

HPC

Paramal

Malleability Traceability

The logo of the Universitat Oberta de Catalunya (UOC), consisting of the letters 'UOC' in a stylized, bold, blue font.

Francisco Rodríguez

TFM – High Performance Computing

Tutor's Name

Sergio Iserte

Subject Responsible

Josep Jorba

Delivery Date

31/08/2024

Universitat Oberta
de Catalunya



CC BY-NC-ND 4.0 DEED

Attribution-NonCommercial-NoDerivs 4.0 International

This work is subject to an Attribution-
NonCommercial-NoDerivs License 4.0
International

<https://creativecommons.org/licenses/by-nc-nd/4.0/>

FINAL WORKSHEET

Title:	<i>HPC Malleability Traceability</i>
Autor:	<i>Francisco Javier Rodriguez Olmeda</i>
Tutor:	<i>Sergio Iserte</i>
Area Responsible:	<i>Josep Jorba</i>
Delivery Date:	<i>09/2024</i>
University Degree:	<i>Master's Degree in Computational Engineering and Mathematics</i>
Master's Thesis Area:	<i>High-Performance Computing</i>
Language:	<i>English</i>
Keywords:	<i>Process, Malleability, Visual Inspection, Analysis Tool, Trace Generator</i>

Abstract

This project aims to develop a set of software tools to generate and analyze the execution of high-performance computing (HPC) malleable applications.

The software consists of two applications. The first application, the DMRTRACE is a library for recording traces in CSV format. It is designed to be integrated into the Barcelona Supercomputing Center's (BSC) malleability library, the DMR. The second application, DMRTRACEPARSER, is a utility to convert the CSV trace files to PRV trace files, to analyse traces using Paraver, the standard analysis tool in the BSC for HPC performance analysis.

The first part of this document explains the research and development process of the trace analysis tools. The second part is a step-by-step guide about how to use the developed tools to trace and analyze several HPC malleable applications.

The software tools have been developed to be used in the BSC. So that they have been developed using BSC standard development tools, procedures, and methodologies. The applications developed are open source, programmed in C++, and tested and validated in the BSC's supercomputing infrastructure, the MareNosendendtrum5. The source code is placed in the BSC's GitLab repository.

Index

1. Introduction	6
1.1. Context and justification of the master's thesis.....	6
1.2. Objectives of the master's thesis	7
1.3. Impact on the sustainability, ethic-social, and over the diversity .	7
1.4. Methodology.....	8
1.5. Project planning.....	9
1.6. Summary of products delivered in this master's thesis.....	11
1.7. Introduction to the rest of the master's thesis memory chapters	11
2. Technical concepts.....	12
2.1. BSC's cluster architecture	12
2.2. Access to the MareNostrum 5 login nodes.....	12
2.3. Running jobs in MareNostrum 5.....	14
2.4. Introduction to the malleability in HPC environments.....	16
2.5. DMR library technical fundamentals	16
2.6. Installing DMR in MareNostrum 5.....	18
2.7. Summary of the technical concepts section	19
3. Library for malleability trace generation	20
3.1. Trace generation library requirements.....	20
3.2. Definition of events to be traced.....	20
3.3. Definition of trace fields	21
3.4. Trace generation library design.....	22
3.5. Programming environment and tools	23
3.6. Source code management	25
3.7. Codification guidelines.....	26
3.8. Trace recording library development.....	28
3.9. Library mock-up.....	28
3.10. Library integration with DMR.....	29
3.11. Testing and validation.....	30
3.12. License Agreement	36
3.13. Summary of the trace generation library section	37
4. Trace format converter for Paraver	38
4.1. Format converter utility requirements	38
4.2. Paraver trace format	39
4.3. Strategy for DMRTRACE event conversion to Paraver format.....	41
4.4. Trace conversion application design	43
4.5. Data conversion rules definition.....	45
4.6. Trace conversion application development	46
4.7. Trace conversion application test and validation	48
4.8. Trace converter command line arguments.....	56
4.9. License Agreement.....	57
4.10. Summary of the trace converter application section	57
5. Study of several malleable applications	58
5.1. Application study procedure	58
5.2. Paraver procedures for application studies	60
5.3. Basic study of the sleepOf application	64
5.4. Study of the Jacobi application	69

5.5. Summary of the study of several malleable applications.....	79
6. Conclusions and future work.....	80
7. Glossary.....	83
8. Bibliography	84

List of Figures

FIGURE 1.1 PROJECT PLANNING.	10
FIGURE 2.1 MARENOSTRUM 5.	12
FIGURE 2.2 LOGIN INTO MARENOSTRUM5.	13
FIGURE 2.3 OUTPUT OF THE COMMAND LSCPU.	13
FIGURE 2.4 QUEUES AVAILABLE FOR THE USER.	14
FIGURE 2.5 EXAMPLE OF SBATCH EXECUTION IN MARENOSTRUM4.	15
FIGURE 2.6 EXECUTION ENVIRONMENT OF A MALLEABILITY APPLICATION USING DMR.	18
FIGURE 2.7 A SIMPLE EXAMPLE OF A MALLEABLE PROGRAM.	18
FIGURE 2.8 DMR FOLDER CONTENTS.	18
FIGURE 2.9 DMR EXAMPLE APPLICATION FOLDER.	19
FIGURE 2.10 DMR EXAMPLE EXECUTION OUTPUT IN MARENOSTRUM 5.	19
FIGURE 3.1 SELECTED SOURCE CODE DEVELOPMENT ENVIRONMENT.	25
FIGURE 3.2 DMR LIBRARY REPOSITORY IN GITLAB	25
FIGURE 3.3 GITLAB DMRTRACE LIBRARY REPOSITORY.	26
FIGURE 3.4 EXAMPLE OF SOURCE CODE FILE HEADER.	27
FIGURE 3.5 EXAMPLE OF FUNCTION CODIFICATION AND DOXYGEN DOCUMENTATION.	27
FIGURE 3.6 EXAMPLE OF ENUMERATION MEMBERS CODIFICATION.	27
FIGURE 3.7 DECLARATION OF THE FUNCTION FOR RECORDING TRACES.	28
FIGURE 3.8 TESTING FUNCTION FOR EVENTS RECORDING.	28
FIGURE 3.9 COMPILATION AND EXECUTION OF THE MOCK-UP.	29
FIGURE 3.10 MAKEFILE FOR TRACE GENERATION AND DMR LIBRARIES COMPILATION.	29
FIGURE 3.11 EXAMPLE OF USE OF THE TRACE GENERATION FUNCTION.	30
FIGURE 3.12 MAKEFILE FOR LINKING THE TRACE LIBRARY TO THE TEST APPLICATION.	30
FIGURE 3.13 A PORTION OF THE TRACE FILE OF THE EXAMPLE APPLICATION.	31
FIGURE 3.14 EXCEL VIEWER, VSCODE PLUGIN.	31
FIGURE 3.15 EXCEL VIEWER FILTERS FOR EVENTS VISUALIZATION.	32
FIGURE 3.16 EXAMPLE OF EVENTS IN THE FIRST ITERATIONS OF THE RANK0.	32
FIGURE 3.17 PYTHON AND JUPYTER NOTEBOOK PLUGINS FOR VSCODE.	33
FIGURE 3.18 PLOTTING OF ALL EVENTS OF ALL RANKS AND RESOURCES ALLOCATED.	34
FIGURE 3.19 RESOURCE ALLOCATION AND FIRSTS RANK 0 EVENTS.	34
FIGURE 3.20 PLOT OF SEND EVENTS OF DIFFERENT RANKS.	35
FIGURE 3.21 PLOT OF RECEIVE EVENTS OF DIFFERENT RANKS.	35
FIGURE 3.22 DETAILED PLOT OF THE SEND AND RECEIVE EVENTS.	36
FIGURE 3.23 LICENSE AGREEMENT IN THE HEADERS OF THE SOURCE CODE FILES.	37
FIGURE 4.1 EXAMPLE OF PARAVR TRACE FILE FORMAT.	39
FIGURE 4.2 PARAVR TRACE FILE HEADER DESCRIPTION.	39
FIGURE 4.3 STATE RECORD DESCRIPTION AND FORMAT.	40
FIGURE 4.4 EVENT RECORD DESCRIPTION AND FORMAT.	40
FIGURE 4.5 PARAVR OBJECT STRUCTURE.	41
FIGURE 4.6 DIRECT REPRESENTATION OF EVENTS IN PARAVR.	42
FIGURE 4.7 DIRECT REPRESENTATION OF STATES IN PARAVR.	42
FIGURE 4.8 PARAVR WINDOW PROPERTY FOR CREATING STATES FROM EVENTS.	43
FIGURE 4.9 REPRESENTATION OF EVENTS CONVERTED TO STATES IN PARAVR.	43
FIGURE 4.10 MAKEFILE FOR COMPILING THE TRACE CONVERSION APPLICATION.	46
FIGURE 4.11 TRACE CONVERTER APPLICATION COMPILATION ON WINDOWS.	47
FIGURE 4.12 EXECUTION OF THE CONVERSION APPLICATION ON WINDOWS AND LINUX.	47
FIGURE 4.13 ERROR DETECTION IN A WRONG CSV INPUT FILE.	48
FIGURE 4.14 NUMBER OF NODES IN THE CSV FILE FOR TEST AND VALIDATION.	49
FIGURE 4.15 THE NUMBER OF PROCESSORS IS ALWAYS THE SAME FOR ALL NODES.	49
FIGURE 4.16 THE RANKS ARE ALWAYS EXECUTED IN THE CPU 0 OF THE NODES.	49
FIGURE 4.17 ALL THE RANKS CREATE ONLY ONE EXECUTION THREAD.	50
FIGURE 4.18 THE NUMBER OF DIFFERENT PROCESSES EXECUTING THE RANK 0.	50

FIGURE 4.19 NUMBER OF RANKS EVERY TIME THAT AN ITERATION STARTS.	50
FIGURE 4.20 CONVERSION OF THE TESTING CSV TRACE FILE TO PRV FORMAT.	50
FIGURE 4.21 HEADER AND FIRST AND LAST EVENTS OF THE GENERATED PRV TRACE FILE.	51
FIGURE 4.22 COMPARISON BETWEEN REGISTERS OF THE CSV FILE AND THE PRV FILE.	52
FIGURE 4.23 MERGE OF TWO CSV TRACE FILES IN A UNIQUE PRV TRACE FILE.	52
FIGURE 4.24 PARAVR TRACE FILE HEADER AFTER MERGING TWO TIMES THE SAME FILE.	53
FIGURE 4.25 POINT OF THE PRV TRACE FILE WHERE TWO CSV TRACE FILES ARE JOINED.	53
FIGURE 4.26. PARAVR ERROR MESSAGE LOADING A WRONG TRACE FILE.	53
FIGURE 4.27 GENERATED TRACE FILE LOADED IN PARAVR.	54
FIGURE 4.28 EVENTS CONVERTED TO STATES IN PARAVR.	54
FIGURE 4.29 TIMELINE OF THE INSTANCE EXECUTION DIVIDED IN SECTIONS.	54
FIGURE 4.30 PARAVR TRACE FILE FROM TWO TRACE FILES.	55
FIGURE 4.31 PARAVR FILE FROM TWO TRACE FILES CONVERTED TO STATES.	56
FIGURE 5.1 RECONFIGURATION POLICY STEP UP TO 8.	59
FIGURE 5.2 OPENING A TRACE FILE IN PARAVR.	60
FIGURE 5.3 OPENING A TIMELINE WINDOW AND ACTIVATING THE EVENT FLAGS.	60
FIGURE 5.4 STATES OF THE JOB TIMELINE.	61
FIGURE 5.5 VISUALIZATION OF THE TIME SCALE IN MILLISECONDS.	61
FIGURE 5.6 VISUALIZATION OF THE JOB EXECUTION BY USING THE TASK LEVEL.	62
FIGURE 5.7 CREATION OF A HISTOGRAM WINDOW FOR STATISTICS.	62
FIGURE 5.8 STATISTICS WINDOW IN PARAVR.	63
FIGURE 5.9 FUNCTION COMPUTE OF THE SLEEPOF APPLICATION.	64
FIGURE 5.10 THREAD AND TASK VIEWS OF THE SLEEPOF APPLICATION.	64
FIGURE 5.11 RECONFIGURATIONS CYCLE FROM 1 TO 8 RESOURCES IN THE THREAD VIEW.	65
FIGURE 5.12 RECONFIGURATIONS CYCLE FROM 1 TO 8 RESOURCES IN TASK VIEW.	65
FIGURE 5.13 COMPUTATION TIMES DEPEND ON RESOURCES.	66
FIGURE 5.14 UNBALANCED COMMUNICATION TIMES.	67
FIGURE 5.15 PARALLELIZATION OF COMMUNICATIONS.	67
FIGURE 5.16 SYNCHRONOUS VERSUS ASYNCHRONOUS STATES.	68
FIGURE 5.17 STATISTICS OF THE SYNCHRONOUS AND ASYNCHRONOUS APPLICATIONS.	68
FIGURE 5.18. MAIN OPERATIONS OF THE JACOBI ALGORITHM.	69
FIGURE 5.19 PART OF THE JACOBI MALLEABLE ALGORITHM.	70
FIGURE 5.20 THREAD AND TASK VIEWS OF THE JACOBI APPLICATION.	70
FIGURE 5.21 RECONFIGURATIONS CYCLE FROM 1 TO 8 RESOURCES IN THE THREAD VIEW.	71
FIGURE 5.22. RECONFIGURATIONS CYCLE FROM 1 TO 8 RESOURCES IN TASK VIEW.	71
FIGURE 5.23. SIMULTANEOUS SEND AND RECEIVE COMMUNICATIONS.	72
FIGURE 5.24 ANALYSIS OF EVENTS WITHOUT STATE.	73
FIGURE 5.25 DMR_DETACH FUNCTION TRACEABILITY.	73
FIGURE 5.26 INDETERMINATE TIMES BETWEEN COMPUTATIONS.	73
FIGURE 5.27 MACRO DMR_COMPUTE OVERLOADED BY TRACE RECORDING.	74
FIGURE 5.28 UNBALANCED DATA TRANSFER TIMES IN JACOBI.	74
FIGURE 5.29 PARALLELIZATION OF THE FUNCTION SEND_EXPAND() IN JACOBI.	74
FIGURE 5.30 SYNCHRONOUS AND ASYNCHRONOUS DATA TRANSFERS.	75
FIGURE 5.31. SYNCHRONOUS VERSUS ASYNCHRONOUS COMPUTING TIMES.	75
FIGURE 5.32 SYNCHRONOUS VERSUS ASYNCHRONOUS COMPUTING STATISTICS.	76
FIGURE 5.33 RESOURCES POLICY FOR MULTIPLE INSTANCE EXECUTION.	76
FIGURE 5.34 RESOURCE SHARING CONFIGURATION VIA DMR MACRO.	77
FIGURE 5.35 EXECUTION OF MULTIPLE JACOBI INSTANCES IN PARALLEL.	77
FIGURE 5.36 TRACE FILES OF 4 INSTANCES IN PARALLEL.	77
FIGURE 5.37 THREAD AND TASK VIEW OF EXECUTION OF MULTIPLE INSTANCES OF JACOBI.	77
FIGURE 5.38. JOB EXECUTION DIVIDED INTO SECTIONS.	78
FIGURE 5.39. STATISTICS OF SIMULTANEOUS EXECUTION OF MULTIPLE INSTANCES.	79

List of tables

TABLE 1. TRACE GENERATION LIBRARY REQUIREMENTS.	20
TABLE 2. MALLEABILITY EVENTS CODIFICATION.	21
TABLE 3. FIELDS OF MALLEABILITY TRACES.	21
TABLE 4. TRACE LIBRARY VALIDATION RULES.	31
TABLE 5. TRACE CONVERTER APPLICATION REQUIREMENTS.	38
TABLE 6. MALLEABLE APPLICATION STUDY PROCEDURE.	58
TABLE 7. WORKLOAD DISTRIBUTION OF THE PROJECT.	81

1. Introduction

This document is a memory of a master's thesis of the studies of Master's Degree in Computational Engineering and Mathematics.

This master's thesis has been developed in the speciality of High-Performance Computing (HPC) subject. It has been made in collaboration with the Barcelona Supercomputing Centre – Centro Nacional de Supercomputación (BSC-CNS).

This project has aimed to develop a set of tools and guidelines to analyze the execution of HPC malleable applications.

The memory contains the development process of the traceability tools and several examples of how to use them. Some basic concepts have also been included to start working with the BSC's supercomputing infrastructure and the malleability library developed by the BSC's research team, the DMR.

This memory is complemented with the source code and binaries of the software applications developed during the master's thesis.

1.1. Context and justification of the master's thesis

HPC environments are very expensive infrastructures made up of thousands of computers interconnected to each other. These infrastructures are known as computer clusters. The clusters are usually composed of computers with the same characteristics.

From the point of view of the cluster users, there are two types of nodes, login nodes and compute nodes. Users access the login nodes and launch computing jobs requesting a fixed number of compute nodes.

Jobs are managed by job scheduling software which implements a determinate scheduling policy. In the most common way of working the scheduler reviews the job requirements and launches the job in the number of requested nodes when they are available, then the assigned nodes get locked in the job execution until it finishes.

