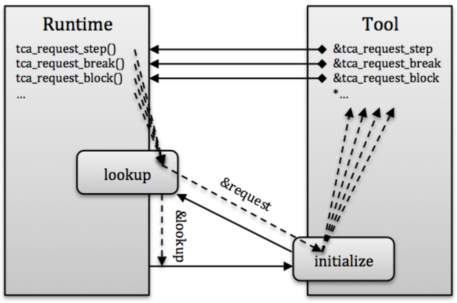


FIGURE 6.9: Interaction between the runtime and the tool, through TCA

Some aspects of the TCA are mandatory for conforming runtime systems or tools and thus form a minimal set allowing basic control. In addition, TCA defines a range of optional features that allow more fine-grained and complex control. The specification of TCA is composed of:

- Control requests
  Control request functions are implemented by the runtime and called by the tools. These functions are not publicly visible. Instead, tools can acquire pointers to control request functions through the lookup mechanism during tool initialisation. Each control request function returns an error code as defined in the previous section. Some control requests are mandatory to ensure minimal tool functionality, others are optional.

- Initialisation
  Any tool using TCA needs to follow a specific initialisation procedure as described in this section. The general principle (as illustrated in Figure 6.9) is simple: the runtime calls a initialisation function implemented by the tool. Within this function, the tool repeatedly calls a lookup function, which is provided by the runtime system, to inquire pointers to any control request function it wishes to use. If initialisation fails, for instance because the

runtime does not support a control request which is critical for the tool's operation, the tool returns with an error code from the initialisation function. The public interface of TCA is very slim; the only visible symbol is the initialisation function `tca_initialize`, which has three parameters. A pointer to the `lookup`, the `runtime_version` and the `tca_version`. All other symbols, i.e. control request functions, are publicised during TCA initialisation through the lookup mechanism described below. The second argument, `runtime_version`, is a version string that unambiguously identifies a runtime system's implementation. The third argument, `tca_version`, indicates the version of the tasking control API, supported by the runtime. The version of TCA described by this document is known as version 1. The first argument, `lookup`, is a pointer to the `lookup` function. This `lookup` function is provided by the runtime system. The tool repeatedly calls `lookup` for every `entry_point`, i.e. control request function. If the named entry point is available, the lookup function returns a pointer to it. Otherwise, the `NULL` pointer is returned. In general, the runtime system should start the initialisation procedure as soon as possible during its on startup procedure and before any user code is executed. If the tool initialisation is unsuccessful, i.e. if the tool does not return with `TCA_SUCCESS`, the runtime system does not need to maintain any information or state to support tools.

All additional information like the tool data structure can be found in "Tools for High Performance Computing 2015"[28].

Implementation

In this section I sketch out some aspects of the TCA implementation on the tool side as well as on the runtime system side. For illustration I will use the OmpSs task-based programming model and the tool Ayudame. For readability, some of the necessary explicit cast operations are ignored, in particular on function pointers. The initialisation function implemented by the tool could look like then listing in Figure 6.10. First, the tool checks if the TCA version used by the runtime system and the tool, respectively, match. If it does not, the initialisation is immediately aborted, returning `TCA_FAIL`. The next step is to retrieve pointers to several control request functions and store them in variables that are visible to the rest of the tool. Some request functions are critical for tool operation; in this example it is just the mandatory set. If any of the critical control requests

```c
#include "tca.h"

tca_success_t tca_initialize(tca_function_lookup_t lookup,
      const char *rt_version, unsigned int tca_version) {

  if (TCA_VERSION != tca_version) {
    dprint("Incompatible TCA version\n");
    return TCA_FAIL;
  }
  ayu_req_continue = lookup("tca_request_continue");
  ayu_req_break = lookup("tca_request_break");
  ayu_req_step = lookup("tca_request_step");
  ayu_req_run = lookup("tca_request_run_task");

  if ((NULL == ayu_req_continue) || (NULL == ayu_req_break) \
        || (NULL == ayu_req_step)) {
    dprint("Minimal requirements are not met\n");
    return TCA_FAIL;

  if (NULL == ayu_req_run) {
    dprint("Doing special treatment for run_task request\n");
    special treatment();
  }
  // all good
  return TCA_SUCCESS;
}
```

FIGURE 6.10: shows the tca_initialize implementation inside the tool.

are not implemented by the runtime system, the tool again aborts with a failure. Other control requests are not critical to tool operation, but require special action if not present. In the example it is `tca_request_run_task` that requires a special treatment.

```c
void * lookup(const char *entry_point) {

  if (0 == strncmp(entry_point, "tca_request_step")) {
    instrument_step();
    return &ompss_request_step;
  }

  // more control requests
  if (...) {...}

  // at this point the control request is unknown
   return NULL;
}
```

FIGURE 6.11: shows the lookup implementation inside the runtime.

The runtime system needs to implement a lookup function similar to the listing in Figure 6.11 The lookup function compares the content of `entry_point` to the list of available request functions. If it finds a match, the runtime might need to record, that the tool will use this control request, and then returns a pointer to the request function. If no match is found, the particular request function is not implemented and the lookup returns a `NULL` pointer.

```
#include "tca.h"

InstrumentationAyudame() {
  // many other things to do

  // TCA init
  if (tca_initialize) {
    tca_init = tca_initialize(lookup, ompss_version, TCA_VERSION);
    if (TCA_SUCCESS != tca_init) {
      dprint("Failed to initialize TCA. Not instrumenting.\n");
      instrument_none();
    }
  }

  // other things
}
```

FIGURE 6.12: shows tca_initialize process inside the runtime.

The runtime system also has to ensure that the TCA initialisation function is called during startup, shown in listing of Figure 6.12. If initialisation of the tool fails, the runtime system memorises the failure in order to avoid unnecessary instrumentation, which might interfere with normal operation and may cause performance issues. Finally, the runtime needs to make sure that scheduling of tasks, etc., honours the control requests issued by the tool. This could, for instance, be accomplished by implementing a finite state machine as part of the scheduler. For controlling OmpSs the constructor *InstrumentationAyudame* (described in Section 6.2.1) additionally initialises TCA, by calling tca_initialize (lookup,"NANOS++", TCA_VERSION).

It is also necessary to hook up the *Simple Stepper* (Section 6.4) inside the OmpSs runtime. Therefore, the function *addResumeTask* (listing in Figure 6.3) has to additionally execute the stepper::stepper_request_progress (task_ID, thread_ID) function (line 18). This function verify if the task_ID or thread_ID is allowed to be executed. The *Simple Stepper* is driven by TCA.

6.4 Simple Stepper

The *Simple Stepper* is a module inside AYUDAME. This module determines if a thread is allowed to return immediately from an AYUDAME event (and continue processing user tasks), or otherwise forced to busy loop inside AYUDAME (and thus break at certain tasks). The drawback of this approach is the assumption that threads issuing AYUDAME events are those, which execute user tasks. This assumption is only valid for OmpSs, but not in general. The *Simple Stepper* is capable of handling the state of individual tasks and threads; *block* or *unblock* a particular task or thread. The overall execution progress can be set to 1) *continue*, 2) *break* or 3) *step*.

1) The *continue* progress will only stop the execution, if tasks are blocked by an arbitrary dependency or an explicit blocked task or thread. 2) The *break* progress stops any task execution. 3) In addition, then *Simple Stepper* has the *step* progress, which can be triggered by a step request. The module is also able to process an *add dependency* or remove *dependency* request. The *add dependency* request is implemented by blocking the successor tasks of a given task. Only these arbitrary dependencies can be removed with the *remove dependency* request. The *Simple Stepper* internally keeps the sequential order of the incoming threads and releases them for stepping purpose in the same way; First In – First Out (FIFO). The FIFO method ensures the regular task order while debugging.

Chapter 7

Case Study: Performance Debugging

7.1 Introduction

This chapter shows the improvement of developing and optimising an application with the usage of a *Model-Centric debugging* tool (Temanejo). Temanejo was heavily used during the application optimisation process, for checking the dependency correctness and detecting performance relevant issues. Before beginning to analyse the different optimisation steps, I will give a brief introduction to the Lattice-Boltzmann Code (LBC), used for this case study. LBC is a solver for the Boltzmann Equation using the Lattice-Boltzmann Method (LBM) and is written in Fortran. This discretisation describes the time-evolution of the phase-space density (fluid or an ensemble of particles) through advection and collisions. The concept of the numerical model is very simple: basic stencil operations solve a single equation without relying on complex matrix operations or an iterative process. The LBM method can be implemented in a highly efficient manner using vector operations. This makes it very suitable for both, vector architectures like the NEC SX-ACE or today's SIMD units as Intel's SSE, AVX, AVX-2 or AVX-512 and ARM's NEON 128-bit architecture extension.

As explained in chapter 2.6 the *hybrid programming model* approach improves the scalability of applications. For the implementation of LBC, the hybrid programming model approach is followed. The domain decomposition is similar to the decomposition of the example in chapter 1.6. The first level domain decomposition divides the simulation domain into subdomains for every MPI rank. The second level domain decomposition splits the subdomain into tiles. Each tile can be executed independently from the other tiles and can be seen as an independent task. The task execution is only restricted by the finished execution of the task (tile) itself and all his neighbouring tasks (tiles) of the previous iteration (iteration n-1).

```
do timestep=1, timesteps

        call MPI_exchange() !possible communication task

        do iTile=1, nOuterTiles
                call compute(iTile) !possible outer tile task
        end do

        do iTile=1, nInnerTile
                call compute(iTile) !possible inner tile task
        end do

        call boundery() !possible task

end do
```

FIGURE 7.1: *default* code structure

The listing in Figure 7.1 shows the LBC pseudo code. The outermost loop is iteration over the time steps. Every time step, one task (MPI_exchange) is created, handling the communication with his neighbouring subdomains. A second loop inside the iteration loop generates the tasks for all outer tiles. These outer tile tasks depend on the MPI_exchange task from the previous iteration. The third loop is also located inside the iteration loop and generates the task for all inner tiles. The boundary exchange task, is the last necessary task needed for parallelising the LBC application and, is dependent on all outer tile tasks.

The standard paradigm for hybrid parallel programming in HPC is MPI+OpenMP, but as a good alternative with some additional features and a strong tasking concept OmpSs is used for the parallelisation of the LBC. This chapter gives a detailed overview of the necessary steps needed to implement an efficient parallel application using MPI+OmpSs. As mentioned earlier the first and second level domain decomposition is already implemented in LBC.

In the following sections the below-listed terms will be used, therefore I will briefly describe them:

- *Communication Hiding*:
  The hiding computation and communication technique can improve the application performance, especially on architectures with a slow network or in a communication intensive application. The feature can only be used in a hybrid application (e.g. MPI+OpenMP), decomposing its tasks into tasks relevant for communication and tasks without a communication dependency. The communication tasks can now be hidden inside the tasks without a communication dependency. Therefore, normally task priorities have to be introduced. These priorities help the runtime to execute tasks on the critical path.

- *Communication Overlap*:
  Using a programming model specific feature of rescheduling tasks gives the possibility of executing a computation task while the communication tasks are waiting for messages. Hiding the communication latencies requires the communication hiding technique.
  Example: The communication task is sending its messages to all relevant neighbouring ranks, but has to wait until the communication is done inside the MPI library. The time between the actual send and the message reaches the receiver is called communication latency in the optimisation chapter. Interrupting the communication task and running computation tasks meanwhile, increases the performance.

- *Iteration Overlap*:
  This technique can be used in a task-based programming model, where every compute task only is dependent on its neighbouring tasks. As soon as these neighbouring task finished their computation for the current iteration (iteration n), the task can start executing its computation for the next iteration (iteration n+1). Although there are compute tasks and communication tasks left from the current iteration (iteration n). This allows to expand the time available communication.

7.2 Optimization

LBC follows the pattern of a regular stencil-code with communication only via ghost cells across next-neighbours of the domain decomposition. At the highest

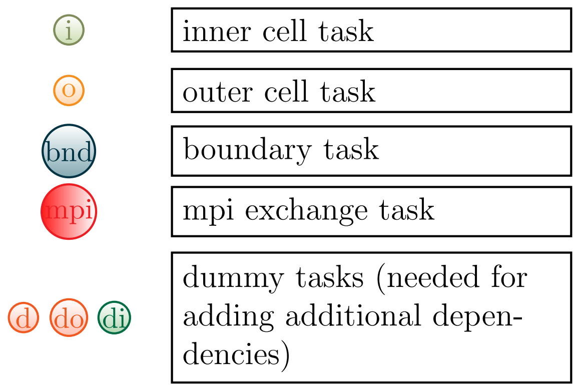


Figure 7.2: Task colouring and their meaning.

level, the parallelisation strategy was to implement a second-level domain decomposition at finer granularity than MPI domains and distribute the computation of this sub-domains, called tiles in the following, as tasks. Further tasks arise from the encapsulation of MPI communication, other book-keeping, etc. Various versions of the code arise from different approaches of expressing dependencies between these tasks. The expressed goal was to modify the original code as little as possible and do most of the parallelisation with OmpSs pragmas. The porting and evaluation started with three different distinct hybrid MPI+OmpSs versions on the Cray XC40 system. For additional evaluation, the three versions of LBC were also ported to the Mont-Blanc platforms. In particular, I ported and evaluated the versions designated as *Fork/Join* and *Comm hiding* (*Communication Hiding*). A third version *Iteration overlap* (*Iteration Overlap*) got also explored, which aimed at overlapping computation and communication of two consecutive time-steps. Recently, I implemented a further version named *Comm overlap* (*Communication Overlap*), which however runs well only on the Cray XC40 due to problems with the MPI library on the Mont-Blanc prototype regarding the MPI progress engine. All the different version improve the application performance and were done with the usage of Temanejo. The implementation could hardly be done without any supporting tool.

This chapter explains the optimisation steps, needed to gain the optimal performance in case of parallel optimisation. Kernel optimisation or vectorisation is not part of the following explanation. All optimisation steps are implemented with minimal changes to the program code. Most of them are done by just changing pragmas. For some optimisation steps, switching function calls were necessary. The verification of the different implementations was done with Temanejo. All improvements were evaluated on different architectures. For visualising the optimisation steps, I used two different chart types.

Figure 7.2 states the colouring of different the tasks in the following charts. All figures in this chapter are mock-ups. Initially, the dependency graphs were visualised with TEMANEJO, but for better understanding, they are replaced with simplified graph mock-ups. The first chart is a DAG (directed acyclic graph) and visualises the dependencies graph between the tasks. Nodes in this graph are tasks/functions/execution units. The edges are dependencies between these tasks. The second chart is a timeline. In y-direction the cores are plotted; the threads on the same or different hardware nodes. The first number represents the node, the second the thread and are separated by a dot. In x-direction, the execution time of each task executed on its thread is plotted. Blue parts in the timeline represent waiting time (no useful work). There are two possibilities for having blue areas in the chart, either there are no available tasks at the moment, or there are no tasks with unsolved dependencies.

As announced before, there are several parts of the code with their characteristic behaviour (computation, communication, etc.). These parts can be packed into functions and, therefore, these code parts are easily taskifyable. The different charts types uses the same task colouring. There are some more tasks, which are not shown, because they are not essential for the dependency structure.

The possibility of the tiled simulation environment allows separating the computation task into an outer and an inner region. This separation is only done by using different call site pragmas for the tasks. For synchronisation, there are some additional tasks needed, these tasks are represented through *dummy tasks*. These *dummy tasks* are plotted in the graphs, but they are not shown in the timeline because the execution of these tasks does not consume a measurable amount of time.

7.2.1 Fork Join

The simplest version, *Fork/Join*, first executes compute tasks on all tiles and then proceeds to the MPI communication in a single task, which blocks all the other following tasks. This is implemented with OmpSs dependencies, rather than OpenMP-style explicit barriers or synchronisation. The resulting code has been benchmarked across various problem sizes and on different platforms. For problem sizes which are typical for production runs, the time spent in MPI communication is between 5%-10% of the execution time, and thus non-negligible.

```
do timestep=1, timesteps
        !$OMP TASK INOUT(OuterTiles, InnerTiles)
        call MPI_exchange()
        !$OMP END TASK

        do iTile=1, nOuterTiles
                !$OMP TASK INOUT(OuterTile)
                call compute(iTile) !outer tile
                !$OMP END TASK
        end do

        do iTile=1, nInnerTiles
                !$OMP TASK INOUT(InnerTile)
                call compute(iTile) !inner tile
                !$OMP END TASK
        end do

        !$OMP TASK INOUT(OuterTiles, InnerTiles)
        call boundery()
        !$OMP END TASK
end do
!$OMP TASK WAIT
```

FIGURE 7.3: *Fork/Join* code structure.

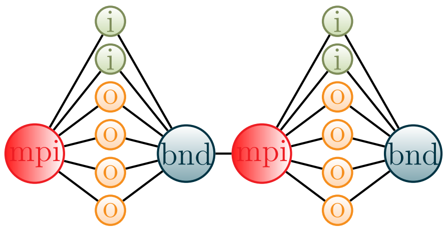


FIGURE 7.4: *Fork/Join* dependency graph

Running the *Fork/Join* implementation, annotated with dependency pragmas (listing in Figure 7.3) in a task-based programming model, the runtime will generate a dependency graph similar to the one in Figure 7.4. In the *Fork/Join* annotated code version, all *compute tasks* (*inner tasks* and *outer tasks*) generate a dependency on their tile. The *MPI exchange task* and the *boundary (bnd) task* take as input and output dependency all compute tiles.

Looking at the same code in a performance analysis tool the timeline would look similar to the illustration shown in Figure 7.5. The different iterations can easily be recognised. The iterations are separated through the *MPI exchange task*. In

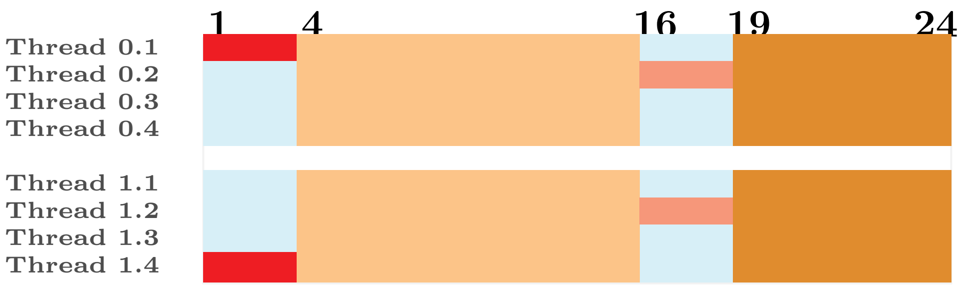


FIGURE 7.5: *Fork/Join* timeline without compute task differentiation

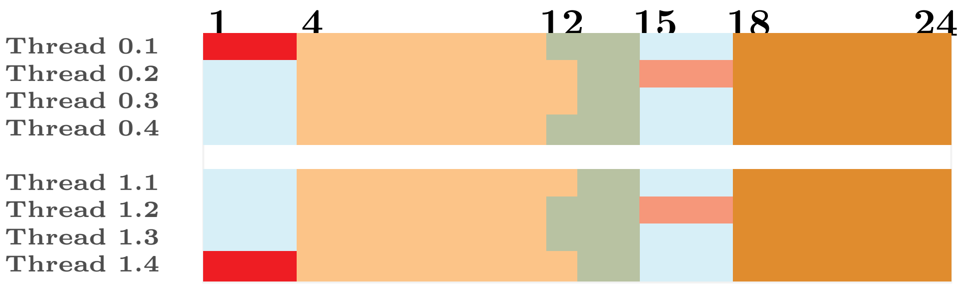


FIGURE 7.6: *Fork/Join* timeline with compute task differentiation

this implementation, there is no differentiation between *inner tasks* and *outer tasks*. For the timeline, the *bnd task* is not shown, because its execution duration is very short. Also assuming that the execution of one computation task (*inner task* or *outer task*) takes one time step (one segment in the timeline).