The explained way of working has several disadvantages:

- Jobs execution must wait until the requested resources are available in the cluster.
- A job cannot use more than the resources requested at starting even if there will be more resources available.
- The scheduler blocks all the resources requested by a job until it finishes, even if the job does not need them anymore.

The ideal way of working would be to assign dynamic resources to a job execution to shorten the time that the job is waiting to be executed, to allow the use of more nodes when they are available, and to allow the liberate unnecessary nodes to leave them available to execute other jobs.

To solve this problem the BSC's Accelerator and Communications (AccelCom) team has developed a library that allows to ask for a range of resources instead of asking for a fixed number of resources. The library is the DMR library.

In this context, the aim and justification of this master's thesis is to develop a set of software tools to generate and analyze traces of malleable applications.

1.2. Objectives of the master's thesis

The main objectives of this master's thesis are the following:

- Reviewing some technical concepts regarding high-performance computing and how to work with the BSC's infrastructure.
- Learning about how to work with the BSC's malleability framework.
- Developing a library integrated into DMR to generate malleability traces.
- Developing a trace format converter to convert trace files to Paraver trace files.
- Analyzing the traces of several applications to use them as a reference about how to use the trace tools for application analysis.

1.3. Impact on the sustainability, ethic-social, and over the diversity

The optimization of the use of high-performance computing infrastructures has an impact on the environment:

- The thousands of computing nodes require large amounts of energy for power supply and cooling systems, which also transfer a lot of heat to the environment.
- The optimization of the efficiency of the infrastructure can extend its life cycle and reduce the environmental impact.
- Improving malleability techniques is translated into delivering more science by increasing supercomputers' productivity.

Due to the nature of the applications executed in high-performance environments, the optimization of the execution times can also have an impact on the research activities and, indirectly, on society.

1.4. Methodology

To achieve the objectives of this project the work has been divided into three phases.

The first phase is a training phase to acquire the basic knowledge needed to work with the HPC infrastructure of the BSC and to be able to manage the execution and compilation of parallel applications:

- Agreement of collaboration with the BSC-CNS to collaborate part-time with the BSC's AccelCom team. To have access to the BSC's HPC infrastructure and tools.
- Reviewing the learning materials and work done on the subject "High-Performance Computing" of the "Master's Degree in Computational Engineering and Mathematics".
- Reading the book "Construya su Propio Supercomputador con Raspberry PI", by Sergio Iserte et al., to get a wider scope of how a computer cluster and a job management system work.
- Experiment with a docker virtual cluster to know how Slurm works and experiment with parallel applications based on OpenMP and MPI.
- Accessing to the BSC supercomputer MareNostrum 5 and experiment with C++ source code compilation, job scheduling, and malleability.

The second phase is the development of the trace utilities:

- Experiment with the DMR library and review the source code to know how it works and the codification style.
- Defining useful trace codes.
- Defining the trace format and data.
- Designing and developing the trace generation library, DMRTRACE.
- Integrating the trace generation library in the DMR library and testing it.
- Documenting the trace library.
- Studying the analysis tool Paraver, and its trace file format.
- Designing and developing a trace file format converter for Paraver.
- Documenting the trace analysis application and file format converter.

The third phase of the project is the application of the developed traceability applications for tracing and analyzing several real malleable applications:

- Select several real malleable applications and study them.
- Executing the applications in the BSC's cluster and tracing them.
- Exporting the recorded traces to Paraver.
- Analyzing the execution by using Paraver.
- Documenting the trace recording and the data analysis performed.

1.5. Project planning

According to the methodology and tasks described, the project work can be divided into several milestones:

- Training and project planning.
- Trace generation library development.
- Trace format converter development.
- Tracing and analyzing several real applications.
- Master's thesis presentation.

Each milestone has been divided into several subtasks. In the case of the development activities are the common software development tasks:

- Playing with several mock-ups to help to design the application.
- Application design.
- Application programming.
- Application testing and validation.
- Application documentation.

The project will be developed in about 50 weeks with a dedication of about 15 hours per week. The total dedication to the project will be approximately 750 hours. Figure 1 shows the project planning.

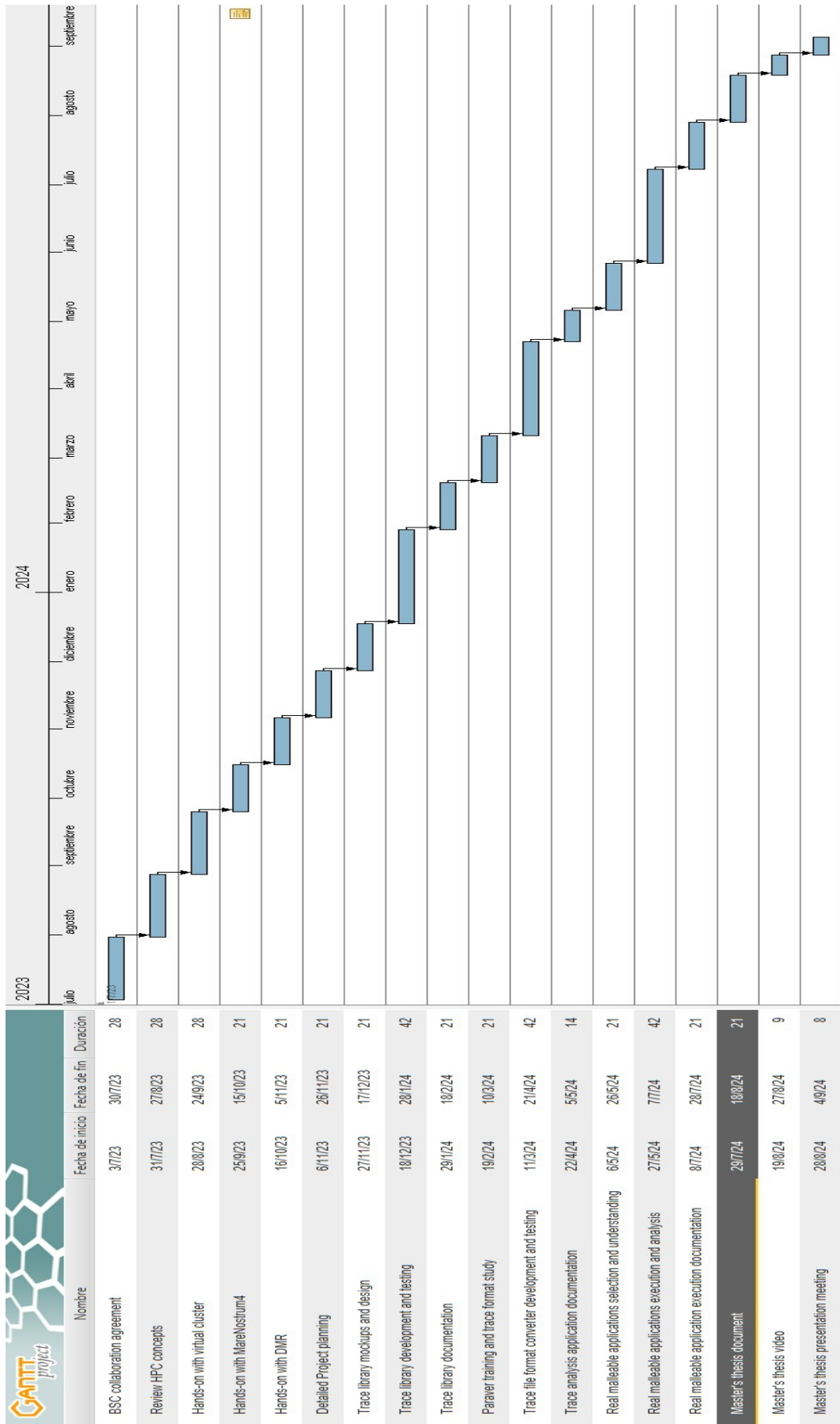


Figure 1.1 Project Planning.

To reach each project milestone, the following resources are available:

- Online documentation.
- Docker virtual cluster.
- BSC intranet documents.
- Access to the BSC's cluster.
- BSC's example source code.
- Software development tools, editors, compilers, etc.

1.6. Summary of products delivered in this master's thesis

The products delivered in this master's thesis will be two software applications and a step-by-step guideline for tracing and analyzing malleable applications:

- The application DMRTRACE, a C++ library for generating traces. This library will be integrated into the current BSC's malleability library DMR.
- The application DMRTRACEPARSER, a C++ application for converting CSV trace files to the Paraver analysis tool trace format.
- A set of examples of tracing and analysis of real malleable applications.
- The master thesis itself is a step-by-step guide for executing, tracing, and analyzing malleable applications.

1.7. Introduction to the rest of the master's thesis memory chapters

The next three sections will be dedicated to each one of the technical parts.

Chapter 2 is a summary of the training phase of the project. This section explains the basic concepts of HPC, malleability, and the BSC's cluster.

Chapter 3 contains the documentation regarding the design and development of the trace recording application.

Chapter 4 contains the documentation regarding the design and development of the trace conversion application.

Chapter 5 shows how to trace and analyze several malleable applications.

2. Technical concepts

This section reviews the technical concepts needed to start working with the BSC's infrastructure and with malleable applications. This section does not include the revision of general high-performance computing concepts.

2.1. BSC's cluster architecture

MareNostrum 5 is a pre-exascale EuroHPC supercomputer hosted at BSC-CNS. The system is supplied by Bull SAS combining Bull Sequana XH3000 and Lenovo ThinkSystem architectures and it has a total peak computational power of 314PFlops. The system will provide 4 partitions with different technical characteristics that jointly can fulfill the requirements of any HPC user.

The BSC's cluster architecture MareNostrum 5 is described in detail in the following link:

- <https://www.bsc.es/marenostrum/marenostrum-5>



Figure 2.1 MareNostrum 5.

2.2. Access to the MareNostrum 5 login nodes

The MareNostrum 5 has 3 login nodes:

- glogin1.bsc.es
- glogin2.bsc.es
- glogin4.bsc.es

The nodes glogin1 and glogin2 are public, however, the node glogin4 is only available from the computers logged into the BSC's network.

The login nodes glogin1 and glogin2 do not have access to computers out of the cluster. It means that all the operations regarding uploading or downloading files must be executed from the client machine. However, the node glogin4 is allowed to access external machines.

The access to the login nodes can be done via SSH connection, for instance, from a console of a client machine by executing the following command:

- `ssh username@glogin1.bsc.es`

```

Seleccioner bsc085551@glogin1:~
Users:USER@ssh bsc085551@glogin1.bsc.es
sc085551@glogin1.bsc.es's password:
BSC |
MareNostrum5

IMPORTANT NOTICE: Before executing any tasks, it is imperative
to review this document:
https://www.bsc.es/supportkc/docs/MareNostrum5/new_essentials/
as it contains crucial information regarding MareNostrum 4 data and
additional modifications.
- The final location for your old MM-Storage data is as follows:
  - /gpfs/home/<PRIMARY_GROUP>/USER/M4/<M4_USER>
  - /gpfs/projects/<GROUP>/M4/<GROUP>
  - /gpfs/scratch/<GROUP>/M4/<GROUP>/<M4_USER>
- User's Guide: https://www.bsc.es/supportkc/docs/MareNostrum5/intro/
- For further questions, don't hesitate to contact us at support@bsc.es

Last login: Sun May 26 15:17:20 2024 from 139.47.122.100
set INTEL compilers as MPI wrappers backend
load impi/2021.10.0 (PATH, MANPATH, LD_LIBRARY_PATH)
load mkl/2023.2.0 (LD_LIBRARY_PATH)
load ucx/1.15.0 (PATH, LD_LIBRARY_PATH, LIBRARY_PATH, C_INCLUDE_PATH, CPLUS_INCLUDE_PATH)
load bsc/1.0 (PATH, MANPATH)
load ucx/1.15.0 (PATH, LD_LIBRARY_PATH, LIBRARY_PATH, C_INCLUDE_PATH, CPLUS_INCLUDE_PATH)
remove mkl/2023.2.0 (LD_LIBRARY_PATH)
remove impi/2021.10.0 (PATH, MANPATH, LD_LIBRARY_PATH)
load bsc/1.0 (PATH, MANPATH)
Modules impi or openmpi not loaded, loading default openmpi
load PAV/7.3.0 (PATH, LD_LIBRARY_PATH, LIBRARY_PATH, C_INCLUDE_PATH, CPLUS_INCLUDE_PATH)
load d10/git (openmpi) (PATH, LD_LIBRARY_PATH, MANPATH, DLB_HOME)
loading km1/openmpi
MPI: /apps/GPP/DNR/mpich-3.2
DWR: /home/bsc085551/dwr
bsc085551@glogin1 ~]$
  
```

Figure 2.2 Login into MareNostrum5.

To examine the node architecture, the following command can be used:

- `lscpu`.

```

bsc085551@glogin1:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Address sizes:          46 bits physical, 57 bits virtual
Byte Order:             Little Endian
CPU(s):                224
On-line CPU(s) list:   0-223
Vendor ID:              GenuineIntel
Model name:             Intel(R) Xeon(R) Platinum 8480+
CPU family:             6
Model:                 143
Thread(s) per core:    2
Core(s) per socket:    56
Socket(s):              2
Stepping:               0
CPU max MHz:            3800.0000
CPU min MHz:            800.0000
BogoMIPS:               4000.00
Flags:                  fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fx
sr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts re
p_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 ds_cpl v
mx smx est tm2 sse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_dea
dline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l3 cat_l2 c
dp_l3 invpcid single_intel_ppin cdp_l2 ssbd mba ibrs ibpb stibp ibrs_enhanced tpr_shadow vmx_l
1xpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid cqm rdt_a avx5
12f avx512dq rdseed adx smap avx512ifma clflushopt clwb intel_pt avx512cd sha_ni avx512bw avx51
2vl xsaveopt xsavec xgetbv1 xsavec qm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local split_lo
ck
_detect avx_vnni avx512_bf16 wbnoinvd dtherm ida arat pln pts avx512vbmi umip pku ospke waitpkg
_avx512_vbmi2 gfn1 vaes vpclmulqdq avx512_vnni avx512_bitalg tme avx512_vpopcntdq la57 rdprid bu
s_lock_detect cldemote movdiri movdir64b enqcmd fsrm md_clear serialize tsxldtrk pconfig arch_l
br lbrt amx_bf16 avx512_fp16 amx_tile amx_int8 flush_l1d arch_capabilities

Virtualization features:
Virtualization:       VT-x
Caches (sum of all):
L1d:                   5.3 MiB (112 instances)
L1i:                   3.5 MiB (112 instances)
L2:                   224 MiB (112 instances)
L3:                   210 MiB (2 instances)
NUMA:
NUMA node(s):          2
NUMA node0 CPU(s):    0-55,112-167
NUMA node1 CPU(s):    56-111,168-223
Vulnerabilities:
Itlb multihit:         Not affected
L1tf:                  Not affected
Mds:                   Not affected
Meltdown:              Not affected
Mmio stale data:       Not affected
Retbleed:              Not affected
Spec store bypass:     Mitigation; Speculative Store Bypass disabled via prctl
Spectre v1:            Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Spectre v2:            Mitigation; Enhanced IBRS, IBPB conditional, RSB filling, PBRASB-eIBRS SW sequence
Srbds:                 Not affected
Tsx async abort:       Not affected
bsc085551@glogin1 ~]$
  
```

Figure 2.3 Output of the command lscpu.

For testing sbatch execution, a simple batch file can be created to execute the command hostname in two nodes:

```
#!/bin/bash
## launches two tasks in two nodes
#SBATCH --time=00:01:00
#SBATCH --job-name=hostname
#SBATCH --error=error.%j.log
#SBATCH --output=output.%j.log
#SBATCH --nodes=2
#SBATCH --ntasks=2
#SBATCH --cpus-per-task=1
#SBATCH --tasks-per-node=1
#SBATCH --ntasks-per-socket=1
#SBATCH --qos=debug
export TIMEFORMAT=%R
time srun hostname
unset TIMEFORMAT
srun echo $SLURM_JOB_NAME "-" $SLURM_JOB_ID
```

Saving those instructions, for instance, in a file called example.sbatch, can be executed in the following way:

- sbatch example.batch

```

bsc85551@login1:/home/bsc85/bsc85551/test2_serial
bsc85551@login1:~/test2_serial> cat tasks2-nodes2.batch
#!/bin/bash
## launches two tasks in two nodes
#SBATCH --time=00:01:00
#SBATCH --job-name=hostname
#SBATCH --error=error.%j.log
#SBATCH --output=output.%j.log
#SBATCH --nodes=2
#SBATCH --ntasks=2
#SBATCH --cpus-per-task=1
#SBATCH --tasks-per-node=1
#SBATCH --ntasks-per-socket=1
#SBATCH --qos=debug
export TIMEFORMAT=%R
time srun hostname
unset TIMEFORMAT
srun echo $SLURM_JOB_NAME "-" $SLURM_JOB_ID
bsc85551@login1:~/test2_serial> sbatch tasks2-nodes2.batch
Submitted batch job 30807792
bsc85551@login1:~/test2_serial> squeue
      JOBID PARTITION   NAME   USER  ST       TIME  NODES NODELIST(REASON)
      30807792      main hostname bsc85551 PD        0:00      2 (None)
bsc85551@login1:~/test2_serial> squeue
      JOBID PARTITION   NAME   USER  ST       TIME  NODES NODELIST(REASON)
bsc85551@login1:~/test2_serial> ls -l
total 4
-rw-r--r-- 1 bsc85551 bsc85  6 Nov 18 20:32 error.30807792.log
-rw-r--r-- 1 bsc85551 bsc85 58 Nov 18 20:32 output.30807792.log
-rw-r--r-- 1 bsc85551 bsc85 401 Nov 18 20:29 tasks2-nodes2.batch
-rwxr-xr-x 1 bsc85551 bsc85  97 Oct 22 15:28 tasks2-nodes2.sh
bsc85551@login1:~/test2_serial> cat *.log
0.261
s02r2b50
s02r2b67
hostname - 30807792
hostname - 30807792
bsc85551@login1:~/test2_serial>

```

Figure 2.5 Example of sbatch execution in MareNostrum4.