Additional to the *Fork/Join* design, shown in Figure 7.5, Figure 7.6 differentiates the compute task in *inner tasks* and *outer tasks*. Task priorities can influence the task execution order. In case of LBC, all communication relevant operations are on the critical path, and they depend only on the outer tasks. Therefore, it is important to execute *outer tasks* before *inner tasks*, by adding higher priorities to the *outer tasks*. This differentiation will be used in the following optimisation steps.

7.2.2 Communication hiding

A performance analysis of the version *Fork/Join* (Chapter 7.2.1) showed, that there is very little room for performance improvement inside the communication task using MPI primitives exclusively. This was done by designing various versions using non-blocking communication routines. Instead, I investigated to overlap the communication task as a whole with computation tasks if possible.

```
do timestep=1, timesteps
        !$OMP TASK INOUT(OuterTiles)  %PRIORITY(HIGH)
        call MPI_exchange()
        !$OMP END TASK

        !$OMP TASK INOUT(OuterTiles, InnerTiles)
        call dummy()
        !$OMP END TASK

        do iTile=1, nOuterTiles
                !$OMP TASK INOUT(iTile_OuterTile) %PRIORITY(HIGH)
                call compute(iTile) !outer tiles
                !$OMP END TASK
        end do

        do iTile=1, nInnerTiles
                !$OMP TASK INOUT(iTile_InnerTile) % PRIORITY(LOW)
                call compute(iTile) !inner tiles
                !$OMP END TASK
        end do

        !$OMP TASK INOUT(OuterTiles)
        call boundery()
        !$OMP END TASK


end do
!$OMP TASK WAIT
```

FIGURE 7.7: *Comm hiding* code structure.

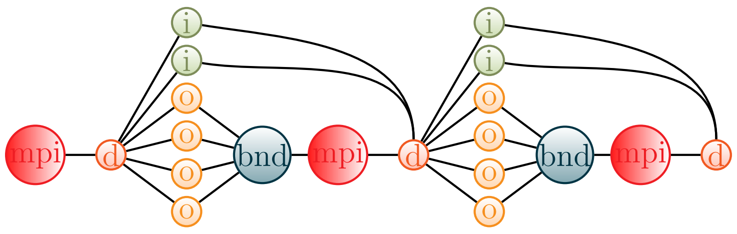


FIGURE 7.8: *Comm hiding* dependency graph with differentiation in inner and outer tasks.

Following that approach, a first optimisation, called *Comm hiding* (listing in Figure 7.7), aims to execute those *compute tasks*, which are necessary inputs for the *MPI exchange task* first. Specifically, only those tiles which lie on the surface of a given MPI domain are needed for the exchange of ghost cells via MPI. The remaining compute tasks on the *inner tiles* are done concurrently with the MPI

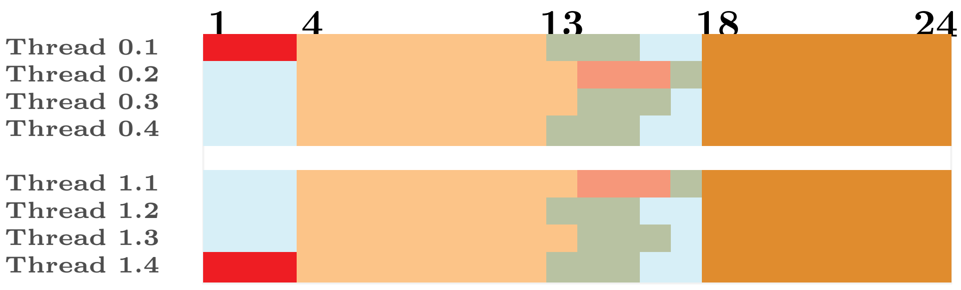


FIGURE 7.9: *Comm hiding* timeline with normal behaviour.

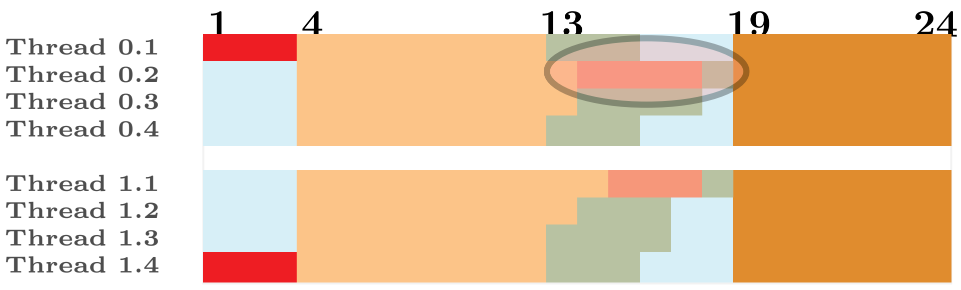


FIGURE 7.10: *Comm hiding* timeline with delay in the MPI communication.

communication task. For typical problem sizes, this version is capable of hiding most, if not all, of the MPI communication.

Figure 7.8 shows the *Comm hiding* concept as a dependency graph. The *dummy task* is introduced ad placed between the *MPI exchange task* of the last iteration (iteration n-1), and the *compute task* of the current iteration (iteration n). With the placement of the *dummy task*, it is possible to hide the MPI communication inside the *inner tasks*. This is possible because the *inner tasks* are not relevant for the communication. To do so, higher priorities for the *outer tasks*, the *bnd tasks* and the *MPI exchange task* were introduced. These priorities force the runtime to execute these tasks as soon as possible.

Figure 7.9 shows the normal communication hiding behaviour. The efficiency of hiding the communication depends on the ratio between *inner tasks* and *outer tasks*. The quantity of inner and outer tasks depends on the problem size and the size of the tiles. A domain with a problem size of 256 in each dimension (x, y, z) and a tile size of 64, will be decomposed in 27 tasks and 37 outer tasks. With a tile size of 32, there are 343 inner tasks and 169 outer tasks. The ratio between inner

and outer tasks affects the communication hiding possibility. Without enough inner tasks, it is not possible to overlap computation and communication. The number of inner tiles can be computed by:
$tiles_{tot} = \frac{d_x}{t_s} * \frac{d_y}{t_s} * \frac{d_z}{t_s}$
$tiles_{in} = (\frac{d_x}{t_s} - 2) * (\frac{d_y}{t_s} - 2) * (\frac{d_z}{t_s} - 2)$
$tiles_{ratio} = \frac{tiles_{tot}}{tiles_{in}}$
under the assumption, that the domain size (d_x, d_y, d_z) is dividable by the tile size (t_s). As long as the computation of inner tasks takes longer than the MPI_exchange tasks, the communication is fully overlapped and even a slight communication delay does not affect the overall application performance on today's HPC systems. Only one core is occupied handling the *MPI exchange task*. Systems with fewer cores are more affected by a communication delay because such a system loses more compute power. The ratio between tasks per core is also playing a crucial role. The more cores an application can use, the less inner tasks per core will be executed. This affects the possibility of hiding communication inside inner tasks, but the overall performance is not affected by a communication delay. In contrast an application using fewer cores has more inner tasks per core and a better possibility of hiding communication, but the overall performance is more affected by a communication delay.

In LBC, the communication uses *MPI_Sendrecv* inside the *MPI_exchange task*. It could happen, that the application is waiting for messages from another rank and is, therefore, not performing actual work: computation. Figure 7.10 illustrates this behaviour by increasing the MPI duration on a specific node. In reality, this delay could easily be longer than one time step or compute task execution.

7.2.3 Communication Overlap

In the version *Comm hiding* (Chapter 7.2.2), MPI communication is done in an OmpSs task which runs concurrently with computation tasks. Modern MPI implementations, however, can do most of the actual communication with minimal assistance of the CPU, by using resources from the network interface directly. OmpSs worker threads could execute actual work while the CPU idles waiting for the MPI communication to complete. Thus, the version *Comm overlap* aims to yield the core during MPI communication. This allows executing other pending tasks during a long MPI communication. The technique is being prototyped on

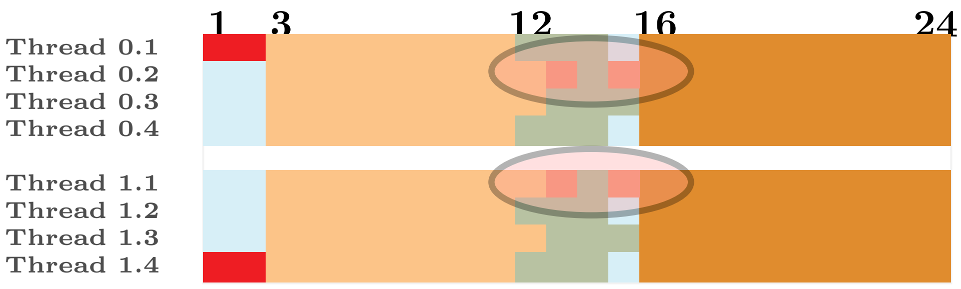


FIGURE 7.11: *Comm overlap* timeline with runtime aware MPI task behaviour.

application level by introducing a special MPI library. To solve the problem, shown in Figure 7.10, a special MPI library splits the *MPI_Sendrecv* into a *MPI_Send* and *MPI_Recv part*. By using the possibility of yielding tasks (this is a feature of the OmpSs runtime), the application is now able to execute the *MPI_Send* and set a flag, allowing to detect if the *MPI_Send* was completed. After the application has finished to sent data and the application couldn't start receiving, computation tasks get executed. The communication task execution is interrupted, and the runtime is capable of rescheduling the task. A very basic implementation of the rescheduling process just restarts the tasks until the necessary MPI message is available. With this concept, the runtime could hide the communication latency, but it also generates some overhead in the runtime by yielding and rescheduling tasks. At this point, I stopped following this concept and accepted the overhead in the runtime.