The state of the execution can be known by using the squeue command, and also the result of the execution, by examining the output file configured in the sbatch file.

In the following link, there are examples of how to execute different types of jobs in the MareNostrum 5:

- <https://www.bsc.es/supportkc/docs/MareNostrum5/slurm>

2.4. Introduction to the malleability in HPC environments

In an HPC environment, malleability is the capability of dynamically changing the resources assigned to the execution of a parallel application.

The possibility of dynamically assigning resources to the execution of a parallel application provides several benefits:

- Speeding up the job starting with a lower assigned number of resources before more HPC resources are available.
- Possibility of releasing resources instead of finalizing the execution of a parallel job when other more priority jobs require resources.
- Possibility of using more HPC resources to speed up a job execution when they are available.
- Possibility of using more HPC resources to increase the throughput of a job execution when they are available.
- Reaching a higher percentage of usability of the cluster when it is close to 100% of usability.

The use of dynamic resources for job executions translates directly into a better utilization of the HPC resources.

To use dynamic resources for job executions it is needed to communicate the job with the resource management system (RMS). The job communicates with the RMS to ask for resources and to know the real number of resources assigned. The RMS assigns resources to the job execution depending on the availability of them in the system. The job in execution must implement a reconfiguration mechanism capable of adapting the number of parallel tasks to the number of assigned resources in real time and continuing the execution without losing the work done.

2.5. DMR library technical fundamentals

The DMR library is the library developed by the BSC for implementing malleability in MareNostrum 5 by using slurm. In this section, the main features and basic technical aspects of the DMR are explained. The library is not explained in detail but just the basic concepts needed to understand its functionality. The DMR library is described in detail in several papers and documents published by the authors of the library. Several links are included at the end of this section to know in deep the technical details of the library.

The library consists of a C/C++ application which can be called from a parallel application to work in a malleable way. The library implements the needed functions to provide a parallel application with the capability of communicating with the RMS to allow the reconfiguration of the assigned resources in real time.

From the point of view of the user, the library has two main objectives:

- Allowing the expansion of a job execution when the RMS have more available resources for the execution.
- Allowing the shrinking of a job execution when the RMS needs to recover resources for other jobs.

The library provides a high level of abstraction allowing users to implement malleable applications without needing to understand technical aspects.

The library is designed to implement the described functionalities in the supercomputing environment of the BSC. For this reason, the library is based on several technologies used in the BSC:

- Slurm, a very common RMS in HPC environments.
- MPI, the most used library for distributed memory parallelization.
- Nanos++, a process control runtime developed by the BSC.

The library communicates with Slurm via a public application development interface (API). It is a set of libraries for low-level application development.

The DMR library uses a set of functions of the MPI library which allow to create and destroy child processes from a parent process..

It is not an objective of this section to describe in detail the features of Slurm and MPI which can be found on the internet in multiple sites:

- <https://slurm.schedmd.com/documentation.html>
- <https://www.open-mpi.org/>
- <https://pm.bsc.es/nanox>

The process used by the DMR library to manage the malleability of a parallel application can be summarized in the following steps:

- Reconfiguration starting based on parameters supplied by the user.
- New process creation based on the MPI's function `MPI_Comm_spawn`.
- Application data redistribution based on MPI's functions (`MPI_Send`, `MPI_Recv`, `MPI_Scatter`, `MPI_Gather`), or via some redistribution functions included in the DMR.
- Reconfiguration, termination of active processes, reallocation of resources, and execution of new processes, via Nanos++.

The execution environment is shown in Figure 2.15 [1]:

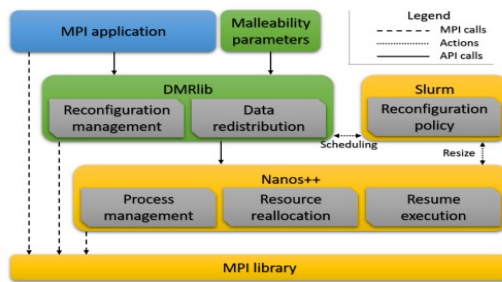


Figure 2.6 Execution environment of a malleability application using DMR.

The library is designed to be used by users in a low-intrusive way. Users have only to add some source code lines in their applications and add a link to the DMR library to them. Then, transforming a fixed resources application into a malleable application becomes an easy task.

The structure and an example of a basic program implementing malleability via DMR library is shown in Figure 2-16 [1]:

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    int world_rank, world_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int len;
    char name[MPI_MAX_PROCESSOR_NAME];
    MPI_Get_processor_name(name, &len);

    double *vector;

    DMR_INIT(initialize_data(&vector, world_size, world_rank));
    while (DMR_it < STEPS) {
        compute(DMR_it, world_size, world_rank);
        DMR_it++;
        DMR_RECONFIGURATION();
    }
    DMR_FINALIZE();

    free(vector);
    MPI_Finalize();
    return 0;
}
```

Figure 2.7 A simple example of a malleable program.

2.6. Installing DMR in MareNostrum 5

To use the DMR library it is needed to install and configure several applications and dependencies in the user workspace. The complete procedure of the library installation is described in Confluence in a private space of the BSC's staff.

The installation and configuration procedure creates a folder with the DMR library, dependencies, and examples:

```

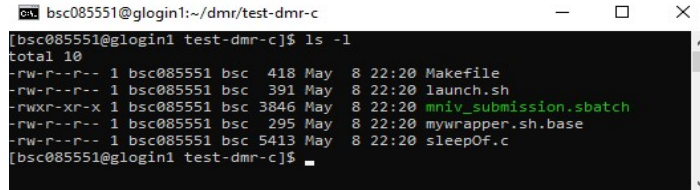
bnc bsc085551@glogin1:~/dmr
[bsc085551@glogin1 ~]$ cd dmr
[bsc085551@glogin1 dmr]$ ls -la
total 56
drwxr-xr-x  5 bsc085551 bsc  4096 May  8 22:20 .
drwxr-xr-x 10 bsc085551 bsc  4096 Jun  8 16:18 ..
drwxr-xr-x  8 bsc085551 bsc  4096 May  8 22:20 .git
-rw-r--r--  1 bsc085551 bsc    99 May  8 22:20 .gitignore
-rw-r--r--  1 bsc085551 bsc   496 May  8 22:20 Makefile
-rwxr-xr-x  1 bsc085551 bsc 20474 May  8 22:20 dmr.c
-rwxr-xr-x  1 bsc085551 bsc 23711 May  8 22:20 dmr.h
-rw-r--r--  1 bsc085551 bsc   295 May  8 22:20 mywrapper.sh.base
drwxr-xr-x  8 bsc085551 bsc  4096 May 11 20:05 slurm-spawn
lrwxrwxrwx  1 bsc085551 bsc    53 May  8 22:20 slurm_select_linear.c
drwxr-xr-x  2 bsc085551 bsc  4096 May 11 09:08 test-dmr-c
[bsc085551@glogin1 dmr]$

```

Figure 2.8 DMR folder contents.

The DMR library is supplied with an example application, sleepOf, and it can be compiled by using a Makefile application by using the following commands:

- `cd $DMR_PATH/test-dmr-c/`
- `make`

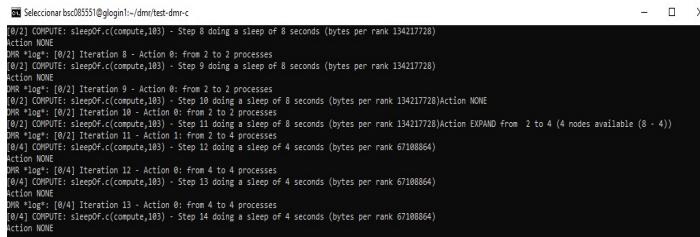


```
bsc085551@glogin1:~/dmr/test-dmr-c
[bsc085551@glogin1 test-dmr-c]$ ls -l
total 10
-rw-r--r-- 1 bsc085551 bsc 418 May  8 22:20 Makefile
-rw-r--r-- 1 bsc085551 bsc 391 May  8 22:20 launch.sh
-rwxr-xr-x 1 bsc085551 bsc 3846 May  8 22:20 mniv_submission.sbatch
-rw-r--r-- 1 bsc085551 bsc 295 May  8 22:20 mywrapper.sh.base
-rw-r--r-- 1 bsc085551 bsc 5413 May  8 22:20 sleepOf.c
[bsc085551@glogin1 test-dmr-c]$
```

Figure 2.9 DMR example application folder.

The example application can be executed by using a sbatch file by using the sbatch command:

- `sbatch mniv_submission.sbatch`



```
Selecionar bsc085551@glogin1:~/dmr/test-dmr-c
[0/2] COMPUTE: sleepOf.c(compute,183) - Step 8 doing a sleep of 8 seconds (bytes per rank 134217728)
Action NONE
MWR *log*: [0/2] Iteration 8 - Action 0: from 2 to 2 processes
[0/2] COMPUTE: sleepOf.c(compute,183) - Step 9 doing a sleep of 8 seconds (bytes per rank 134217728)
Action NONE
MWR *log*: [0/2] Iteration 9 - Action 0: from 2 to 2 processes
[0/2] COMPUTE: sleepOf.c(compute,183) - Step 10 doing a sleep of 8 seconds (bytes per rank 134217728)Action NONE
MWR *log*: [0/2] Iteration 10 - Action 0: from 2 to 2 processes
[0/2] COMPUTE: sleepOf.c(compute,183) - Step 11 doing a sleep of 8 seconds (bytes per rank 134217728)Action EXPAND from 2 to 4 (4 nodes available (8 - 4))
MWR *log*: [0/2] Iteration 11 - Action 1: from 2 to 4 processes
[0/4] COMPUTE: sleepOf.c(compute,183) - Step 12 doing a sleep of 4 seconds (bytes per rank 67108864)
Action NONE
MWR *log*: [0/4] Iteration 12 - Action 0: from 4 to 4 processes
[0/4] COMPUTE: sleepOf.c(compute,183) - Step 13 doing a sleep of 4 seconds (bytes per rank 67108864)
Action NONE
MWR *log*: [0/4] Iteration 13 - Action 0: from 4 to 4 processes
[0/4] COMPUTE: sleepOf.c(compute,183) - Step 14 doing a sleep of 4 seconds (bytes per rank 67108864)
Action NONE
```

Figure 2.10 DMR example execution output in MareNostrum 5.

2.7. Summary of the technical concepts section

In this section, it has been explained the technical concepts and the work done to be able to work with the BSC supercomputing infrastructure, the MareNostrum 5, and the virtual cluster, a useful tool for experimenting without a real cluster. They have also been explained the basic technical concepts under the DMR library, the malleability library of the BSC,

This preliminary work is a useful guideline to get the basic technical knowledge about the BSC's infrastructure. It is also a fundamental set of knowledge needed to start the design and development of the malleability trace recording and analysis applications.

3. Library for malleability trace generation

This section is dedicated to the design and development of the library for malleability trace generation.

3.1. Trace generation library requirements

In this section, the design requirements of the library for trace generation are defined. The library must be able to generate traces in malleable applications and must be integrated into the BSC's malleability library, DMR.

There are other important requirements to consider regarding functionalities, codification, and application development standards.

The set of requirements are the ones shown in Table 1.

Requirements
Must trace each malleability event of an application execution
Trace files must be in CSV text format easy to read and exporting
Only ASCII characters will be allowed in trace files
Trace files must be generated in the folder where the application is executed
The library must be Integrated within the BSC's DMR library
The development language must be C/C++
The library must be compiled as a shared library
The library must be an opensource application
The codification must follow standards, according to the DMR coding style
Codification and commentaries must be in English
The source code must be hosted in the BSC's GitLab repository
Provide documentation in English

Table 1. Trace generation library requirements.

3.2. Definition of events to be traced

Traces must contain the right data to allow users to extract useful information about the execution process of a malleable application. To fulfil this requirement the adopted strategy will be recording execution events, such as data transfers, reconfigurations, active resources, etc.

The list of the defined malleability events is shown in Table 2.

Code	Description
0	Set up
1	Start Initialization
2	End Initialization
3	Start to Receive

4	End Receive
5	Start to Send
6	End Send
7	Start Reconfiguration
8	End Reconfiguration
9	Start Iteration
10	End Iteration
11	Start Finalization
12	End Finalization
13	Start Detaching
14	End Detaching

Table 2. Malleability events codification.

The trace generation library must create a new record in the trace file each time one of the defined malleability events occurs.

3.3. Definition of trace fields

One of the most interesting analyses to perform over the data recorded will be execution times. By recording execution times, it will be possible to get metrics of performance and overhead. To calculate the execution times, the best strategy consists of recording the timestamps to each event to trace.

Other important information to be recorded are the processes that generate the events and how the resources allocated for the application change along the execution. This information can be obtained by recording process identifications, ranks, and number of ranks when an event is generated.

It is also important to record in traces the information about the infrastructure where a job has been executed, such as execution nodes and processors.

The defined trace fields will be the ones shown in Table 3.

Code	Description
Node Name	Name of the node
Node CPUs	Number of CPUs in the node
CPU Id	The numeric identifier of the CPU
Process Id	The numeric identifier of the process
Thread Id	The numeric identifier of the thread
DateTime	Timestamp of the event, in DateTime format
Time	Timestamp of the event, in seconds
Event	The numeric identifier of the event
Event Desc	Text description of the event
Rank	Rank in which the event occurs
Num Ranks	Number of ranks in execution
Iteration	Iteration of the application in which the event occurs

Table 3. Fields of malleability traces.

3.4. Trace generation library design

The trace generation library must be designed according to the user requirements, the defined events to trace, and the defined trace fields.

The library will have the following functionalities:

- Enumeration and description of trace events.
- Getting the identifiers of the node, cpu, process, and thread in execution where the event is generated.
- Getting the identifier of the rank that generates an event.
- Getting the iteration of the program where the event is generated.
- Getting the number of ranks in execution when the event is generated.
- Getting the timestamp when an event is generated.
- Creating the trace with the defined fields.
- Recording the trace in a trace file.

From the user requirements, there are other important considerations regarding the type of application and development process:

- The library must be integrated into the DMR library.
- The library must be developed in C/C++.
- The codification style must be according to the DMR codification style.

Based on those considerations, the trace library has been designed to be integrated into the DMR library, and it will call a trace generation function each time a relevant event occurs:

Two types of functions have been defined in the trace library.

- Exported functions to be called from the DMR library.
- Local functions to generate trace data.

Some of the data needed for the traces are known from the DMR library, these data will be transferred to the trace functions.

The following data can be transferred from the DMR library to the trace library:

- The Rank that generates the event.
- Number of ranks in execution.
- Iteration where the event is generated.

At this point, a design decision must be taken. One of the options is to use a unique function to create the traces and send the event to record as a parameter of the function, and another option is to create a different function for each type of event to record. Each option has pros and cons and generates different function prototypes.

Function prototype by using a unique function:

```
generate_event ( EVENT_ID_1, rank, nrank, iteration )  
.  
generate_event ( EVENT_ID_N, rank, nrank, iteration )
```

Function prototype by using multiple functions.

```
generate_event_function_1 ( rank, nrank, iteration )  
.  
generate_event_function_N ( rank, nrank, iteration )
```

After tasting the use of both approaches in a mock-up, the decision was to use a unique function and pass the event as a parameter. The final function definition will be the following one:

```
int dmrtrace_write_event(event_t event,  
                        int dmr_comm_rank,  
                        int dmr_comm_size,  
                        int dmr_it);
```

The event definition enumeration will be the following:

```
enum event_t  
{  
    EVENT_SETUP = 0,  
    EVENT_INIT_START,  
    EVENT_INIT_END,  
    EVENT_RECV_START,  
    EVENT_RECV_END,  
    EVENT_SEND_START,  
    EVENT_SEND_END,  
    EVENT_RECONFIGURATION_START,  
    EVENT_RECONFIGURATION_END,  
    EVENT_ITERATION_START,  
    EVENT_ITERATION_END,  
    EVENT_FINALIZE_START,  
    EVENT_FINALIZE_END,  
    EVENT_DETACHING_START,  
    EVENT_DETACHING_END  
};
```

3.5. Programming environment and tools

It is necessary to select a programming environment for the library development. The right selection can save a lot of development time.

The application is a small library developed in C/C++ languages, with just a couple of files, and with a few numbers of functions. Based on those premises it will be possible to define a list of requirements for the development tools:

- C/C++ syntax highlighting.
- C/C++ compilation and debugging.
- Possibility of remote editing via SSH connection.
- Multiplatform.
- Open Source.
- CMake tools.

Several open-source code editors and integrated development environments have been reviewed and evaluated. The following ones have been selected for the final decision, all of them fulfil the requirements:

- Visual Studio Code (<https://code.visualstudio.com/>)
- CodeBlocks (<https://www.codeblocks.org/>)
- Netbeans (<https://netbeans.apache.org/front/main/>)
- Eclipse (<https://eclipseide.org/>)

The selected development environment has been Visual Studio Code. It is a lightweight modern multiplatform and multilanguage source code editor. It includes compilation, debugging, and a lot of useful plugins.

Some Visual Studio Code plugins have been used for the development:

- Remote SSH.
(<https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.remote-ssh>)
- C/C++ for Visual Studio Code.
(<https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools>)
- C/C++ Extension Pack.
(<https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools-extension-pack>)
- CMake for Visual Studio Code.
(<https://marketplace.visualstudio.com/items?itemName=twxs.cmake>)
- CMake Tools.
(<https://marketplace.visualstudio.com/items?itemName=ms-vscode.cmake-tools>)
- VS Code Makefile Tools.
(<https://marketplace.visualstudio.com/items?itemName=ms-vscode.makefile-tools>)
- VSCode C/C++ Runner.
(<https://marketplace.visualstudio.com/items?itemName=franneck94.c-cpp-runner>)
- VS Makefile Tools.
(<https://marketplace.visualstudio.com/items?itemName=ms-vscode.makefile-tools>)
- Doxygen Documentation Generation.
(<https://marketplace.visualstudio.com/items?itemName=cshlosser.doxdocgen>)

Figure 3-1 shows an image of the development environment connected to the cluster. The left side shows some of the installed plugins.

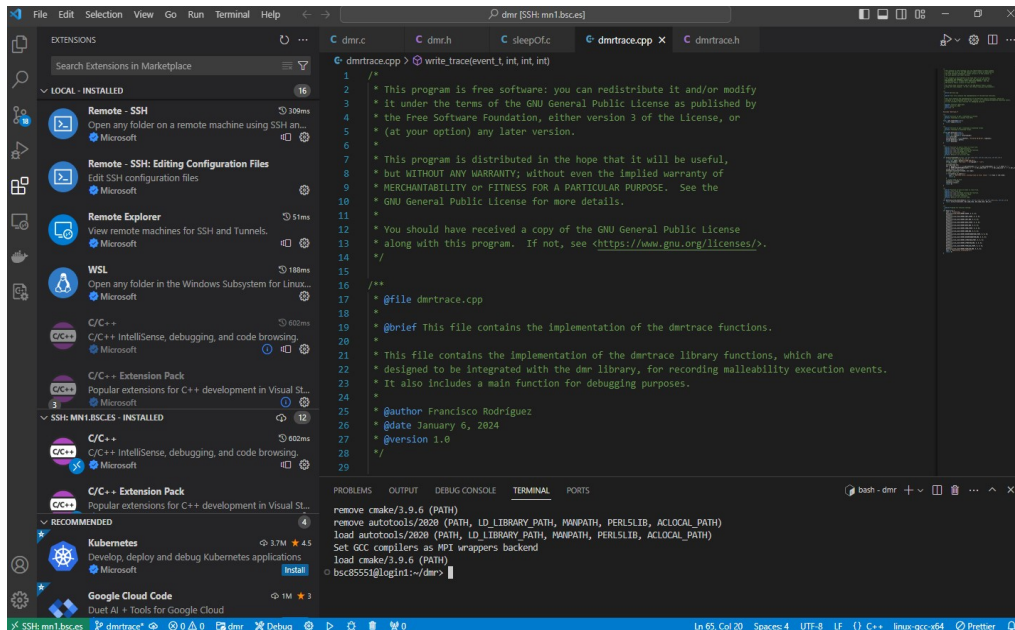


Figure 3.1 Selected source code development environment.

3.6. Source code management

The trace generation library will be a utility for tracing malleable applications. One of the requirements is to be useful for the BSC so the source code must be managed professionally following the standards of source code management.

The source code is managed at BSC via a source code repository based on GitLab. The DMR library source code is stored and maintained in a GitLab repository, as shown in Figure 3-2.

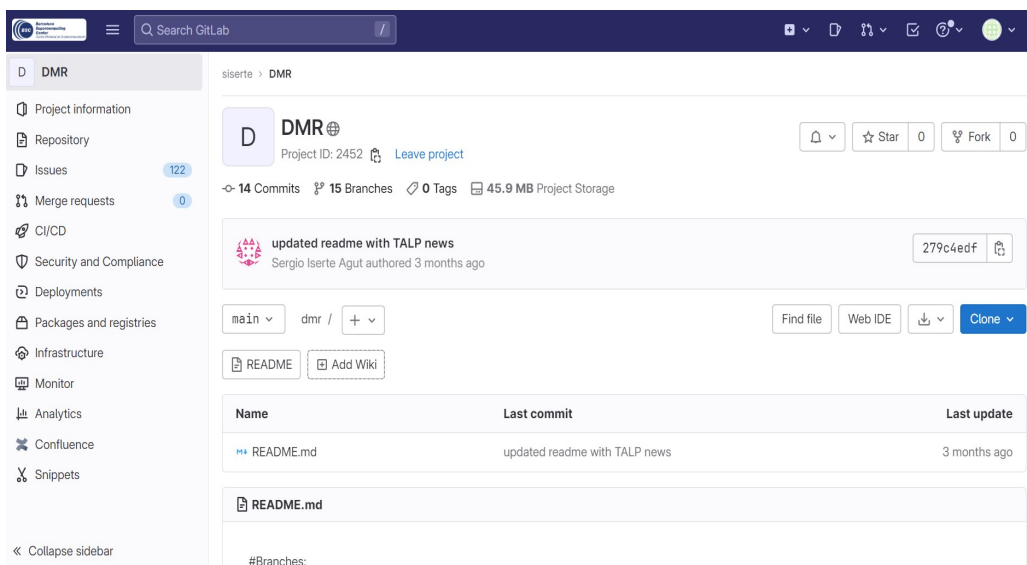


Figure 3.2 DMR library repository in GitLab

The best way to manage the source code for the trace generation library is to handle the code as a new branch of the DMR library. The new branch is called DMRTRACE and will contain the library source code.

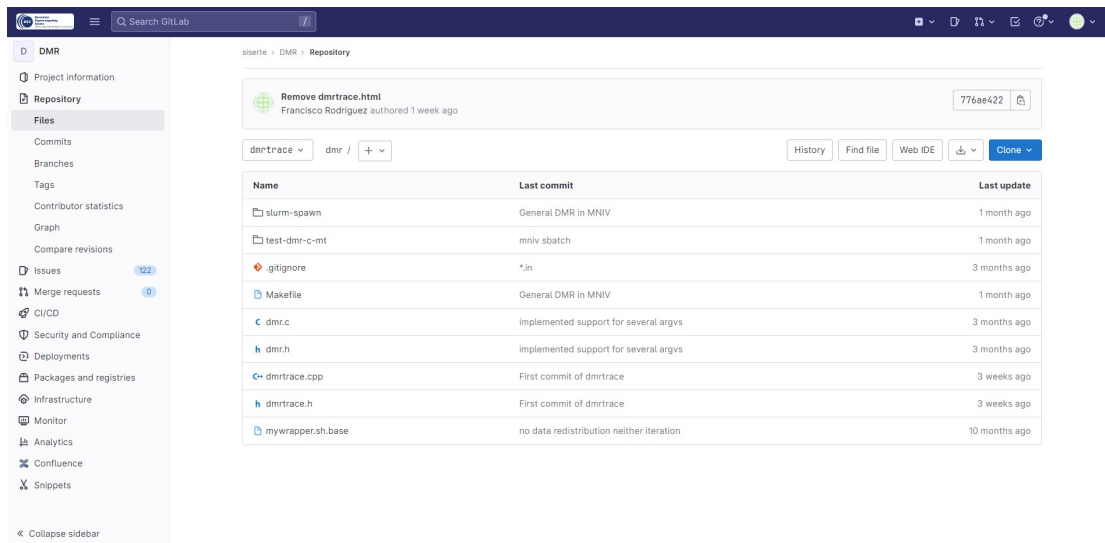


Figure 3.3 GitLab dmrtrace library repository.

3.7. Codification guidelines

Another important topic before developing the library source code is to decide the codification guidelines.

Due to the trace library being part of the DMR library and the source code being used by the same users, it will be necessary to keep a uniform codification style. To meet this requirement, it is necessary to review the DMR library's source code. After analyzing the DMR library's codification style the following codification rules were adopted:

- Snack case style, for function and variable names, where the name is a composition of lower-case words separated by underscores. It is a very usual codification style in C/C++ languages.
- SCREAMING_SNAKE_CASE style, for constants and enumeration members. It is also a common codification style in C/C++ languages.
- Doxygen-compatible commentaries. It will allow to generate automatic documentation of the library's functions.
- License explanation at the header of each file.

Some examples of the codification styles are shown in the following figures.

```
/*
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program. If not, see <https://www.gnu.org/licenses/>.
 */

/**
 * @file dmrtrace.cpp
 *
 * @brief This file contains the implementation of the dmrtrace functions.
 *
 * This file contains the implementation of the dmrtrace library functions, which are
 * designed to be integrated with the dmr library, for recording malleability execution events.
 * It also includes a main function for debugging purposes.
 *
 * @author Francisco Rodríguez
 * @date January 6, 2024
 * @version 1.0
 */
```

Figure 3.4 Example of source code file header.

```
/**
 * @brief Function to save an event in trace file.
 * @param trace Trace Id.
 * @param dmr_comm_rank Rank calling the function.
 * @param dmr_comm_size Number of ranks.
 * @param dmr_it Current iteration.
 * @return Returns 0 or an error code.
 */
int dmrtrace_write_event(event_t event, int dmr_comm_rank, int dmr_comm_size, int dmr_it) {
    return write_trace(event, dmr_comm_rank, dmr_comm_rank, dmr_it);
}
```

Figure 3.5 Example of function codification and Doxygen documentation.

```
/**
 * @brief Enumeration of event codes.
 */
enum event_t
{
    EVENT_SETUP = 0,           /**< @brief Set up. */
    EVENT_INIT_START,         /**< @brief Start Initialization. */
    EVENT_INIT_END,           /**< @brief End Initialization. */
    EVENT_RECV_START,         /**< @brief Start Receive. */
    EVENT_RECV_END,           /**< @brief End Receive. */
    EVENT_SEND_START,         /**< @brief Start Send. */
    EVENT_SEND_END,           /**< @brief End Send. */
    EVENT_RECONFIGURATION_START, /**< @brief Start Reconfiguration. */
    EVENT_RECONFIGURATION_END, /**< @brief End Reconfiguration. */
    EVENT_ITERATION_START,    /**< @brief Start Iteration. */
    EVENT_ITERATION_END,      /**< @brief End Iteration. */
    EVENT_FINALIZE_START,     /**< @brief Start Finalization. */
    EVENT_FINALIZE_END        /**< @brief End Finalization. */
};
```

Figure 3.6 Example of enumeration members codification.

3.8. Trace recording library development

After the detailed definition of requirements and the development and source code management tools selection, all the prerequisites are accomplished to be able to start the library development.

The library consists of a C/C++ source code file, “dmrtrace.cpp” with several functions and a C/C++ header file, “dmrtrace.h”.

The unique function to call from the DMR library for recording traces is the “dmrtrace_write_event” function, for this reason, it is declared in the header file. It will allow to call the function from an external source code just including the header. The function is declared as “extern C” to keep compatibility with C and C++ compiled code.

The declaration of the function “dmrtrace_write_event” is the following one.

```
// Trace functions
extern "C"
{
    int dmrtrace_write_event(event_t event, int dmr_comm_rank, int dmr_comm_size, int dmr_it); /**< @brief Function to save an event in trace file. */
}
```

Figure 3.7 Declaration of the function for recording traces.

3.9. Library mock-up

For testing the trace recording function before linking it with the DMR library, the trace recording library includes a “main” program to write all the possible events in a file. The mock-up consists of compiling the source code as an executable instead of a library and executing it to test the functionality.

```
/**
 * @brief Program for function testing.
 */
int main(void)
{
    cout << "Processing... \n";
    dmrtrace_write_event(EVENT_SETUP, 0, 0, 0);
    sleep(1);
    dmrtrace_write_event(EVENT_INIT_START, 0, 0, 0);
    sleep(1);
    dmrtrace_write_event(EVENT_INIT_END, 0, 0, 0);
    sleep(1);
    dmrtrace_write_event(EVENT_RECV_START, 0, 0, 0);
    sleep(1);
    dmrtrace_write_event(EVENT_RECV_END, 0, 0, 0);
    sleep(1);
    dmrtrace_write_event(EVENT_SEND_START, 0, 0, 0);
    sleep(1);
    dmrtrace_write_event(EVENT_SEND_END, 0, 0, 0);
    sleep(1);
    dmrtrace_write_event(EVENT_RECONFIGURATION_START, 0, 0, 0);
    sleep(1);
    dmrtrace_write_event(EVENT_RECONFIGURATION_END, 0, 0, 0);
    sleep(1);
    dmrtrace_write_event(EVENT_ITERATION_START, 0, 0, 0);
    sleep(1);
    dmrtrace_write_event(EVENT_ITERATION_END, 0, 0, 0);
    sleep(1);
    dmrtrace_write_event(EVENT_FINALIZE_START, 0, 0, 0);
    sleep(1);
    dmrtrace_write_event(EVENT_FINALIZE_END, 0, 0, 0);
    cout << "Application terminated!\n";
    return 0;
}
```

Figure 3.8 Testing function for events recording.

The library can be compiled as an executable executing the following command:

```
g++ dmrtrace.cpp -o dmrtrace
```

Executing it on any Linux platform a CSV file is generated. It contains one trace line for each defined event.

Figure 3-9 shows the compilation and execution of the executable application used as a mock-up.

```

bsc85551@login1:~/dmr> g++ dmrtrace.cpp -o dmrtrace
bsc85551@login1:~/dmr> ./dmrtrace
Processing...
Application terminated!
bsc85551@login1:~/dmr> ls *.csv
dmrtrace-test.csv
bsc85551@login1:~/dmr> cat dmrtrace-test.csv
187447;2024-01-27 21:34:03;1706387643;0;Setup;0;0;0;
187447;2024-01-27 21:34:04;1706387644;1;Init Start;0;0;0;
187447;2024-01-27 21:34:05;1706387645;2;Init End;0;0;0;
187447;2024-01-27 21:34:06;1706387646;3;Receive Start;0;0;0;
187447;2024-01-27 21:34:07;1706387647;4;Receive End;0;0;0;
187447;2024-01-27 21:34:08;1706387648;5;Send Start;0;0;0;
187447;2024-01-27 21:34:09;1706387649;6;Send End;0;0;0;
187447;2024-01-27 21:34:10;1706387650;7;Reconfigure Start;0;0;0;
187447;2024-01-27 21:34:11;1706387651;8;Reconfigure End;0;0;0;
187447;2024-01-27 21:34:12;1706387652;9;Iteration Start;0;0;0;
187447;2024-01-27 21:34:13;1706387653;10;Iteration End;0;0;0;
187447;2024-01-27 21:34:14;1706387654;11;Finalize Start;0;0;0;
187447;2024-01-27 21:34:15;1706387655;12;Finalize End;0;0;0;
202970;2024-01-27 21:37:23;1706387843;0;Setup;0;0;0;
202970;2024-01-27 21:37:24;1706387844;1;Init Start;0;0;0;
202970;2024-01-27 21:37:25;1706387845;2;Init End;0;0;0;
202970;2024-01-27 21:37:26;1706387846;3;Receive Start;0;0;0;
202970;2024-01-27 21:37:27;1706387847;4;Receive End;0;0;0;
202970;2024-01-27 21:37:28;1706387848;5;Send Start;0;0;0;
202970;2024-01-27 21:37:29;1706387849;6;Send End;0;0;0;
202970;2024-01-27 21:37:30;1706387850;7;Reconfigure Start;0;0;0;
202970;2024-01-27 21:37:31;1706387851;8;Reconfigure End;0;0;0;
202970;2024-01-27 21:37:32;1706387852;9;Iteration Start;0;0;0;
202970;2024-01-27 21:37:33;1706387853;10;Iteration End;0;0;0;
202970;2024-01-27 21:37:34;1706387854;11;Finalize Start;0;0;0;
202970;2024-01-27 21:37:35;1706387855;12;Finalize End;0;0;0;
bsc85551@login1:~/dmr>

```

Figure 3.9 Compilation and execution of the mock-up.

3.10. Library integration with DMR

After developing and testing basic functionalities via a mock-up the trace generation library is ready to be integrated into the DMR library to test the complete functionality.

First, it is necessary to define how to compile the source code to be compatible with the DMR library. The DMR library is usually compiled as a dynamic link library so that the trace library will be compiled in the same way.

The Makefile included in the DMR project folder includes the information about how the DMR library is compiled. The rules to compile both libraries can be added in the Makefile, first the trace library and after the DMR library, including the trace library.