Figure 7.11 shows the improvement of the simple concept (just rescheduling tasks). The next iteration can now start one time step earlier, and there are fewer gaps (blue time steps) between the iterations.

An improvement to this basic implementation could be a design rescheduling tasks only if a specific operation is valid. This could be the validation of *MPI_Test*, and only if the *MPI_Test* is valid for the task's necessary parameters, the task gets rescheduled. A concept along these lines is also described in the paper [29] and the thesis of Vladimir Marjanović [30].

7.2.4 Iteration Overlap

This last optimisation step is named *Iteration overlap*. The goal is to overlap calculation of multiple interactions if possible. This step adds a lot more complexity to the application, but allows better hiding of communication; the communication can be hidden inside multiple iterations.

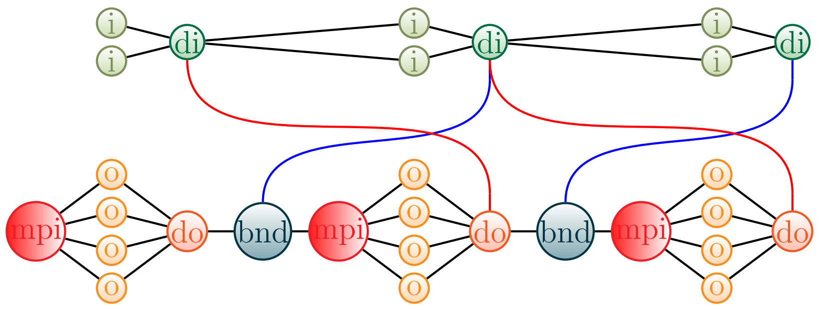


FIGURE 7.12: *Iteration overlap* dependency graph. The necessary dependencies are getting complex.

First of all, two *dummy tasks* have to be introduced; one for the *inner tasks* and one for the *outer tasks*. This has little effects on the original code structure, e.g. both *dummy* tasks have to be placed behind the compute execution calls. After adding these two tasks, the dependency pragmas has to be adapted. As in the steps before this optimisation step would not be possible without a graphical tool visualising the dependencies graph generated by the runtime. For this step double buffering of the main data structure is also needed. This means, two time-steps are stored in memory (*iteration n-1* and *iteration n*). Due to problems expressing these complex dependencies through a pragma-based programming model, this work is only conceptional and is not completely implemented. Without a pragma-based concept, a correct implementation would require an additional buffering and management of data, complicating the original code, significantly.

Figure 7.12 illustrates the dependencies between the iterations. The *inner dummy task* of *iteration n-1* has a dependency on the *outer dummy task iteration n* . The *bnd task* of *iteration n-1* has a dependency on the *inner dummy task iteration n*.

Figure 7.13 shows the expected result from overlapping and getting rid of the light blue idling time steps. In practice, there will be some idling parts in the performance analysis. This happens because at some point the application will run into the problem of a delayed *MPI exchange task*. This problem can be solved with the concept shown in Figure 7.11 and in Figure 7.14. Always execution *inner*

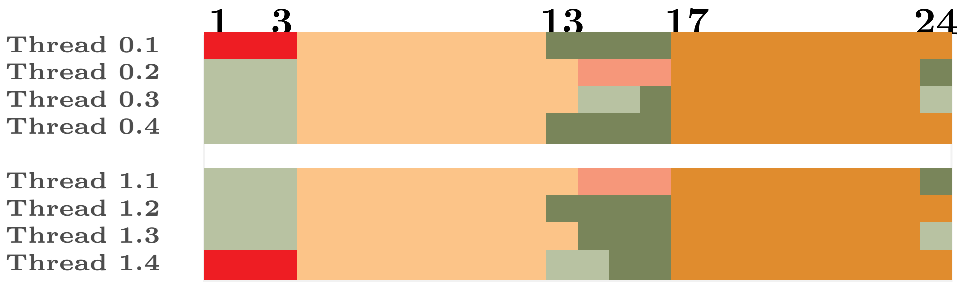


FIGURE 7.13: *Iteration overlap* timeline.

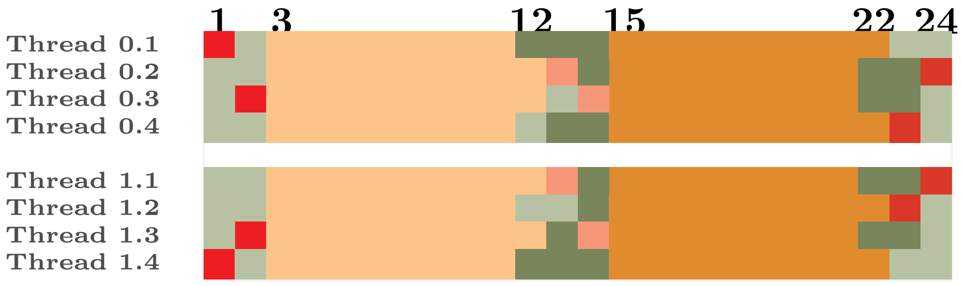


FIGURE 7.14: timeline of combied *Iteration overlap* and *Comm hiding*

tasks during communication could lead to a lack of *inner tasks*, this could, at some point, decrease the application performance.

The last Figure 7.14 includes all previous optimisation steps and also the optimisation step of this chapter. The overlap optimisation is combined with the task yielding concept. The light red *MPI exchange task* is interrupted, because of a communication delay, and a *inner tasks* (computation task) gets executed. Besides, the communication is hidden inside two different iterations.

7.3 Performance Evaluation

This sections presents systematic performance reports and, in addition, briefly discusses the the need for tuning the number of OmpSs worker threads per MPI process on the Mont-Blanc ThunderX prototype. Note, that the application specific metric MLUP/s (mega-lattice updates per second) is proportional to GFLOPs, with a constant translation factor of 255 FLOP/LUP. The results are reported as performance per node.

7.3.1 Evaluation Platform

For evaluating the LBC, application three conceptually different systems have been chosen. This allows a good overview of the overall performance on different architectures. The different versions show not only implementation specific bottlenecks depending on a given hardware, but also allow to evaluate the underlying hardware with a real world application, running on current HPC systems.

- Mont-Blanc prototype
  This is a system placed in BSC and consists of 930 compute nodes (Samsung Exynos 5 Dual). Each node has two cores (Cortex-A15 @ 1.7GHz), and is based on the ARM 32-bit architecture. Every node also includes 4GB of LPDDR3 memory and an ARM Mali T604 GPU. The network uses 10GbE Ethernet over USB.

- Hazel Hen
  The Cray XC40 Hazel Hen is located at the HLRS and consists of 7712 compute nodes. Each node has two NUMA domains with 12 cores (Intel Xeon CPU E5-2680 v3 (30M Cache, 2.50 GHz)) each (in total 185088 cores). The memory capacity of each node is 128GB, the systems peak performance is around 7420 TFlops and consumes $\tilde{3}200$ kW of power. The node to node interconnect is developed by Cray and named Aries. Hazel Hen is a typical system used in HPC.

- Cavium ThunderX
  This system is also ARM based and is located at BSC. It consists of 4 compute nodes, each node has two NUMA domains with 48 cores (ARMv8-A @ 1.8 GHz), in total 384 cores. The memory capacity in each node is 128GB and the interconnect is based on 10Gb Ethernet. The system is one of the first ARM-based 64-bit server architectures, and therefore an interesting evaluation platform.

7.3.2 Tuning of number of OmpSs workers and process placement

One of the motivations for hybrid OmpSs/MPI programming (or for hybrid OpenMP/MPI for that matter) is to reduce the number of MPI processes, both, in order

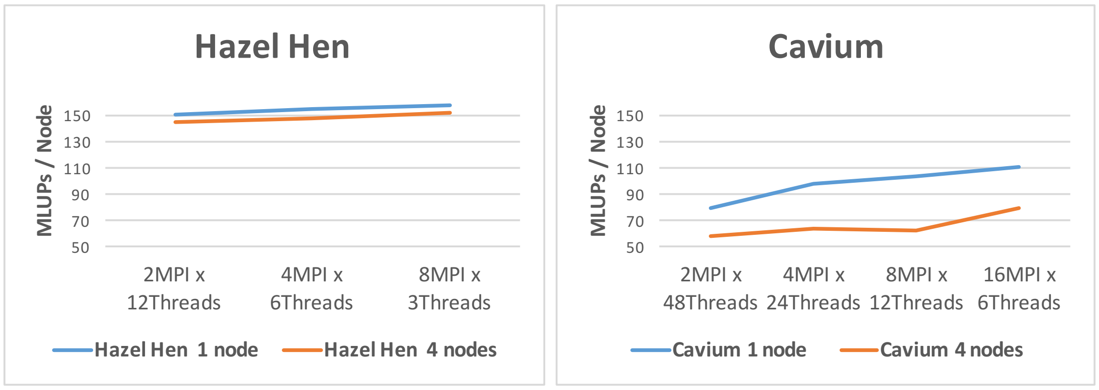


FIGURE 7.15: shows the performance as a function of MPI processes and OmpSs worker threads per node.

to aggregate communication, but also to circumvent potential MPI scalability issues. It is, therefore, expected to run such hybrid applications with only one MPI process per node and set the number of OmpSs worker threads equal to the number of cores per node. In practice, however, to avoid performance loss due to NUMA issues, it is often favourable to use one MPI process per NUMA domain and to pin the threads to cores in the respective domain. The previous experience with hybrid OmpSs/MPI applications suggested, that the performance did not change much if the number of MPI processes per NUMA domain was increased further beyond one. However, on the Mont-Blanc Cavium ThunderX system the performance changes considerably by increasing further the number of MPI processes per node. This is illustrated in Figure 7.15. The Performance is reported as a function of MPI processes and OmpSs worker threads per node for the Cavium ThunderX and the Cray XC4 system, respectively. For the Cray XC40 (left), the performance does not depend much on the distribution of cores to MPI processes and OmpSs worker threads. For the Cavium (right) the performance can be improved significantly by using more MPI processes and less OmpSs worker threads per process. Clearly, the number of MPI processes per node on the Cray XC40 does not affect the performance significantly. However, on the Cavium ThunderX performance improves with an increasing number of MPI processes and decreasing number of OmpSs worker threads per MPI process. Concluding this evaluation, the Cavium ThunderX is strongly affected by the amount of processes per NUMA domain, however, the Hazel Hen system is only slightly affected and can easily get along with one process per NUMA domain. For the following evaluation 2 MPI with 12 Threads each are used for Hazel Hen system and 4 MPI with 24 Threads each for the Cavium system.