```

1 Makefile
2 DMRFLAGS      = -I${DMR_PATH} -L${DMR_PATH} -ldmr
3 FLAGS         = -O3 -Wall
4 MPIFLAGS      = -I/apps/dmr/mpich-3.2/include -L/apps/dmr/mpich-3.2/lib -lmpi
5 SLURMFLAGS    = -I${DMR_PATH}/slurm-install/include -L${DMR_PATH}/slurm-install/lib -lslurm
6
7 all: trace lib
8
9 trace: dmrtrace.cpp
10      g++ -shared -fPIC -o libdmrtrace.so dmrtrace.cpp
11
12 lib: dmr.c
13      g++ $(MPIFLAGS) $(SLURMFLAGS) -c -fPIC dmr.c -o dmr.o
14      g++ dmr.o -shared -o libdmr.so -L${DMR_PATH} -ldmrtrace
15
16 clean:
17      rm -f *.out *.o core.* *.err *.so tmp

```

Figure 3.10 Makefile for trace generation and DMR libraries compilation.

It has added a rule and modified the compilation parameters to compile the trace library as a shared library with the DMR library.

The calls to the trace functions must be included inside the DMR library. It is one of the more difficult tasks of the project because it requires understanding how the DMR library works and how its source code is organised.

Each time a malleable process calls the library to change the allocated resources for the application execution, some processes are destroyed, and new ones are created. This behaviour makes it difficult to keep the persistence of the trace variables between different malleability iterations.

After examining the DMR library source code and experimenting with different mock-ups, it has been possible to integrate all the calls to the trace generation function in the DMR source code header file, the “dmr.h” file. In this way, it is not necessary to add extra source code to the malleable program, just by using the DMR library in the usual way the traces will generated automatically.

```
#define DMR_COMPUTE(compute) \
{ \
    dmtrace_write_event(EVENT_ITERATION_START, DMR_comm_rank, DMR_comm_size, DMR_it); \
    compute; \
    dmtrace_write_event(EVENT_ITERATION_END, DMR_comm_rank, DMR_comm_size, DMR_it); \
}

#define DMR_FINALIZE(finalize) \
{ \
    dmtrace_write_event(EVENT_FINALIZE_START, DMR_comm_rank, DMR_comm_size, DMR_it); \
    finalize; \
    dmtrace_write_event(EVENT_FINALIZE_END, DMR_comm_rank, DMR_comm_size, DMR_it); \
    if (DMR_comm_rank == 0) \
        printf("DMR *log*: %s(%s,%d) - Execution succesful!\n", __FILE__, __func__, __LINE__); \
}
```

Figure 3.11 Example of use of the trace generation function.

3.11. Testing and validation

The final task of the development process consists of testing and validating the trace generation by using it in a parallel malleable application.

The example program included in the DMR library source code has been used to validate the traces. It is a simple program to understand how the malleability library works. The program launches an instruction “sleep()” in several nodes in parallel. The sleep time depends on the number of resources assigned. The resource reconfiguration is done via the macros defined in the DMR library.

There is no need to modify the source code to generate traces, the code to do that is inside the DMR library macros. The code is transparent for the developer. However, it is necessary to link the trace library with the application in the same way that the DMR library is linked.

```
1 DMRFLAGS = -I${DMR_PATH} -L${DMR_PATH} -ldmr -ldmtrace
2 FLAGS = -O3 -Wall
3 MPIFLAGS = -I/apps/dmr/mpich-3.2/include -L/apps/dmr/mpich-3.2/lib -lmpi
4 SLURMFLAGS = -I${DMR_PATH}/slurm-install/include -L${DMR_PATH}/slurm-install/lib -lslurm
5
6 all: clean sleep
7
8 sleep: sleepOf.c
9     mpic++ $(FLAGS) ${DMRFLAGS} ${SLURMFLAGS} sleepOf.c -o sleepOf
10
11 clean:
12     rm -f *.out *.o core.* *.err *.so tmp hostfile.txt sleepOf
```

Figure 3.12 Makefile for linking the trace library to the test application.

By executing the testing application, a CSV trace file is generated. The obtained file is used to evaluate the right behaviour of the trace generation library and to validate that the fields meet the fields defined in the requirements section.

```

75 gs02r1b62,224,0,3054107,1,2024-07-31-22-12-54,1722456774521,9,ReceiveStart,2,4,101
76 gs02r1b65,224,0,2482259,1,2024-07-31-22-12-54,1722456774524,9,ReceiveStart,3,4,101
77 gs02r1b69,224,0,1874604,1,2024-07-31-22-12-54,1722456774544,6,SendEnd,4,8,100
78 gs02r1b71,224,0,896332,1,2024-07-31-22-12-54,1722456774545,6,SendEnd,6,8,100
79 gs02r1b71,224,0,896332,1,2024-07-31-22-12-54,1722456774550,13,DetachingStart,6,8,100
80 gs02r1b69,224,0,1874604,1,2024-07-31-22-12-54,1722456774549,13,DetachingStart,4,8,100
81 gs02r1b70,224,0,3983637,1,2024-07-31-22-12-54,1722456774553,6,SendEnd,5,8,100
82 gs02r1b70,224,0,3983637,1,2024-07-31-22-12-54,1722456774555,13,DetachingStart,5,8,100
83 gs02r1b62,224,0,3054107,1,2024-07-31-22-12-54,1722456774555,10,ReceiveEnd,2,4,101
84 gs02r1b65,224,0,2482259,1,2024-07-31-22-12-54,1722456774574,10,ReceiveEnd,3,4,101
85 gs02r1b72,224,0,263595,1,2024-07-31-22-12-54,1722456774572,6,SendEnd,7,8,100

```

Figure 3.13 A portion of the trace file of the example application.

For evaluating the right generation of the trace files, it is necessary to define some validation rules. The selected rules are listed in Table 4.

Validation rules
Trace fields must meet the definition
Trace timestamps must be incremental
All iterations must be traced, and the records must be incremental
All ranks in execution must be traced
For each event of type START must be an event of type END
Iteration events must precede Reconfiguration events in the same iteration
The change of the number of ranks must occur after Reconfiguration events
Events Init and Finalize only will be in the Rank 0

Table 4. Trace library validation rules.

For evaluating the generated trace file, the Excel Viewer application has been used, it is a VSCode plugin for editing Excel spreadsheets.

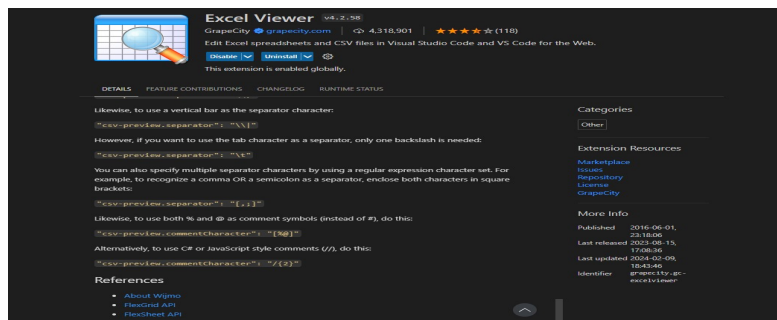


Figure 3.14 Excel Viewer, VSCode plugin.

- Excel Viewer (<https://marketplace.visualstudio.com/items?itemName=GrapeCity.gc-excelviewer>)

For a detailed analysis of the trace file, Excel Viewer allows to load and apply multiple filters to the file. Via those filters, the application can show, for instance, only some types of traces, traces for selected ranks, iteration, processors, etc. Figure 3-15 shows several examples of filters.

Gs02r1b54	224	0	989847	1	2024-07-31-22-12-53	1722456773699	0	Setup	0	8	0
gs02r1b54	224	0	989847	1	2024-07-31-22-12-53	1722456773696	7	Ascending	Descending		0
gs02r1b65	224	0	2482248	1	2024-07-31-22-12-53	1722456773734	7	Filter by Condition Filter by Value			0
gs02r1b55	224	0	2342602	1	2024-07-31-22-12-53	1722456773739	7	Search			0
gs02r1b70	224	0	3983637	1	2024-07-31-22-12-53	1722456773737	7	Select All			0
gs02r1b71	224	0	896332	1	2024-07-31-22-12-53	1722456773738	7	InitEnd			0
gs02r1b62	224	0	3054098	1	2024-07-31-22-12-53	1722456773736	7	InitStart			0
gs02r1b69	224	0	1874604	1	2024-07-31-22-12-53	1722456773750	7	IterationEnd			100
gs02r1b72	224	0	263595	1	2024-07-31-22-12-53	1722456773756	7	IterationStart			100
gs02r1b72	224	0	263595	1	2024-07-31-22-12-54	1722456774042	8	ReceiveEnd			100
gs02r1b65	224	0	2482248	1	2024-07-31-22-12-54	1722456774042	8	Apply		Cancel	Clear
gs02r1b54	224	0	989847	1	2024-07-31-22-12-54	1722456774042	8				100
gs02r1b69	224	0	1874604	1	2024-07-31-22-12-54	1722456774042	8				100
gs02r1b62	224	0	3054098	1	2024-07-31-22-12-54	1722456774042	8				100
gs02r1b71	224	0	896332	1	2024-07-31-22-12-54	1722456774042	8				100

Figure 3.15 Excel Viewer filters for Events visualization.

Filters can be used to analyze traces, to check the validation rules. Figure 3-16 shows the first Rank 0 events to validate that it follows the right sequence of events.

Gs02r1b54	224	0	989847	1	2024-07-31-22-12-53	1722456773693	0	Setup	0	8	0
gs02r1b54	224	0	989847	1	2024-07-31-22-12-53	1722456773694	1	InitStart	0	8	0
gs02r1b54	224	0	989847	1	2024-07-31-22-12-53	1722456773696	2	InitEnd	0	8	0
gs02r1b54	224	0	989847	1	2024-07-31-22-12-53	1722456773696	7	IterationStart	0	8	0
gs02r1b54	224	0	989847	1	2024-07-31-22-12-54	1722456774042	8	IterationEnd	0	8	100
gs02r1b54	224	0	989847	1	2024-07-31-22-12-54	1722456774046	3	ReconfigureStart	0	8	100
gs02r1b54	224	0	989847	1	2024-07-31-22-12-54	1722456774422	4	ReconfigureEnd	0	8	100
gs02r1b54	224	0	989847	1	2024-07-31-22-12-54	1722456774436	7	IterationStart	0	8	100
gs02r1b54	224	0	989847	1	2024-07-31-22-12-54	1722456774436	5	SendStart	0	8	100
gs02r1b54	224	0	989880	1	2024-07-31-22-12-54	1722456774521	9	ReceiveStart	0	4	101
gs02r1b54	224	0	989847	1	2024-07-31-22-12-55	1722456775084	6	SendEnd	0	8	100
gs02r1b54	224	0	989880	1	2024-07-31-22-12-55	1722456775094	10	ReceiveEnd	0	4	101
gs02r1b54	224	0	989847	1	2024-07-31-22-12-55	1722456775096	13	DetachingStart	0	8	100
gs02r1b54	224	0	989847	1	2024-07-31-22-12-57	1722456777131	14	DetachingEnd	0	8	100

Figure 3.16 Example of events in the first iterations of the Rank0.

To validate the functionality of the trace library multiple analyses of the trace files were done. During the validation process, some issues were detected, and some improvements were applied. Several iterations were needed to get the right trace file format.

As the trace file is in a standard text format, useful information can be extracted by using other applications. Office applications such as Excel, LibreOffice Calc, or Google Sheets can also be used.

A plotting Python notebook has been developed to validate the coherence of the sequences of events in the trace files. It has been developed using Python and Jupyter notebook plugins for VSCode.

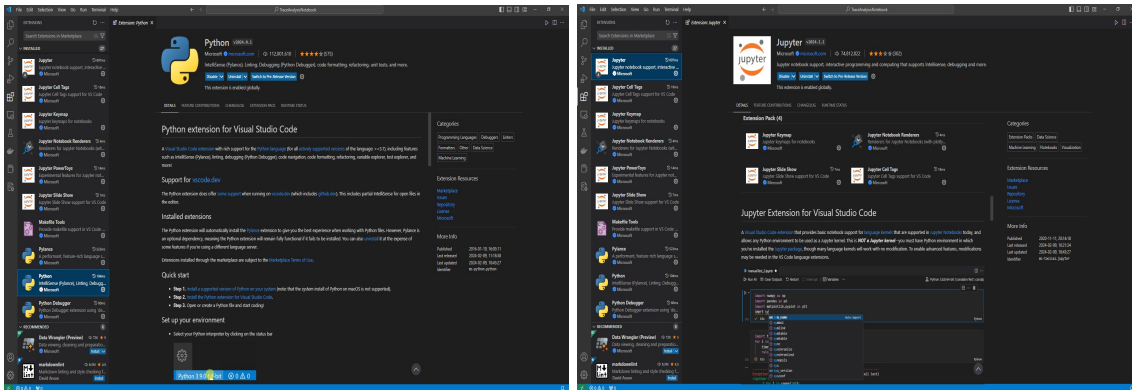


Figure 3.17 Python and Jupyter notebook plugins for VSCode.

Information and downloads for those plugins can be found in the following links:

- Python (<https://marketplace.visualstudio.com/items?itemName=ms-python.python>)
- Jupyter (<https://marketplace.visualstudio.com/items?itemName=ms-toolsai.jupyter>)

The Jupyter Notebook can load the CSV trace file as well as filter and plot it by using Pandas, Matplotlib, and PyQ5, Python libraries. More information and downloads of those libraries can be found in the following links:

- Pandas (<https://pandas.pydata.org/>)
- Matplotlib (<https://matplotlib.org/>)
- PyQ5 (<https://pypi.org/project/PyQt5/>)

The notebook loads the CSV file and extracts information about the events of each rank executed in each iteration. The notebook can plot the sequence of events in the same or different plots, to get a visual representation of the event sequence. The visualization method used by Matplotlib and PyQT5 allows the application of zoom and scroll to the created plots, to visualize trace sequence details.

The notebook creates several plots. The first plot is a general plot where all events and all ranks are present. It also includes a plot of resource evolution and the reconfiguration event of Rank 0, which is the most relevant event regarding resource allocation. An example is shown in Figure 3-18.

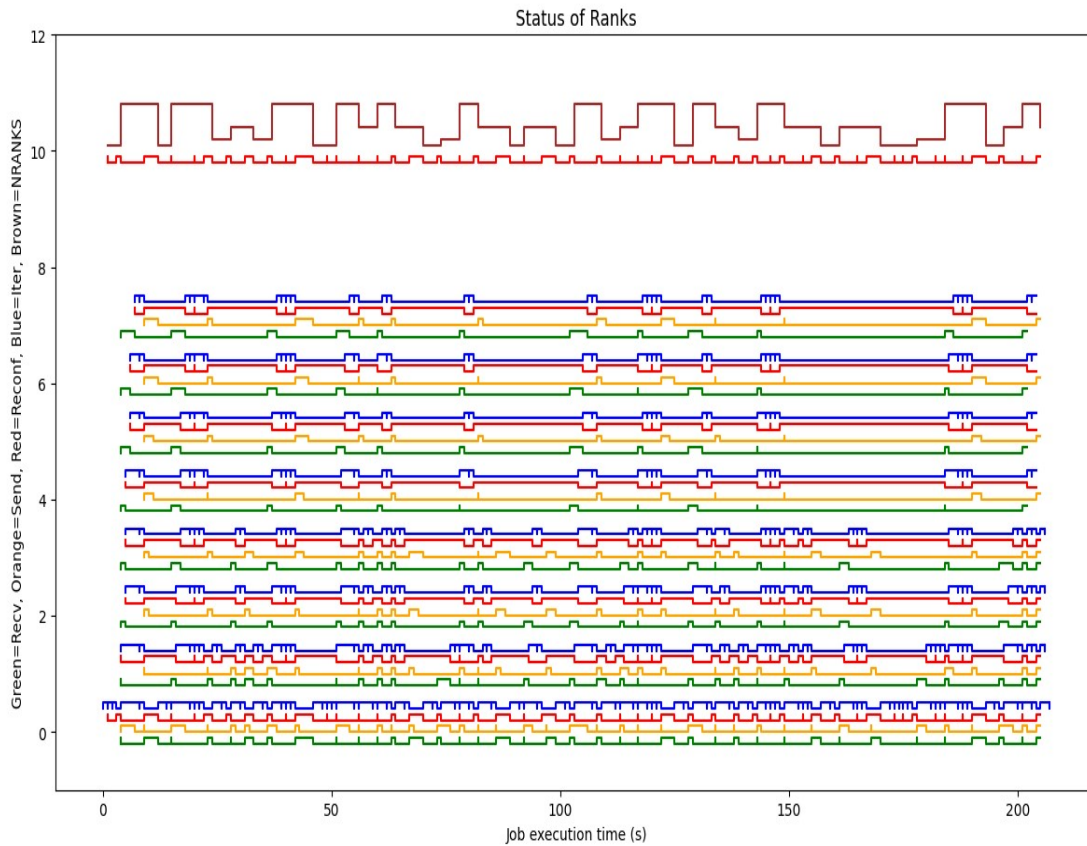


Figure 3.18 Plotting of all events of all ranks and resources allocated.

Using zoom, specific details of the execution can be reviewed, for instance relation between the reconfiguration event and the allocated resources, and the relation between different events in the rank 0.

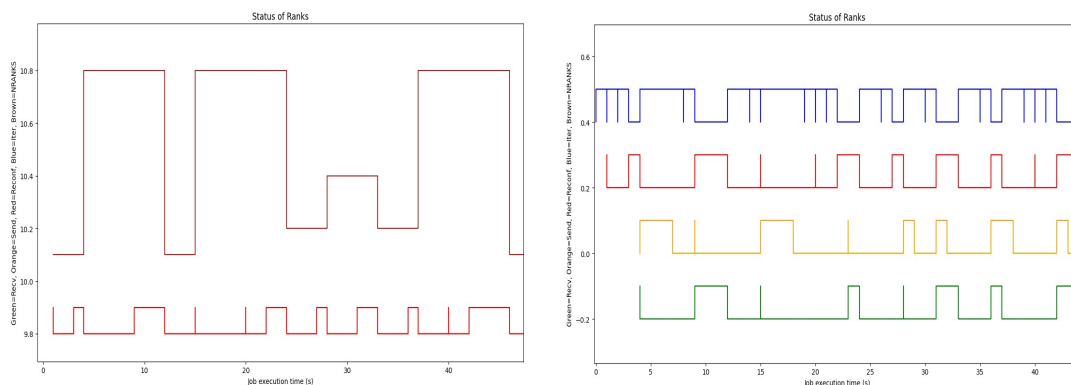


Figure 3.19 Resource allocation and firsts Rank 0 events.

From those plots, information useful to validate the traces can be obtained. For instance, to validate some sequences:

- The change in the allocation of resources is synchronized with the final of the reconfiguration.
- The iteration and the reconfiguration events are alternated.
- The send and receive events are simultaneous but in different tasks.

Another plot obtained executing the notebook compares the send events of different ranks. This plot shows differences in the data transmission times between nodes.

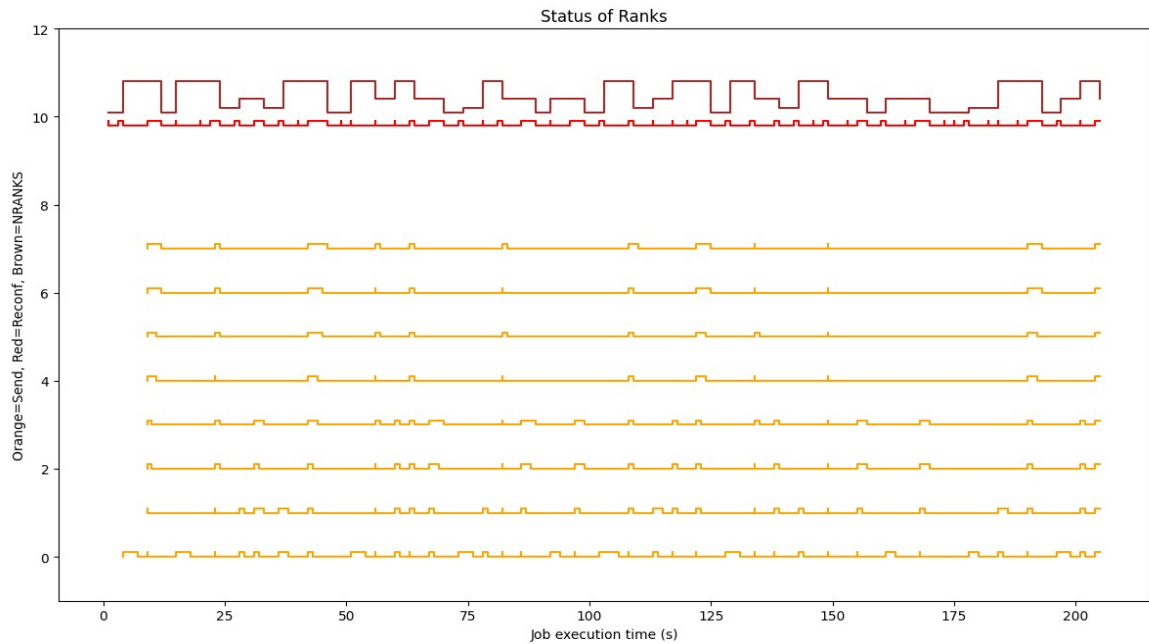


Figure 3.20 Plot of send events of different ranks.

Another interesting plot is the comparison between the receive events of different ranks, it can be useful to find differences between data transmission and times between nodes.

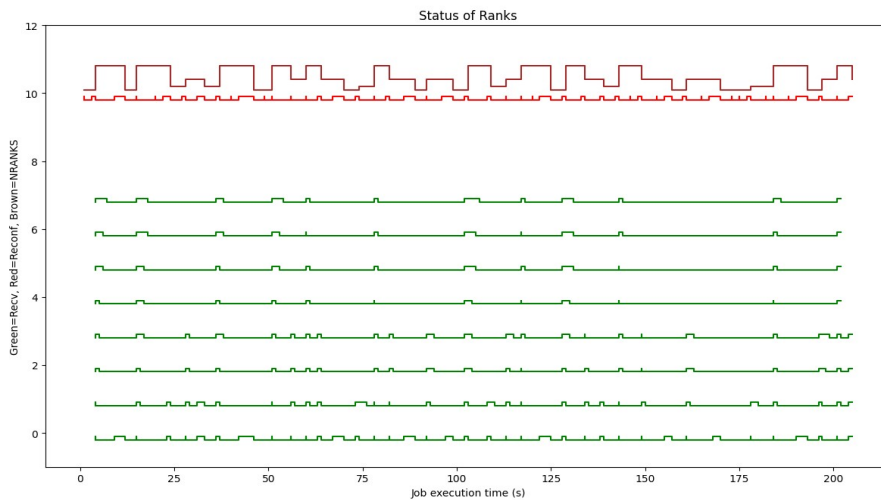


Figure 3.21 Plot of receive events of different ranks.

By using the plots, information about several ranks can be compared. For instance, the send and receive communication times.

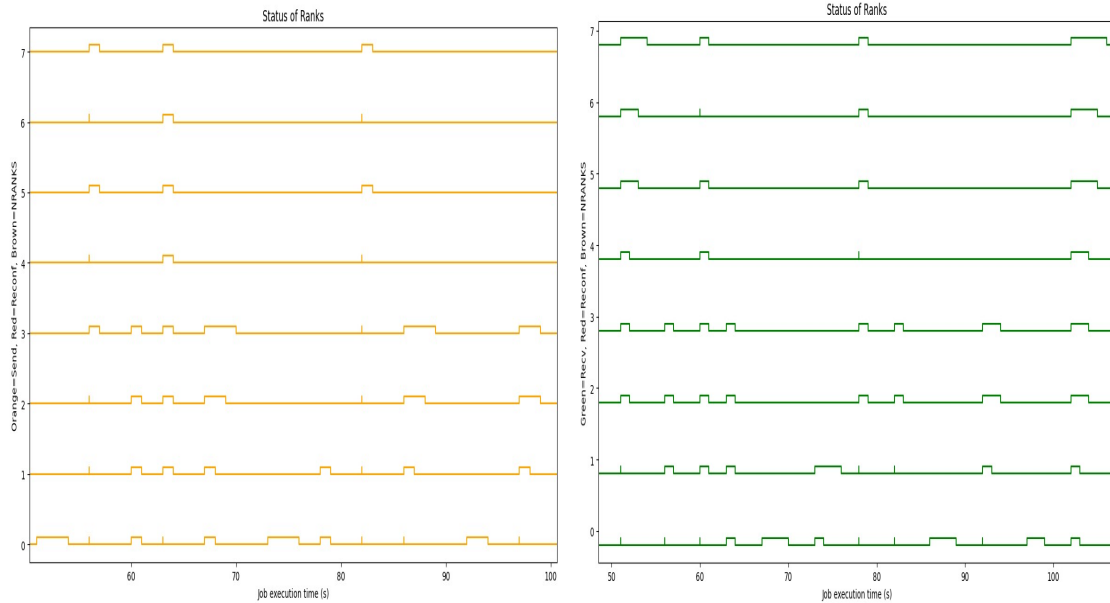


Figure 3.22 Detailed plot of the send and receive events.

The test and validation process of the trace files can be summarized in the following steps:

- Definition of validation rules.
- Execution of a test application to generate trace files.
- Trace file analysis via a CSV file visualization application.
- Events timelines plotting and analysis.

It is important to highlight that data analysis has been a tool for fine-tuning the trace generation application development. The data test and validation procedures have been executed several times over different versions of the library until the first version of the trace library has been released.

3.12. License Agreement

One important point is the selection of an appropriate license agreement for the trace generation library.

One of the development requirements of the library is to be developed as an open-source application. This is the case for most of the applications developed by researchers at BSC.

There are multiple license agreements for open-source projects, the GNU's General Public License v3 (GPLv3) agreement is one of the most common agreements used and this is the selected to be applied to the development of the trace library.

To avoid future misunderstandings, the license agreement is included in the header of each source code file in the way that is shown in Figure 3.23.