7.3.3 Benchmark on Hazel Hen

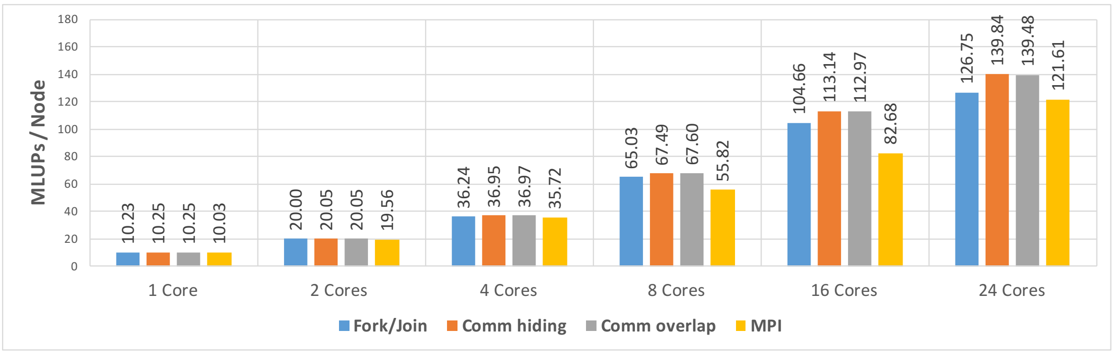


FIGURE 7.16: show the performance comparison of the different LBC versions (strong scaling experiment). These results were generated on the Hazel Hen.

This first section compares the three hybrid parallelisation concepts, *Fork/Join*, *Comm hiding* and *Comm overlap* with a *pure MPI* parallelisation. For benchmarking purpose four nodes of the Cray XC40 are used, and the results are reported in MLUP's per node. The three versions run fine on the Cray XC40 as shown in Figure 7.16. The *Comm overlap* version is able to reuse some of the time spent in MPI communication to execute pending compute tasks, and is thus slightly faster on the Hazel Hen architecture, than the *Comm hiding* version. For this particular setup, the potential for performance improvement of *Comm overlap* over *Comm hiding* is small, since communication time on the XC40's network is already very low. With increasing numbers of cores per node, the hybrid LBC implementations outperform the *pure MPI* implementation. These hybrid runs use 4 nodes with one MPI process per NUMA domain (8 MPI ranks in total). An evaluation of the ratio between threads and processes can be found in Chapter 7.3.2.

7.3.4 Benchmark on the Mont-Blanc prototype

Figure 7.17 compares the *Fork/Join* and *Comm hiding* versions with the pure-MPI version. It shows the results of a weak scaling experiment on the Mont-Blanc prototype. However, I encountered an issue with the MPI version which is installed on the MontBlanc prototype. This version does not allow to yield the CPU in an efficient way. Therefore, I have not benchmarked the *Comm overlap* version on the Mont-Blanc prototype. The parallel efficiency (scaling efficiency; $E_N = \frac{T_N}{T_1}$) of all versions remains higher than 90% for all versions up to 448 nodes. Closer inspection shows that *Comm hiding* effectively hides communication and thus

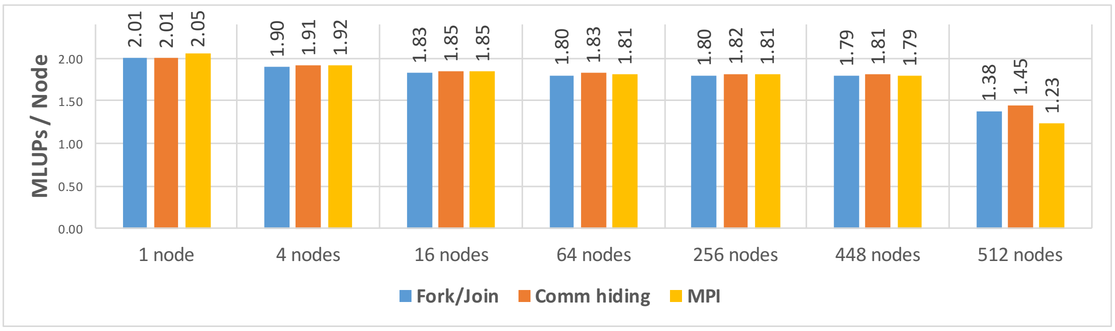


FIGURE 7.17: shows the comparison of two OmpSs versions with the pure-MPI implementation of LBC(weak scaling experiment). These results were generated on the Mont-Blanc prototype

outperforms the pure-MPI version, as well as the simpler OmpSs/MPI version *Fork/Join*. At 512 nodes the parallel efficiency of all version drops significantly to approximately 57-75%. Again, the *Comm hiding* version is the fastest, followed by *Fork/Join*, and with a larger distance by the pure-MPI version. The reason for the performance drop in the first place is not clear. There are indications that this is temporary problem due to the prototypes interconnect, but more recent trial runs seem to indicate that the problem no longer exists.

7.3.5 Comparing the Cavium ThunderX with the Cray XC40

For comparing two different hardware architectures, the *Comm hiding* version running a weak scaling experiment, was taken. On the Cavium, the best performance results were obtained by splitting the domain into 16 MPI ranks with 6 OmpSs threads, per node. On the Cray XC40, the domain is divided into 8 MPI ranks with 3 OmpSs threads, per node. The different configurations are evaluated in chapter 7.3.2, and the selected configuration achieves the best overall performance. With this configuration, the benchmark was executed and four figures, representing the output in different views, were generated.

1) "MLUPs / node" (Figure 7.18) shows a weak scaling experiment running on the target systems. In this benchmark, the Cray XC40 outperformed the Cavium ThunderX when comparing node by node. On both systems, the hybrid OmpSs/MPI version of LBC performs better than the *pure MPI* version.
2) "MLUPs / GFLOs (peak)" (Figure 7.19) sets the MLUPs in relation to the peak performance of the system. For the Cavium system, the peak performance

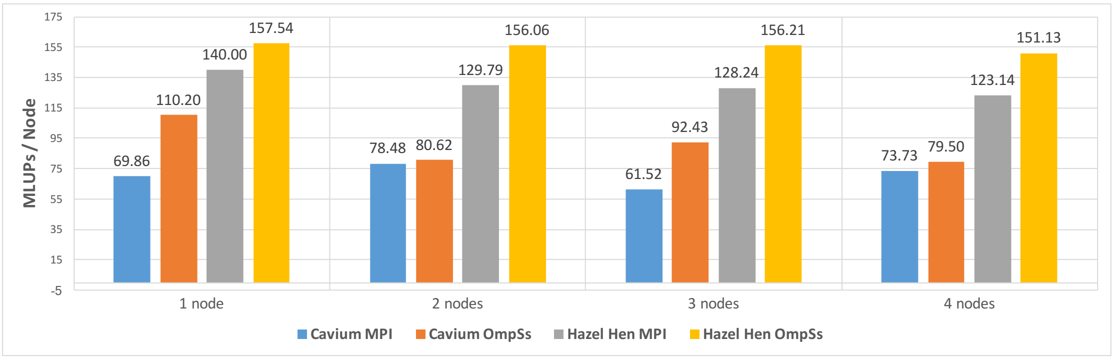


FIGURE 7.18: shows a weak scaling experiment running on the target systems.

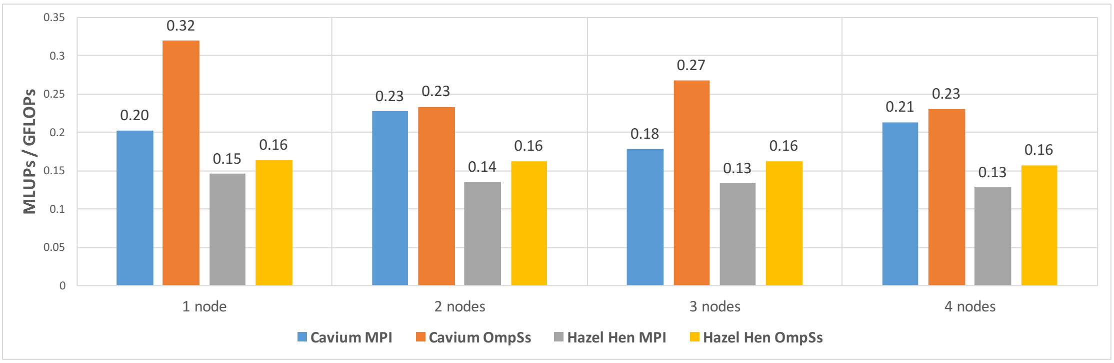


FIGURE 7.19: sets the MLUPs in relation to the peak performance of the system (Cavium: 345 GFLOPs, XC40: 960 GFLOPs).

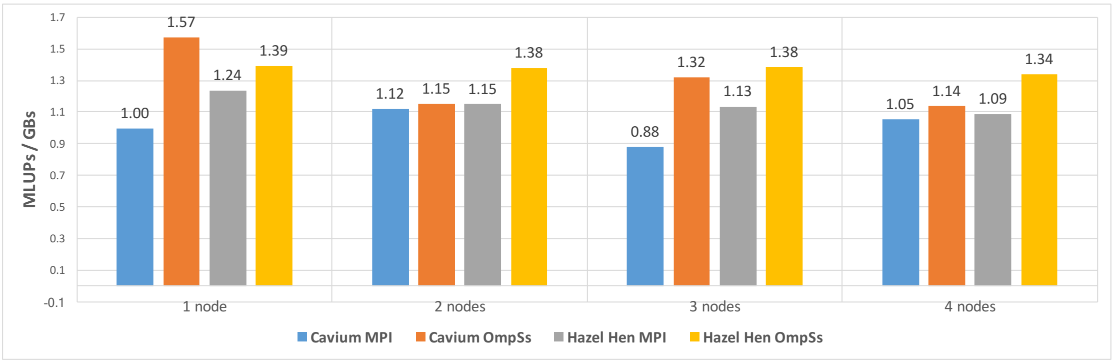


FIGURE 7.20: sets the MLUPs in relation to the measurable memory bandwidth (Cavium: 70 GBs, XC40: 113 GBs).

is around 345 GFLOPs. The XC40 has a peak performance of 920 GLOPs.
3) “MLUPs / GBs (stream)” (Figure 7.20) sets the MLUPs in relation to the measurable memory bandwidth (measured with the stream benchmark). For the Cavium system the bandwidth is around 70 GBs. The XC40 maximum bandwidth is around 113 GBs.
4) “Fraction of Peak Performance” (Figure 7.21) shows the percentage of peak performance the application reaches on the target platform. As theoretical peak

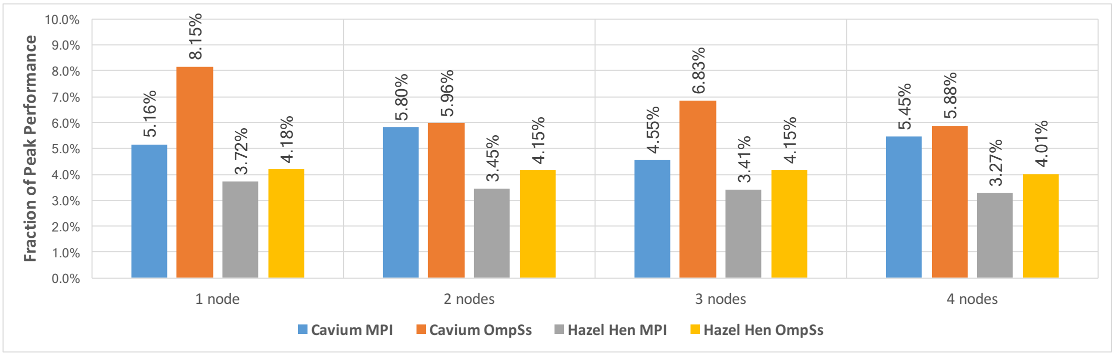


FIGURE 7.21: shows the fraction of peak performance the application reaches on the target platform.

performance for the Cavium ThunderX system 345 GFLOPs are used, and 960 GFLOPS for the Cray XC40. The GFLOPs for LBC can be calculated according to the relation 255 FLOP/LUP. Normalising the result to the theoretical peak performance shows a big benefit for the Cavium ThunderX system and especially the hybrid OmpSs/MPI version gains a lot from the architecture.

7.4 Use Case Conclusion

In general, the application efficiency in case of parallel performance, results in a complex dependency graph. Verifying such dependency implementations is difficult without tools like TEMANEJO, supporting the developer in his actions. Also TEMANEJO was used for tuning the performance by monitoring the data locality of the tasks. Nevertheless, these results could only be achieved by tuning the application a lot and splitting the domain into several sub-domains. In particular, on both systems, we had to fine-tune the number of MPI processes and OmpSs threads. Running naively with one MPI process per node (or per NUMA domain) did not achieve high performance for the hybrid OmpSs/MPI version. This indicates, that data-locality on NUMA architectures, which is the main performance issue of LBC, are still not well handled by the OmpSs runtime, at least not with the combination of data-layout and dependencies as present in LBC. In case of system utilisation, the LBC code is running more efficient on the Cavium ThunderX system than on the Cray XC40, but in case of MLUPs or time to solution, the Cray XC40 is around a factor of 2, for one node, faster than the Cavium ThunderX. For four nodes the factor for *pure MPI* is around 1.6. Comparing the

OmpSs implementation for one node, the factor is around 1.4, and for four nodes the Cray XC40 is by the factor of 1.9 faster than the Cavium ThunderX.

Chapter 8

Conclusions & Open research topics

In this chapter I try to formulate some conclusions and give an overview about open research topics in the future. The chapter is split up into two section. First, I will give a conclusion about AYUDAME, the tool's back-end, and second, I will briefly present some ideas continuing the presented work. The future work or open research topics are mostly related to TEMANEJO, the tool's front-end.

8.1 Discussion

Today's HPC ecosystem is an evolving field and requires future debugging techniques. The presented work shows a new approach, called *Model-Centric debugging* for task-based programming models. The idea and design of such an approach is pointed out in the TEMANEJO toolchain. The different programming models and their characteristics were briefly introduced. These characteristics are relevant for connecting TEMANEJO to the programming models. As mentioned, TEMANEJO consists of an back-end, called AYUDAME, and a front-end, which gives the tool its name. In the context of the presented work I only payed attention to the design of the back-end library, including a clear and generic interface for an easy adoption. The connection between a programming model and TEMANEJO is based on an generic designed event & request system. For the most common programming models OpenMP, MPI and OmpSs the work shortly describes the characteristics and the way how the instrumentation is done.

Temanejo has to fulfil demands essential for HPC. Therefore, the following design requirements have to be met:

- The possibility of scaling across multiple nodes. In the chapter 5.5 the issues for a scalable back-end library are described. A solution for solving these issues is explained in chapter 5.5.1. Basically, Temanejo uses multi-level *communication tree* connecting the different Ayudame instances. In this *communication tree* some Ayudame instances get additional functionally and only one instance is directly connected to the front-end.
- The handling of multiple programming models at the same time based on a flexible design is also necessary. For every programming model an specialised *Event Handlers* can be created. This *Event Handlers* takes care of the special needs for a given runtime. In chapter 5.4.1 different designed *Event Handlers* and their specialities are described. The *OMPT Event Handlers*, for example, provides the necessary function for OMPT. The *OMPT Event Handlers* takes care about the *communication tree* initialisation. The *OmpSs Event Handlers* implements TCA for controlling the runtime.
- The unique identification of events and requests in a distributed application is also an requirement. The solution therefor was to combine several identifiers. To make the identifier unique Temanejo uses three distinct IDs, the MPI rank, the process ID and an ID for every programming model inside a process. This concept is briefly described in the chapter 5.6.2.

8.1.1 Evaluation of the design

The design of Temanejo is evaluated in a performance improvement case study. The case study shows the productive usage of *Model-Centric* debugging, in particular performance debugging, and the benefits an application developer could gain. In this case study several sub-steps with their optimisations are described in detail. Getting the best performance out of a given application needs a fundamental knowledge about the application and the used programming models. Even with the usage of a *Model-Centric* debugging tool developing such an application is a complex undertaking. As a use case I took the LBC application. The first implemented version of LBC follows the fork/join paradigm, this version blocks the communication until the computation is done. With Temanejo the dependency graph, of the *Fork/Join* version was analysed and the bottleneck identified. The second version solves the identified issue and is named *Comm hiding*. The goal of this version was to overlap communication and computation by rearranging dependencies between tasks. Again Temanejo was used to check the correctness of the dependency graph. A third version *Iteration overlap* got also explored, which aimed at overlapping computation and communication of two consecutive time-steps.

8.2 Open research topics

The main part of this thesis was the design of the back-end library, most of the future work has to be done inside the graphical debugger's front-end. Ayudame can handle most necessary cases (hybrid programming models, distributed environment, nested models or tasks, etc.). Temanejo was prototyped besides this thesis, but is only supporting basic functionality. Meaningfully visualising all these things in a graphical tool isn't a straight forward implementation. The actual performance for rendering and analysing the dependency graph is sluggish. In addition there are some features necessary in the future:

- Known limitation in Ayudame.
  The generic interface showed a big advantage for transferring the events & requests between Temanejo & Ayudame. In case of performance it would be better to create the *Intern_Event* objects only if it is necessary, otherwise they could be kept as bytes. For routing the events & requests to their destination, it is sufficient to know the *unique_ID* and maybe the *event type*. The transferred data are only relevant at their destination. Changing the internal design would lead to a performance improvement inside the backend-library.

- On demand forwarding of events.
  At the moment, every message is transferred directly to Temanejo, but in a scaling environment the amount of transferred data will dramatically increase. Therefore, a future work package has to care about remote data storage, where only necessary information (Chapter 4.2.1) has to be processed in the front-end. All additional information (Chapter 4.2.2) can then

be requested by Temanejo. But even then information can be filtered, if the dependency graph is nested inside another graph and is not needed in the current visualisation view. This will decrease the memory needed at the node Temanejo is running on.

- Reducing the complexity of the visual representation.
  In a hybrid environment the visualisation can get overcrowded. Therefore, collapsing graphs, containing identical information, is a necessary technique and will help solving this issue. For example such a collapsible graph could be a nested sub-graph, which looks the same on ever MPI rank. For an application developer it is only interesting to see abnormal behaviour, which could lead to a different looking graph.