```
1  /*
2  * This program is free software: you can redistribute it and/or modify
3  * it under the terms of the GNU General Public License as published by
4  * the Free Software Foundation, either version 3 of the License, or
5  * (at your option) any later version.
6  *
7  * This program is distributed in the hope that it will be useful,
8  * but WITHOUT ANY WARRANTY; without even the implied warranty of
9  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
10 * GNU General Public License for more details.
11 *
12 * You should have received a copy of the GNU General Public License
13 * along with this program. If not, see <https://www.gnu.org/licenses/>.
14 */
```

Figure 3.23 License agreement in the headers of the source code files.

More information about the GPLv3 license agreement is available at the following link:

- GPLv3 (<https://www.gnu.org/licenses/gpl-3.0.html>)

3.13. Summary of the trace generation library section

This section started defining the trace library requirements, the right development tools, and the codification guidelines.

Afterwards, it has been explained how the trace library has been developed, some detailed information about its source code, and how it has been integrated inside the source code of the DMR library.

Finally, it has been explained how the library has been tested via executing and tracing a simple malleable parallel application, and how the final validation has been done, analyzing in detail the traces recorded by using CSV analysis tools and data science tools based on Python.

4. Trace format converter for Paraver

Paraver is the performance analysis tool of the BSC. This tool has been developed for several years by the BSC's team and is the standard tool for parallel applications performance analysis.

This section is focused on the development of a software utility for converting trace files from CSV format to PRV format, the Paraver trace format. The section contains the application requirements, the Paraver trace format, the process of application development, and the process of testing and validation of the Paraver trace files.

4.1. Format converter utility requirements

This section defines the list of requirements that the conversion utility must fulfil. The features required are, basically, the generation of PRV files from CSV files, and the possibility of merging several CSV files into a unique PRV file.

Besides the basic functional requirements mentioned above, there are also other important requirements to consider regarding functionalities, codification, and application development standards.

Table 5 shows the trace format converter requirements.

Requirements
Convert CSV DMRTRACE trace files to PRV Paraver trace files
Merge several CSV files in a unique PRV file
The PRV file must be visualized in Paraver by using easily identifiable colors
The PRV file must respect the resource hierarchy of Paraver
Detection of errors in the source CSV trace files at conversion time
PRV files must be generated in the folder where the application is executed
The development language must be C/C++
The utility must be multiplatform, compact, and without dependencies
The utility must be an opensource application
The codification must be according to the trace generation library
Codification and commentaries must be in English
The source code must be hosted in the BSC's GitLab repository
Provide documentation in English

Table 5.Trace converter application requirements.

4.2. Paraver trace format

This section is a summary of the Paraver trace format definition. All the information exposed in the section is extracted from the Paraver trace manual [3], which can be found at the following link:

- Paraver trace format (<https://tools.bsc.es/doc/1370.pdf>)

It is not the intention of this section to transcribe the Paraver trace manual, but just to summarize the key points. For this reason, some images of parts of the Paraver manual have been reproduced.

The Paraver trace files are composed of two parts. The first part is the header of the file, which contains a readable text line with the description of the application execution, and the hierarchy of the resources involved in the execution. The second part is the trace body, which contains a list of readable text lines with information corresponding to states, events and communications generated during the execution of the application.

Figure 4-1 shows a small portion of a trace file.

```
#Paraver (22/05/01 at 16:20):1021312:2(16,16):1:2(1:1,1:2)
1:1:1:1:1:0:100:4
1:2:1:2:1:0:200:4
1:1:1:1:1:100:300:1
1:1:1:1:1:200:500:4
3:1:1:1:1:300:325:2:1:2:1:200:330:10:3000
2:1:1:1:1:300:60000000:1
```

Figure 4.1 Example of Paraver trace file format.

The detailed header description is shown in Figure 4-2.

3.1.1 Paraver trace header

The trace header defines the process and the resource model of the tracefile. It contains the information about the number of applications of the tracefile, the number of tasks for each application and the number of threads in each task. Furthermore, the header defines the number of nodes and its number of processors.

The trace header is a line where the different fields define the object structure (fields are separated by colons). The trace header format is :

- **#Paraver (dd/mm/yy at hh:mm)**: defines the date and hour where trace has been generated. Is important to use the symbol # at the beginning of the header line because it indicates that it is in ASCII trace format.
- **ftime**: total trace time in microseconds
- **nNodes(nCpus1[,nCpus2,...,nCpusN])**: defines the number of nodes and number processors per node. After the number of nodes (**nNodes**), the list of the number of processors must be specified (**nCpus1** is the number of processors on node 1, **nCpus2** is the number of processors on node 2,...).
- **nAppI**: number of applications in the trace file.
- **applicationList** : The application list defines the application object structure. Each application has its application list (**applicationList**) separated by a colon. The application list format is:

```
nTasks(nThreads1:node,...,nThreadsN:node)
```

Figure 4.2 Paraver trace file header description.

The Paraver trace body can contain three types of information.

- States (1).
- Events (2).
- Communications (3).

Each type has a different numeric identificatory and a different recording format in the body of the trace file.

Figure 4-3 shows the State record description and format.

State record

State records represent intervals of actual thread status. The first field is the record type identifier (for state records, type is 1). The next fields identify the resource (`cpu_id` field) and the object to which the record belongs (`appl_id`, `task_id` and `thread_id` fields). Remember that `cpu_id` is the global processor identifier (if no resource levels have been defined the processor identifier must ever be a zero). Beginning time and ending time of the state record also have to be specified, and finally, the state field is an integer that encodes the activity carried out by the thread.

```
1:cpu_id:appl_id:task_id:thread_id:begin_time:end_time:state
```

If state is not assigned to any processor, its `cpu_id` should be set to zero. For example, in next state records :

```
#Paraver(23/02/01 at 18:57):500:1(2):1:1(1:1)
1:2:1:1:1:0:200:1
1:0:1:1:1:200:300:1
1:1:1:1:1:300:500:1
```

Figure 4.3 State record description and format.

The Event record description and format are shown in Figure 4-4.

Event record

Event records represent punctual events. These user events are often used to mark the entry and exit of routines or code blocks, hardware counter reads, ... They can also be used to flag changes in variable values. Event records include a type and a value.

```
2:cpu_id:appl_id:task_id:thread_id:time:event_type:event_value
```

The first field is the record type identifier (for event records, type is 2). The next fields identify the resource (`cpu_id` field) and the object where record belongs (`appl_id`, `task_id` and `thread_id` fields). Then, the current absolute time of the event occurrence is specified. Event type and event value are also specified. The type identifies the event and the value is used to distinguish events of the same type.

If a zero value is assigned to the `cpu_id`, the event will not be displayed in resource model views (`CPU`, `NODE` and `SYSTEM`). For example, in next trace:

```
#Paraver(23/02/01 at 18:57):500:1(2):1:1(1:1)
1:2:1:1:1:0:100:1
2:2:1:1:1:100:5000:1
1:2:1:1:1:100:200:1
1:0:1:1:1:200:300:1
1:1:1:1:1:300:400:1
2:0:1:1:1:400:5000:0
1:1:1:1:1:400:500:1
```

Figure 4.4 Event record description and format.

The Communication record is not relevant for the format converter, for this reason, it is not going to be commented in this section.

It is also needed to take into consideration several important rules for the right trace file generation.

- Ascending order of time, for states or events with the same timestamp.
- Descending order for records of different types with the same timestamp, in the order, communications, events, and states.

Finally, it is relevant to understand the object structure due to it will be a key point at the time of generating the header of the trace file.

Figure 4-5 shows the object structure.

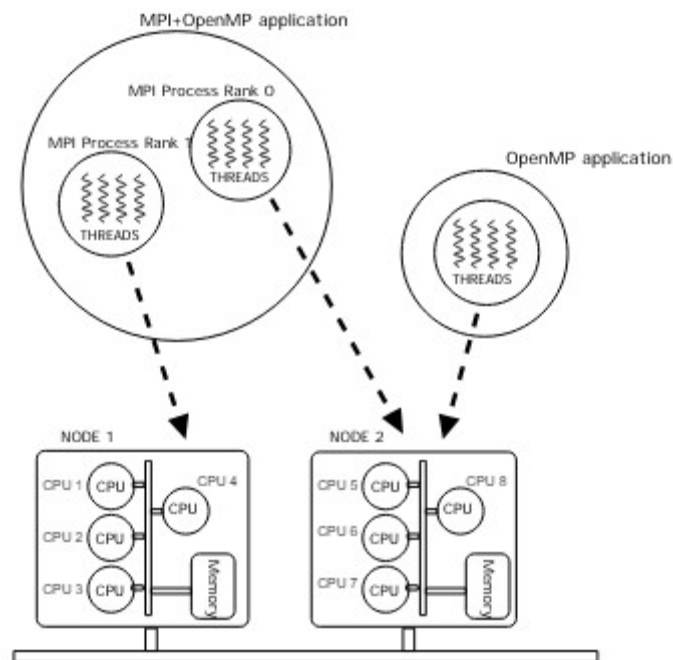


Figure 4.5 Paraver object structure.

It is important to highlight that the nodes and cpus are enumerated. It means that the nodes have incremental numbers, each different node has a new node ID and each cpu in the same or different node follows the same rule.

4.3. Strategy for DMRTRACE event conversion to Paraver format

Paraver trace files can contain states, events, and communications. So, the first design decision to take is to decide if the DMRTRACE events must be converted to Paraver events or if they must be converted to Paraver states.

In the beginning seems more logical to convert DMRTRACE events to Paraver events due to it is a more direct conversion. Converting DMRTRACE events to Paraver states is quite more complex because it requires defining the states and converting the events to the starting and ending points of those states. However, Paraver represents graphically the states in a very clear way while the direct representation of the events is not so useful.

Figure 4-6 shows the direct event representation in Paraver. The event representation of Paraver consists of a set of green flags without relation.

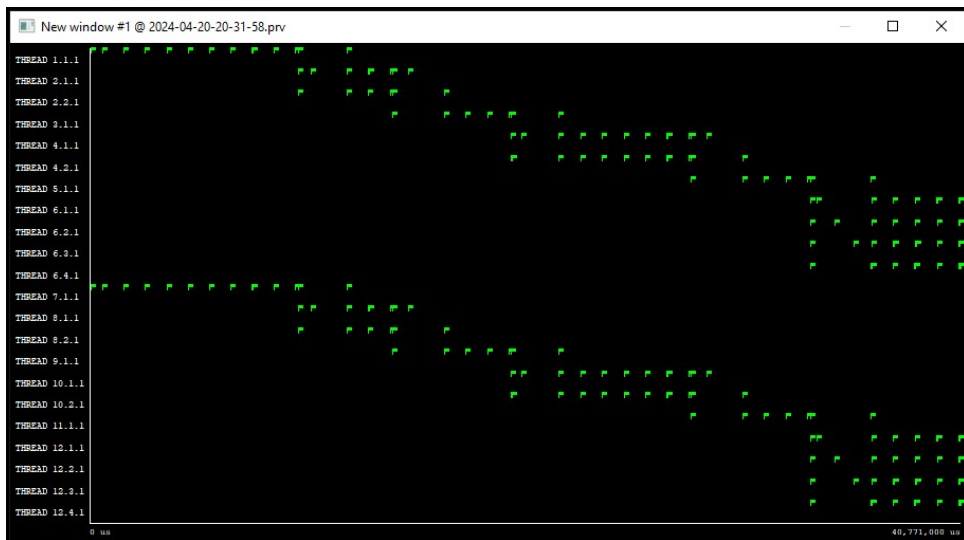


Figure 4.6 Direct representation of events in Paraver.

Figure 4-7 shows the direct state representation in Paraver. The state representation of Paraver consists of a set of bars with and starting and ending point. Each different color corresponds to a different state.

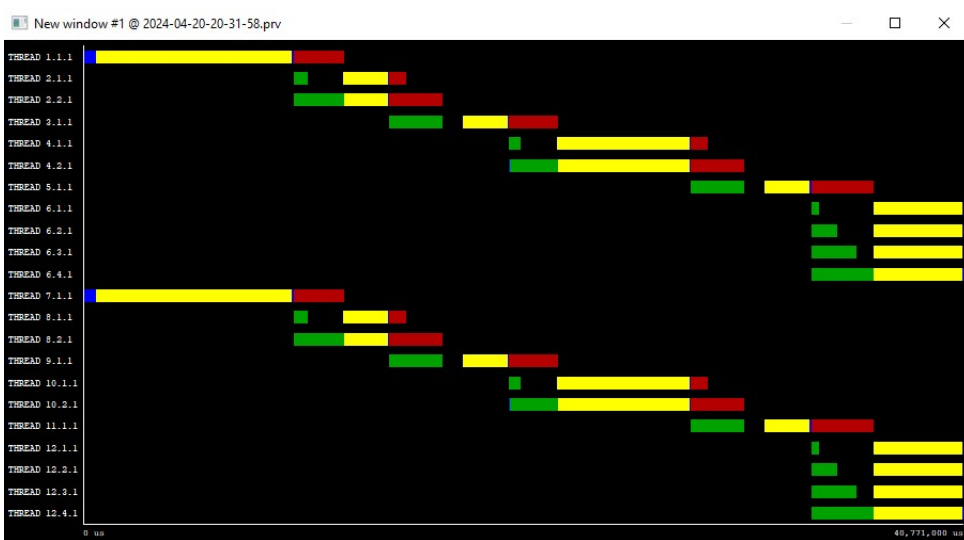


Figure 4.7 Direct representation of states in Paraver.

Paraver has several options that allow users to create semantics from non-semantic trace files and one of those options allows to create states from events. Using this option DMRTRACE events can be converted to Paraver trace events, and after, create Paraver states with a clear visualization and data analysis.

Figure 4-8 shows how to create Paraver states from events. As it is shown in the image, the conversion is done via the semantics properties of the window, by selecting in the thread field the value “Last Evt Val”.

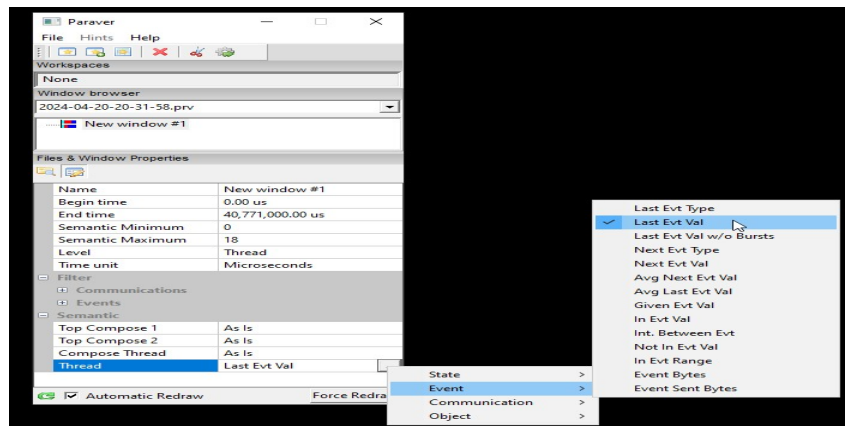


Figure 4.8 Paraver window property for creating states from events.

Applying that event to state conversion rule the result is a composition of states and events, such as is shown in Figure 4-9.

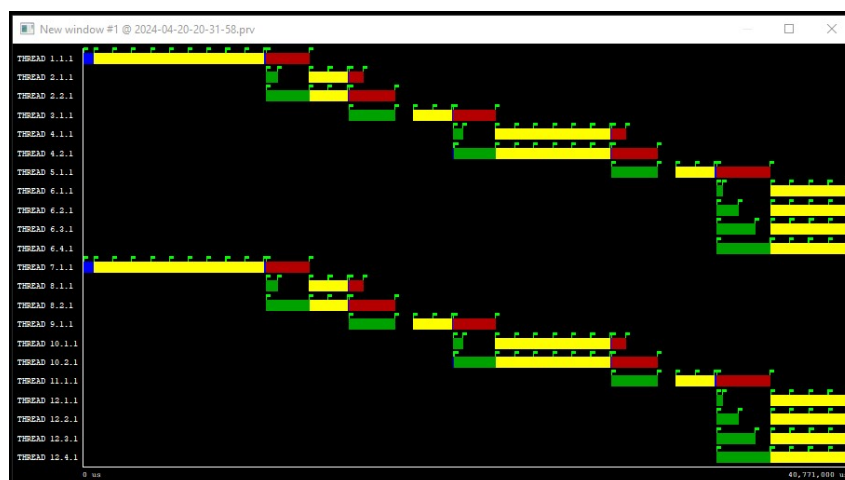


Figure 4.9 Representation of events converted to states in Paraver.

It can be concluded that converting DMRTRACE events to Paraver trace events and converting the events to states by Paraver conversion rules at the analysis time is the best strategy.

4.4. Trace conversion application design

The application design is strongly conditioned by the requirements of being a compact application, free of dependencies, and multiplatform. Those requirements cannot be fulfilled by using a very high-level programming language such as Python which has powerful libraries for extracting and manipulating data from files, such as Pandas. The use of Python and Pandas is not possible at this moment in MareNostrum 5 and even Python has some extensions to create standalone dependency-free applications, the results are big executable applications with large loading times and low performance.

To meet the compact, portable, free of dependency, small, and high-performance applications there are diverse programming languages available, but to keep a unique programming language and development tools for all the deliverable applications of this project, the language selected has been C/C++.

As the programming language, the programming tools, and the codification guidelines will be the same that were selected for the trace recording library development, it is not necessary to explain another time the tools selection procedure explained in sections 3.5, 3.6, and 3.7 of this document.

There are several very good C/C++ libraries for extracting data from files and converting that data to different formats. At the time of designing the application has been evaluated several ones:

- SQLite (<https://sqlite.org/>)
- DuckDB (<https://duckdb.org/>)
- DataFrame (<https://github.com/hosseimoein/DataFrame>)

SQLite and DuckDB are embedded relational databases with similar functionalities. The way of working with them in the development of the trace conversion application would be quite similar:

- Loading CSV trace files and converting them into database tables.
- Using SQL sentences to get the data needed to create the PRV file.

Even though those libraries are easy to use with powerful features, it was decided not to use them due to the growth of the final application is not justified by the benefit of using them. So, it is preferable to dedicate more time to the application development and obtaining a more compact application.

DataFrame follows a different approach, it works with frames like Pandas. This library requires a high learning curve. So, it is preferable to dedicate more time to the application development to obtaining a compact application.

After evaluating the pros and cons of using C/C++ libraries to ease the data management in the application, it has been decided not to use an external library, and to develop the source code from scratch.

The following application structure has been decided:

- Define a struct with the fields of the CSV trace files.
- Load each CSV file in a vector of structs.
- Generate the PRV body converting each item in the vector to PRV format.
- Recording relevant data for the PRV header during the CSV loading and format conversion.
- Generate the PRV header with the data recorded.
- Save the header and the body in a plain text PRV file.

Based on this design the most complex part of the application is converting each CSV trace line into a PRV trace line and recording the right data during the process to be able to generate the PRV file header.

4.5. Data conversion rules definition

It is necessary to define how the data for the PRV file, header, and body, will be generated from the data recorded in the CSV trace files. The definition of the conversion rules is part of the design of the format conversion application.

The trace fields defined in previous sections have been used.

#Paraver (header). This is a constant text at the beginning of the Paraver files.

Date and time (header). The conversion application gets the date and time of the system at the time of executing the converter.

Total trace time (header). The conversion application gets the lower and higher timestamps of all the trace files in merging and calculates the total trace time as the difference between those timestamps. The timestamp is converted to microseconds due to it is the default timescale of Paraver.

Number of nodes and cpus per node (header). The conversion application enumerates the nodes present in all the trace files and assigns one cpu per node. The reason to do this assignment is that, even though each node has multiple cpus, at this moment, DMR uses only one cpu.

Number of applications (header). It is the number of trace files to merge to create the Paraver trace file.

Number of tasks and threads for application (header). The conversion application creates a list of ranks for each application (file). The list contains each rank present in the trace file and assigns to each rank a "1" thread of the total number of iterations recorded in the trace file, it depends on the conversion option selected when the application is executed.

Cpu (Body). Even the right way to fill this field would be to enumerate the cpus of the application, the conversion application is forced to use the rank number in this field to get the right visualization in Paraver.

Application (body). The conversion application enumerates the trace files to merge and uses those values in the application field.

Task (body). The conversion application uses the rank of the trace record.

Thread (body). The conversion application uses the iteration of the trace record of uses the value "1" depending on the execution parameters.

Timestamp (body). The conversion application gets the timestamp of the trace record and subtracts the lower timestamp of all the trace files.

Event code (body). The conversion application uses the event of the trace record but assigns the same event code to the same types of events independently if they are of type START or END. The value assigned is always the value of the equivalent event of type END.

Event value (body). The conversion application assigns to the events of the type start a hardcoded value to get a nice visualization color in Paraver and assigns the value 0 to all the events of type END. With this assignment, Paraver can convert events to states and show a clear timeline visualization.

The following example shows the header of the conversion of a unique trace file. It uses 8 nodes of 1 cpu and 1 application executing a maximum of 8 ranks during 17 iterations:

```
#Paraver (2024/06/21 at 08:19):576862000:8(1,1,1,1,1,1,1,1):1:8(17:1,17:2,17:3,17:4,17:5,17:6,17:7,17:8)
```

The following example shows several converted trace records. The first value (2) indicates that it is a record of type event, the second value (1) is the rank of the record, the third value (1) is the number of the trace file used to create the Paraver trace file, the fourth value (1) is also the rank, the fifth value is the iteration of the record, the sixth value is the incremental timestamp, the seventh value is the event of the record, and the last value indicates to Paraver if it is a record of an event of type START (value different of zero) or an event of type END (value equal to zero):

```
2:1:1:1:4:132918000:4:1  
2:1:1:1:4:133289000:4:0
```

4.6. Trace conversion application development

The application has been developed in C/C++ following the application design explained in the application design section, and the development tools and codification rules have been the same ones used for developing the trace generation library.

As was defined in the requirements section, the source code can be compiled on several platforms. A Makefile file is provided to compile the code in Linux and Windows to create a standalone small executable file.

```
1 # Compiler and compiler flags  
2 CXX = g++  
3 CXXFLAGS = -std=c++11 -Wall  
4  
5 # Source file and executable name  
6 SRC = dmtraceparser.cpp  
7 TARGET = dmtraceparser  
8  
9 .PHONY: all linux windows clean  
10  
11 # Default target  
12 all: $(TARGET)  
13  
14 # Linux compilation  
15 linux:  
16     $(CXX) $(CXXFLAGS) $(SRC) -o $(TARGET)  
17  
18 # Windows compilation  
19 windows:  
20     $(CXX) $(CXXFLAGS) $(SRC) -o $(TARGET).exe  
21  
22 # Clean  
23 clean:  
24     rm -f $(TARGET) $(TARGET).exe
```

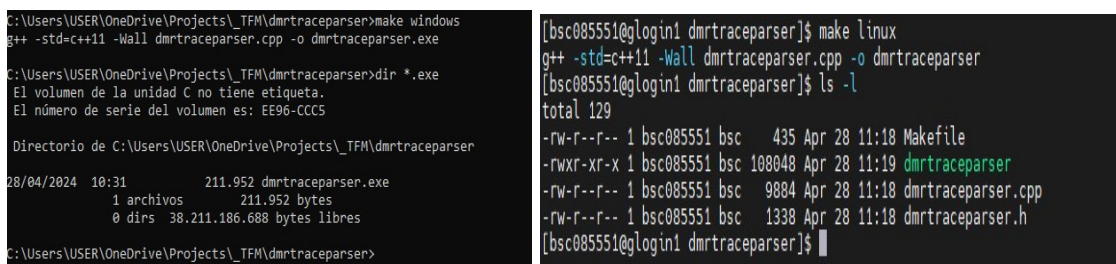
Figure 4.10 Makefile for compiling the trace conversion application.

The Makefile can create Windows and Linux executables by using make, in the following way:

- make windows.
- make linux.

It creates the executable files “dmrtraceparser.exe” on Windows and “dmrtraceparser” on Linux.

Figure 4-11 shows how to compile the source code of the application to create an executable on Windows and also on Linux.



```
C:\Users\USER\OneDrive\Projects\_TFM\dmrtraceparser>make windows
g++ -std=c++11 -Wall dmrtraceparser.cpp -o dmrtraceparser.exe

C:\Users\USER\OneDrive\Projects\_TFM\dmrtraceparser>dir *.exe
El volumen de la unidad C no tiene etiqueta.
El número de serie del volumen es: EE96-CCC5

Directorio de C:\Users\USER\OneDrive\Projects\_TFM\dmrtraceparser
28/04/2024 10:31          211.952 dmrtraceparser.exe
                1 archivos          211.952 bytes
                0 dirs 38.211.186.688 bytes libres

C:\Users\USER\OneDrive\Projects\_TFM\dmrtraceparser>

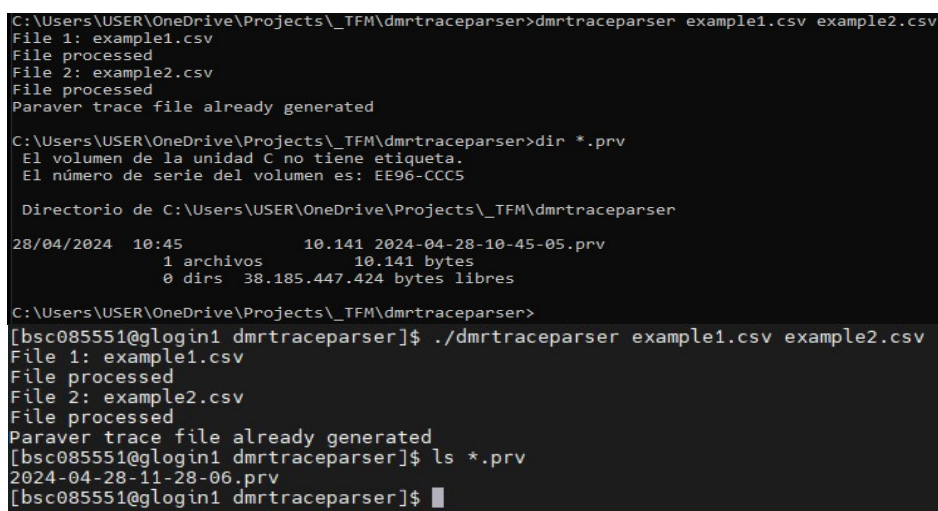
[bsc085551@glogin1 dmrtraceparser]$ make linux
g++ -std=c++11 -Wall dmrtraceparser.cpp -o dmrtraceparser
[bsc085551@glogin1 dmrtraceparser]$ ls -l
total 129
-rw-r--r-- 1 bsc085551 bsc  435 Apr 28 11:18 Makefile
-rwxr-xr-x 1 bsc085551 bsc 108048 Apr 28 11:19 dmrtraceparser
-rw-r--r-- 1 bsc085551 bsc  9884 Apr 28 11:18 dmrtraceparser.cpp
-rw-r--r-- 1 bsc085551 bsc  1338 Apr 28 11:18 dmrtraceparser.h
[bsc085551@glogin1 dmrtraceparser]$
```

Figure 4.11 Trace converter application compilation on Windows.

The Makefile generates and small executable application, with a size of just 211kB on Windows and 108kB on Linux. The executable size meets the defined requirement of having and small, standalone multiplatform application.

To execute the application, it is needed to call the executable file from the operating system console, adding as parameters the CSV files to convert and merge.

Figure 4-12 shows how to execute the application merging and converting two CSV trace files on Windows and Linux.