- Performance of the front-end.
  As for now, the performance of the actual implementation of Temanejo is sluggish. This is mostly related to the treatment of threads and locks inside Python (global interpreter lock). Therefore, I propose a Temanejo implementation based on QT and C++. Especially the layout routine has to be re-implemented and simplified. With this reimplementation the layout routine has to be adapted to support the layout of nested graphs, which is, at the moment, implemented as proof of concept without a clear design.

Appendix A

Appendix

A.1 Hardware Technology

A.1.1 Non-uniform memory access

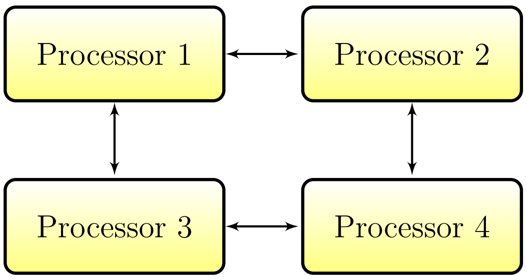


FIGURE A.1: Ring

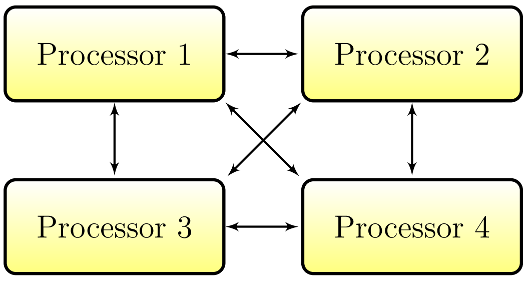


FIGURE A.2: Cross link

Non-uniform memory access, or short NUMA, is a memory design for multicore systems. In a NUMA system, every processor has his local memory, but can access the memory of every other processor in the system through a global address space.

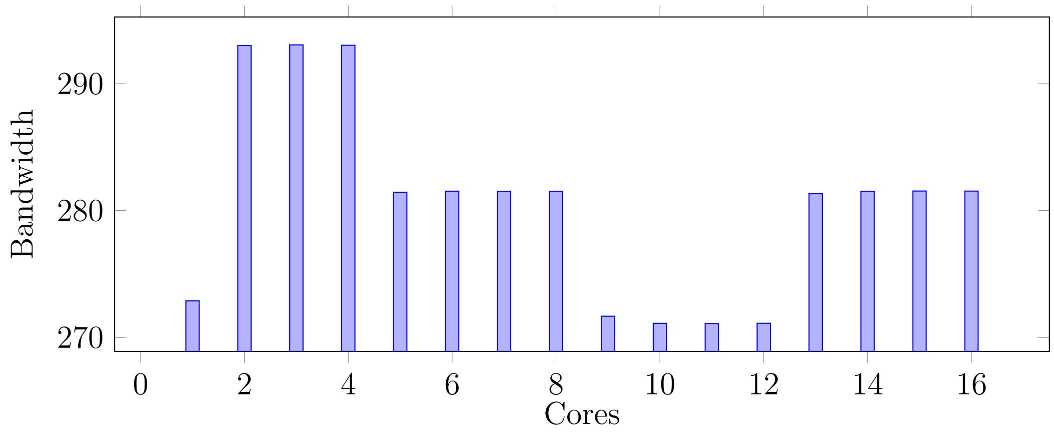


FIGURE A.3: Memory bandwidth 4 socket system with 4 cores per socket. In this system no cross link available and therefore a second stage performance drop is measurable

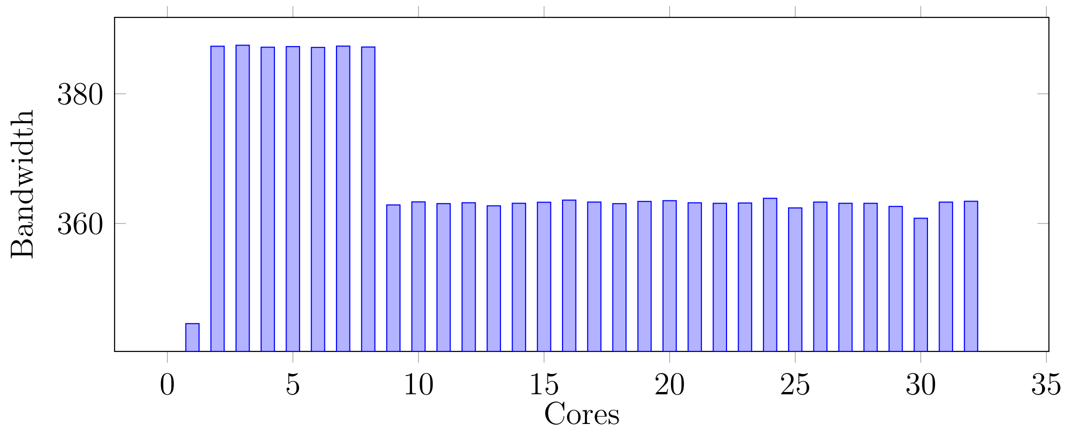


FIGURE A.4: Memory bandwidth 4 socket system with 6 cores per socket. In this system a cross link available and therefore no second stage performance drop is measurable

Data locality, therefore, impacts the memory latency and bandwidth. Accessing an address region placed in another NUMA domain increases the latency and decreases the bandwidth. In Figure A.1, every memory access to a non directly connected processor causes an additional increase of latency and bandwidth. This issue is solved by an extra link between the two sockets (processors) in Figure A.2. Figures A.3 and A.4 show the benchmark results for two different NUMA systems. The architecture illustrated in Figure A.3 has no cross-link attached. In contrast, Figure A.4 shows the attached cross-link and therefore, no second performance drop is measurable. In addition, a basic system configuration is shown in Figure A.6, there are two sockets connected with each other through an interconnect. But each of them has its own memory. The table in Figure A.5 shows some of the NUMA systems placed at HLRS. For two of those non-uniform memory systems,

Generation	CPU	Sockets	Threads per Socket	NUMA	Memory	Systems @ HLRS
Interlagos November 2011	AMD Opteron 6238 @2.60GHz	4	12	8	DDR3 @ 1600MHz	Laki Interlagos Figure A.7
Nehalem EP March 2009	Intel Xeon X5560 @2.80GHz	2	8	2	DDR3 @ 1333MHz	Laki Nehalem
Nehalem EX March 2010	Intel Xeon x7542 @2.67GHz	8	6	8	DDR3 @ 1066MHz	Laki smp Figure A.8
Sandy Bridge EP March 2012	Intel Xeon E5-2670 @2.60GHz	2	16	2	DDR3 @ 1600MHz	Laki Sandy Bridge
Haswell EP September 2014	Intel Xeon E5-2660v3 @2.60GHz	2	20	2	DDR4 @ 2133MHz	Laki Haswell
Ivy Bridge EP September 2013	Intel Xeon E5-2580v2 @2.80GHz	2	20	2	DDR3 @ 1866MHz	TheoSIM
Haswell EP September 2014	Intel Xeon E5-2680v3 @2.60GHz	2	24	2	DDR4 @ 2133MHz	Hazel Hen

FIGURE A.5: Non-Uniform Memory Systems @ HLRS

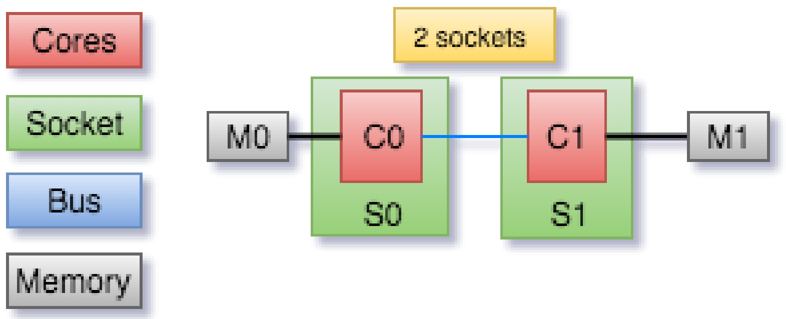


FIGURE A.6: Two socket architecture. Each socket has its own Memory

a detailed view of the node's architecture is attached. Figure A.7 shows the *laki Interlagos* architecture, and Figure A.8 shows the *laki smp* architecture. In reality, the NUMA diversity and variety is too complex to be handled by an application developer. In modern CPUs, we can find two or even more rings, inside a socket, each of them connected to its memory controllers. This introduces an additional

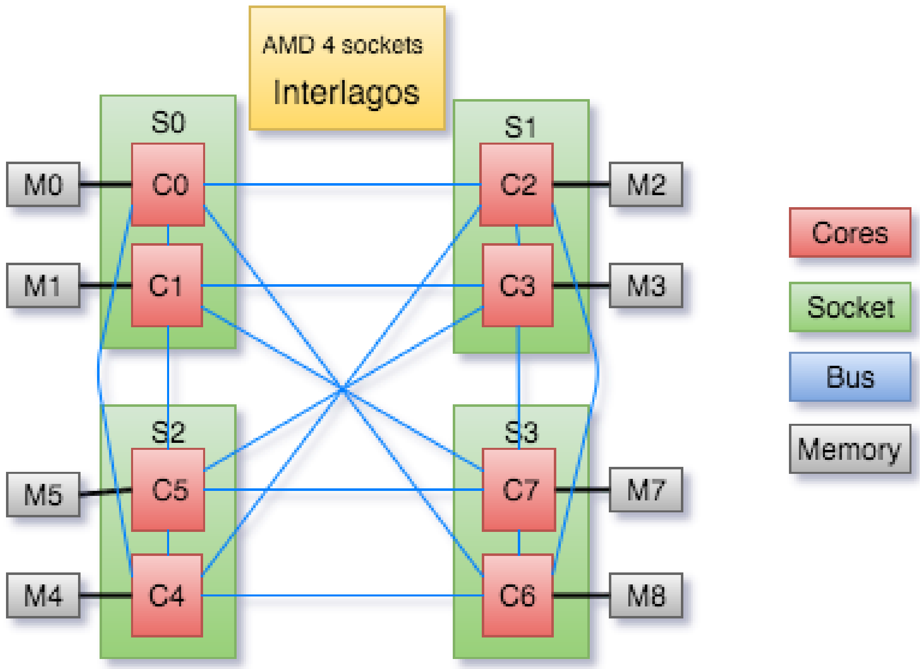


FIGURE A.7: Interlagos architecture.

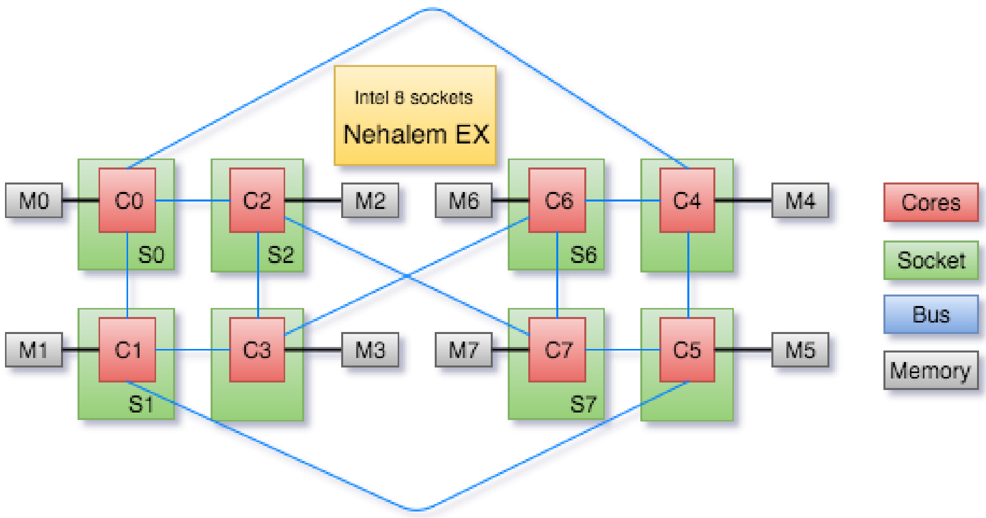


FIGURE A.8: Nehalem EX architecture.

level of complexity to the non-uniform memory access.

A.1.2 Memory layers

2)The hardware in high-performance computing systems is getting more and more complex, this increasing complexity is also visible in other computer systems. Also, the complexity of the different memory layers inside a node has increased in today's hardware. In the area of memory layers, there are different cache levels:

level 1 cache, level 2 cache, level 3 cache and maybe level 4 cache. Behind these cache levels the DRAM, High Bandwidth Memory (HBM), Hybrid Memory Cube (HMC) and Non-volatile Memory (NVM) like NVRAM is placed. NVRAM, for example, needs a special function call, nv_alloc() instead of a alloc(), to allocate memory. Finally, there are discs (I/O) which can be separated into several levels (Buffer, SSD, HDD, tape storage).

A.1.3 Distributed memory

Particularly in the high-performance computing field, where applications run across several nodes, the application developer has to think about shared and distributed memory systems. In addition to the circumstances associated with the intra-node hardware architecture, the complexity can be increased by adding the internode communication to the overall system design. This increases the levels of complexity by one or multiple layers, depending on the network infrastructure and topology (3D Torus, dragonfly, etc.).

A.1.4 Heterogeneous systems

Another aspect of the increasing hardware complexity is the heterogeneity of today's and future hardware. One future trend is the usage of big and little cores or specialised cores for specific operations. This could be, for example cores, running the operation system (OS) or cores handling network communication, e.g. big and little cores, CPGPU, FPGA and SPARK 2.

A.1.5 Knights Landing

Looking at the actual hardware technology from *Intel*, the Knights Landing there are three different memory mode's: Cached Mode, Flat Mode, and Hybrid Mode. These three memory modes can be combined with the four different clustering modes: All-to-All, Quadrant, Hemisphere, SNC (sub-NUMA cluster modes)-4 and SNC-2. The application developer cannot change these mode's, but depending on the application the configuration will impact the performance.

Bibliography

[1] TOP500 Authors. Top500 the list june 2016. URL https://www.top500.org/lists/2016/06/.

[2] DAVID B. SKILLICORN. Models and languages for parallel computation. 1998. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.1801&rep=rep1&type=pdf.

[3] Ingo Molnar. The native posix thread library for linux. Technical report, Tech. Rep., RedHat, Inc, 2003.

[4] Kevin Pouget, Patricia López Cueva, Miguel Santana, and Jean-François Méhaut. Interactive Debugging of Dynamic Dataflow Embedded Applications. In *Proceedings of the 18th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, Boston, Massachusetts, USA, may 2013. Held in conjunction of IPDPS.

[5] The OpenMP Architecture Review Board. The openmp® api specification for parallel programming. URL http://http://openmp.org/.

[6] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[7] Barcelona Supercomputing Center. Programming models @ bsc boosting parallel computing research since 1989. URL https://pm.bsc.es/ompss.

[8] J.M. Perez, R.M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142–151, Sept 2008. doi: 10.1109/CLUSTR.2008.4663765.

[9] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011. doi: 10.1142/S0129626411000151. URL http://www.worldscientific.com/doi/abs/10.1142/S0129626411000151.

[10] Roger Ferrer, Judit Planas, Pieter Bellens, Alejandro Duran, Marc Gonzalez, Xavier Martorell, Rosa M. Badia, Eduard Ayguade, and Jesus Labarta. Optimizing the exploitation of multicore processors and gpus with openmp and opencl. In *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing*, LCPC'10, pages 215–229, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19594-5. URL http://dl.acm.org/citation.cfm?id=1964536.1964551.

[11] Inventeurs du monde numérique. Starpu is a task programming library for hybrid architectures. URL http://starpu.gforge.inria.fr.

[12] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André; Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, feb 2011. ISSN 1532-0626. doi: 10.1002/cpe.1631. URL http://dx.doi.org/10.1002/cpe.1631.

[13] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 863–874, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03868-6. doi: 10.1007/978-3-642-03869-3_80. URL http://dx.doi.org/10.1007/978-3-642-03869-3_80.

[14] Kamran Idrees, Mathias Nachtmann, and Colin W Glass. Evaluation of fastflow technology for real-world application. In *Sustained Simulation Performance 2013*, pages 77–88. Springer, 2013.

[15] StarPU consortium. Starpu handbook. URL http://starpu.gforge.inria.fr/doc/starpu.pdf.

[16] Swig. Swig. URL http://www.swig.org/.

[17] Rainer Keller, Steffen Brinkmann, José Gracia, and Christoph Niethammer. Temanejo: Debugging of thread-based task-parallel programs in starss. In Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2011*, pages 131–137. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-31475-9. doi: 10.1007/978-3-642-31476-6_11. URL http://dx.doi.org/10.1007/978-3-642-31476-6_11.

[18] Steffen Brinkmann, José Gracia, and Christoph Niethammer. Task debugging with temanejo. In Alexey Cheptsov, Steffen Brinkmann, José Gracia, Michael M. Resch, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2012*, pages 13–21. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-37348-0. doi: 10.1007/978-3-642-37349-7_2. URL http://dx.doi.org/10.1007/978-3-642-37349-7_2.

[19] Christoph Niethammer Steffen Brinkmann, José Gracia and Rainer Keller. TEMANEJO - a debugger for task based parallel programming models. *CoRR*, abs/1112.4604, 2011. URL http://arxiv.org/abs/1112.4604.

[20] Allinea. Allinea ddt: The debugger for c, c++ and f90 threaded and parallel code. URL http://www.allinea.com/.

[21] Gnu Project. Gdb, the gnu debugger. URL http://www.gnu.org/software/.

[22] UPC. Unified parallel c (upc). URL http://www.allinea.com/.

[23] RogueWave. Faster fault isolation, improved memory optimization, and dynamic visualization for your high performance computing apps. URL http://www.roguewave.com/.

[24] Kevin Pouget. Programming-model centric debugging for multicore embedded systems. 2014. URL https://tel.archives-ouvertes.fr/tel-01010061/file/these.pdf.

[25] J. Protze, T. Hilbrich, M. Schulz, B. R. d. Supinski, W. E. Nagel, and M. S. Mueller. Mpi runtime error detection with must: A scalable and crash-safe approach. In *2014 43rd International Conference on Parallel Processing Workshops*, pages 206–215, Sept 2014. doi: 10.1109/ICPPW.2014.37.

[26] Libevent is maintained by Nick Mathewson and Niels Provos. libevent – an event notification library. URL http://libevent.org/.

[27] Alexandre Eichenberger, John Mellor-Crummey, Martin Schulz, Nawal Copty, Jim Cownie, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz. Openmp technical report 2 on the ompt interface. URL http://openmp.org/mp-documents/ompt-tr2.pdf.

[28] Mathias Nachtmann and José Gracia. Enabling model-centric debugging for task-based programming models – a tasking control interface. In Andreas Knüpfer, Tobias Hilbrich, Christoph Niethammer, José Gracia, Wolfgang E. Nagel, and Michael M. Resch, editors, *Tools for High Performance Computing 2015, Proceedings of the 9th International Workshop on Parallel Tools for High Performance Computing, September 2015, Dresden, Germany*, pages 147–160. Springer, 2016.

[29] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Overlapping communication and computation by using a hybrid mpi/smpss approach. In *Proceedings of the 24th acm International Conference on Supercomputing*, pages 5–16. ACM, 2010.

[30] Marjanović. The mpi/ompss parallel programming model. 2015. URL https://upcommons.upc.edu/bitstream/handle/2117/98109/TVM1de1.pdf.