```
C:\Users\USER\OneDrive\Projects\_TFM\dmrtraceparser>dmrtraceparser example1.csv example2.csv
File 1: example1.csv
File processed
File 2: example2.csv
File processed
Paraver trace file already generated

C:\Users\USER\OneDrive\Projects\_TFM\dmrtraceparser>dir *.prv
El volumen de la unidad C no tiene etiqueta.
El número de serie del volumen es: EE96-CCC5

Directorio de C:\Users\USER\OneDrive\Projects\_TFM\dmrtraceparser
28/04/2024 10:45          10.141 2024-04-28-10-45-05.prv
                1 archivos          10.141 bytes
                0 dirs 38.185.447.424 bytes libres

C:\Users\USER\OneDrive\Projects\_TFM\dmrtraceparser>

[bsc085551@glogin1 dmrtraceparser]$ ./dmrtraceparser example1.csv example2.csv
File 1: example1.csv
File processed
File 2: example2.csv
File processed
Paraver trace file already generated
[bsc085551@glogin1 dmrtraceparser]$ ls *.prv
2024-04-28-11-28-06.prv
[bsc085551@glogin1 dmrtraceparser]$
```

Figure 4.12 Execution of the conversion application on Windows and Linux.

The program confirms the processed files and generates a unique PRV file whose name of the date and time of the computer when the file is created.

The converter can detect errors in the CSV source files at the time of format conversion. It is easy to test this feature by using a wrong CSV file as an input.

```
C:\Users\USER\OneDrive\Projects_TFM\dmrtraceparser>dmrtraceparser example1.csv example_wrong.csv
File 1: example1.csv
File processed
File 2: example_wrong.csv
Error in line 3: stoi
Invalid trace file
C:\Users\USER\OneDrive\Projects_TFM\dmrtraceparser>
```

Figure 4.13 Error detection in a wrong CSV input file.

As it is shown, an error has been detected in line 3 of the trace file

4.7. Trace conversion application test and validation

This section explains the process of validation of the format conversion application. After the design and development of the application, it is needed to validate that the application meets the defined requirements.

The key features and more difficult ones to validate are to perform the right format conversion and perform the right merge of several CSV trace files in a unique PRV trace file.

The test and validation of those features is quite manual and requires a work procedure. The test and validation procedure will have the following steps:

- Executing a traced application to get a CSV trace file.
- Studying the generated trace file. Analyzing the number of nodes, number of reconfigurations, cpus per node, and number of ranks.
- Executing the conversion application by using the CSV file as a source (dmrtraceparser.exe example1.csv).
- Analyzing the header of the PRV output file and verifying that the time, resources, and application fields meet with the data in the CSV file.
- Analyzing the body of the PRV output file and verifying that nodes, applications, tasks, threads, events, and values meet with the data in the CSV file.
- Repeat the previous steps by using two input files for the format converter application (dmrtraceparser.exe example1.csv example2.csv).
- Load the output files by Paraver, convert events to states, and verify that the program sequences meet with the sequences in the CSV trace files.

Executing this validation sequence will be enough to consider that the conversion utility is ready to be used to convert any CSV trace file.

The first step of validation is getting a CSV trace file from the execution of a traced malleable application and to analyze the recorded data. To analyze the data, it has been used the application Excel Viewer. It is a VSCode plugging already explained in the data trace generation development sections.

Figure 4-14 shows the number of nodes used during the execution of the DMR example application. The application has used four nodes.

Gs04r2b04	224	0	2443594	1	2024-06-21-08-00-55	171894965572
		0	2443594	1	2024-06-21-08-00-55	1718949655721
		0	2466006	1	2024-06-21-08-00-55	1718949655721
		0	2575739	1	2024-06-21-08-00-55	1718949655721
		0	2512386	1	2024-06-21-08-00-55	1718949655721
		0	2670456	1	2024-06-21-08-00-55	1718949655721
		0	2614588	1	2024-06-21-08-00-55	1718949655721
		0	2545080	1	2024-06-21-08-00-55	1718949655726
		0	2653772	1	2024-06-21-08-00-55	1718949655729
		0	2614588	1	2024-06-21-08-00-55	1718949655730
		0	2466006	1	2024-06-21-08-00-55	1718949655730
		0	2670456	1	2024-06-21-08-00-55	1718949655730
		0	2575739	1	2024-06-21-08-00-55	1718949655731
		0	2512386	1	2024-06-21-08-00-55	1718949655731
		0	2653772	1	2024-06-21-08-00-55	1718949655738
gs04r2b11	224	0	2545080	1	2024-06-21-08-00-55	1718949655741
gs04r2b04	224	0	2443594	1	2024-06-21-08-00-55	1718949655755

Figure 4.14 Number of nodes in the CSV file for test and validation.

The number of cpus is always the same (224 in the case of MareNostrum 5).

Gs04r2b04	224	0	2443594	1	2024-06-21-08-00-55	171894965572
gs04r2b04	224	0	2443594	1	2024-06-21-08-00-55	1718949655721
gs04r2b09	224	0	2466006	1	2024-06-21-08-00-55	1718949655721
gs04r2b10	224	0	2575739	1	2024-06-21-08-00-55	1718949655721
gs04r2b05	224	0	2512386	1	2024-06-21-08-00-55	1718949655721
gs04r2b08	224	0	2670456	1	2024-06-21-08-00-55	1718949655721
gs04r2b06	224	0	2614588	1	2024-06-21-08-00-55	1718949655721
gs04r2b11	224	0	2545080	1	2024-06-21-08-00-55	1718949655726
gs04r2b07	224	0	2653772	1	2024-06-21-08-00-55	1718949655729
gs04r2b06	224	0	2614588	1	2024-06-21-08-00-55	1718949655730
gs04r2b09	224	0	2466006	1	2024-06-21-08-00-55	1718949655730
gs04r2b08	224	0	2670456	1	2024-06-21-08-00-55	1718949655730
gs04r2b10	224	0	2575739	1	2024-06-21-08-00-55	1718949655731
gs04r2b05	224	0	2512386	1	2024-06-21-08-00-55	1718949655731
gs04r2b07	224	0	2653772	1	2024-06-21-08-00-55	1718949655738
gs04r2b11	224	0	2545080	1	2024-06-21-08-00-55	1718949655741
gs04r2b04	224	0	2443594	1	2024-06-21-08-00-55	1718949655755

Figure 4.15 The number of processors is always the same for all nodes.

Each rank is executed in a different node always in the cpu 0 of the nodes.

Gs04r2b04	224	0	2443594	1	2024-06-21-08-00-55	171894965572
gs04r2b04	224	0	2443594	1	2024-06-21-08-00-55	1718949655721
gs04r2b09	224	0	2466006	1	2024-06-21-08-00-55	1718949655721
gs04r2b10	224	0	2575739	1	2024-06-21-08-00-55	1718949655721
gs04r2b05	224	0	2512386	1	2024-06-21-08-00-55	1718949655721
gs04r2b08	224	0	2670456	1	2024-06-21-08-00-55	1718949655721
gs04r2b06	224	0	2614588	1	2024-06-21-08-00-55	1718949655721
gs04r2b11	224	0	2545080	1	2024-06-21-08-00-55	1718949655726
gs04r2b07	224	0	2653772	1	2024-06-21-08-00-55	1718949655729
gs04r2b06	224	0	2614588	1	2024-06-21-08-00-55	1718949655730
gs04r2b09	224	0	2466006	1	2024-06-21-08-00-55	1718949655730
gs04r2b08	224	0	2670456	1	2024-06-21-08-00-55	1718949655730
gs04r2b10	224	0	2575739	1	2024-06-21-08-00-55	1718949655731
gs04r2b05	224	0	2512386	1	2024-06-21-08-00-55	1718949655731
gs04r2b07	224	0	2653772	1	2024-06-21-08-00-55	1718949655738
gs04r2b11	224	0	2545080	1	2024-06-21-08-00-55	1718949655741
gs04r2b04	224	0	2443594	1	2024-06-21-08-00-55	1718949655755
gs04r2b04	224	0	2443594	1	2024-06-21-08-00-55	1718949655757

Figure 4.16 The ranks are always executed in the cpu 0 of the nodes.

The total number of iterations in the trace is 17 (0-16).

Rank ID	Iteration	Operation	Start	End	Count	Start	End
171894965572	0	Setup					
1718950230962	3	ReconfigureStart					
1718950230962	3	ReconfigureStart					
1718950230964	9	ReceiveStart					
1718950230965	9	ReceiveStart					
1718950230967	9	ReceiveStart					
1718950230969	9	ReceiveStart					
1718950230972	3	ReconfigureStart					
1718950230976	9	ReceiveStart					
1718950230976	3	ReconfigureStart					
1718950230977	9	ReceiveStart					
1718950230995	3	ReconfigureStart					
1718950230997	9	ReceiveStart					
1718950231344	10	ReceiveEnd					
1718950231348	10	ReceiveEnd					
1718950231352	4	ReconfigureEnd	4	8	16		
1718950231362	4	ReconfigureEnd	6	8	16		

Figure 4.17 All the ranks create only one execution thread.

There have been 1, 2, 4 and 8 ranks active at the same time. The number of ranks is always a power of 2.

Rank ID	Iteration	Operation	Start	End	Count	Start	End
171894965572	0	Setup					
1718949655721	1	InitStart					
1718949655755	2	InitEnd					
1718949655757	7	IterationStart					
1718949657757	8	IterationEnd					
1718949657759	3	ReconfigureStart					
1718949658113	4	ReconfigureEnd					
1718949658119	5	SendStart					
1718949658243	3	ReconfigureStart					
1718949658243	9	ReceiveStart					
1718949671865	6	SendEnd					
1718949674157	10	ReceiveEnd					
1718949674169	4	ReconfigureEnd					
1718949675213	7	IterationStart					
1718949691214	8	IterationEnd					
1718949691214	3	ReconfigureStart	0	1	1		
1718949691538	4	ReconfigureEnd	0	1	1		
1718949691538	5	SendStart	0	1	1		

Figure 4.18 The number of different processes executing the Rank 0.

The number of ranks changes each time that a reconfiguration with expand or shrink of resources has been performed.

Rank ID	Iteration	Operation	Start	End	Count	Start	End	
2443594	1	2024-06-21-08-00-55	171894965572	7	IterationStart	0	8	0
2444128	1	2024-06-21-08-01-15	1718949675213	7	IterationStart	0	1	1
2447227	1	2024-06-21-08-02-27	1718949747553	7	IterationStart	0	2	2
2447310	1	2024-06-21-08-03-04	1718949784632	7	IterationStart	0	4	3
2447396	1	2024-06-21-08-03-23	1718949803317	7	IterationStart	0	8	4
2447427	1	2024-06-21-08-03-43	1718949823059	7	IterationStart	0	1	5
2447466	1	2024-06-21-08-04-54	1718949895009	7	IterationStart	0	2	6
2447594	1	2024-06-21-08-05-31	1718949931970	7	IterationStart	0	4	7
2447636	1	2024-06-21-08-05-50	1718949950521	7	IterationStart	0	8	8
2447666	1	2024-06-21-08-06-10	1718949970111	7	IterationStart	0	1	9
2447754	1	2024-06-21-08-07-22	1718950042113	7	IterationStart	0	2	10
2447837	1	2024-06-21-08-07-59	1718950079059	7	IterationStart	0	4	11
2447880	1	2024-06-21-08-08-17	1718950097656	7	IterationStart	0	8	12
2447961	1	2024-06-21-08-08-37	1718950117327	7	IterationStart	0	1	13
2448041	1	2024-06-21-08-09-49	1718950169443	7	IterationStart	0	2	14
2448118	1	2024-06-21-08-10-26	1718950225408	7	IterationStart	0	4	15

Figure 4.19 Number of ranks every time that an iteration starts.

The PRV file is obtained by executing the conversion application using as input the CSV trace file. The next step consists of checking the PRV file key points. The execution of the trace conversion application is shown in Figure 4-20.

```
C:\Users\USER\OneDrive\Projects\_TFM\dmrtraceparser>dmrtraceparser example.csv
File 1: example.csv
File processed
Paraver trace file already generated
C:\Users\USER\OneDrive\Projects\_TFM\dmrtraceparser>
```

Figure 4.20 Conversion of the testing CSV trace file to PRV format.

The first point to check is the PRV file's header, to verify that all the fields have been generated in the right way.

```

1 #Paraver (2024/06/21 at 08:19):576862000:8(1,1,1,1,1,1,1,1):1:8(17:1,17:2,17:3,17:4,17:5,17:6,17:7,17:8)
2 2:1:1:1:1:0:0:0
3 2:1:1:1:1:0:2:1
4 2:6:1:6:1:0:0:0
5 2:7:1:7:1:0:0:0
6 2:2:1:2:1:0:0:0
7 2:5:1:5:1:0:0:0
8 2:3:1:3:1:0:0:0
9 2:3:1:3:1:9000:2:1
10 2:6:1:6:1:9000:2:1
11 2:7:1:7:1:10000:2:1
12 2:5:1:5:1:9000:2:1
610 2:4:1:4:16:575963000:6:0
611 2:8:1:8:17:575964000:10:0
612 2:8:1:8:17:575966000:4:0
613 2:3:1:3:17:575968000:10:0
614 2:3:1:3:17:575981000:4:0
615 2:3:1:3:16:576050000:6:0
616 2:6:1:6:17:576050000:10:0
617 2:6:1:6:17:576052000:4:0
618 2:4:1:4:17:576048000:10:0
619 2:2:1:2:16:576048000:6:0
620 2:4:1:4:17:576062000:4:0

```

Figure 4.21 Header and first and last events of the generated PRV trace file.

The first fields of the header are according to the header definition. The total time calculation in the header meets with the time stamp of the last register of the body, so those fields are right.

The rest of the header is analyzed in the following way:

- The number of nodes (8) and the number of cpus per processor (1), meet with the trace definitions done in previous sections for DMR traces.
- The number of applications (8) meets the maximum number of ranks.
- The number of tasks (8) meets with the maximum number of ranks.
- The number of threads per task (17) meets with the number of iterations.

From the analysis done, it can be concluded that the header of the PRV trace file is right. The final check will be to load the PRV trace file in Paraver and check the resources created.

The next step is to analyze the content of the file's body. Due to the body of the trace containing hundreds of lines, it cannot be analyzed line per line. The estimation of the coherence can be done following some rules:

- All the lines must start with the value 2 because all registers are events.
- The second value, the node, must meet with the rank of the trace record.
- The third value, the application number, must be incremental from 1 to the number of files merged during the conversion and must meet with the header values.
- The fourth value, the task, must meet with the rank of the trace record.
- The fifth value, the thread, must be 1 or the iteration of the trace record, it depends on the application execution arguments.
- The sixth value, the timestamp, must be incremental, from 0 to the application execution time.
- The seventh value, the event, must meet with the equivalent event in the trace record but always with the value of the equivalent event of type END.

- The last value, the event value, must be different 0 for the events of the type START and 0 for the events of type END. In this way, Paraver will be able to convert events to states in the right way and assign a unique color.

Figure 4-22 shows the correspondence of the event values between the CSV trace file and the PRV trace file by using some registers of both files.

75	gs04r2b04, 224, 0, 2444128, 1, 2024-06-21-08-01-14, 1718949674157, 10, ReceiveEnd, 0, 1, 1	76	2:1:1:1:2:18436000:10:0
76	gs04r2b04, 224, 0, 2444128, 1, 2024-06-21-08-01-14, 1718949674169, 4, ReconfigureEnd, 0, 1, 1	77	2:1:1:1:2:18448000:4:0
77	gs04r2b04, 224, 0, 2444128, 1, 2024-06-21-08-01-15, 1718949675213, 7, IterationStart, 0, 1, 1	78	2:1:1:1:2:19492000:8:7
78	gs04r2b04, 224, 0, 2444128, 1, 2024-06-21-08-01-31, 1718949691214, 8, IterationEnd, 0, 1, 1	79	2:1:1:1:2:35493000:8:0
79	gs04r2b04, 224, 0, 2444128, 1, 2024-06-21-08-01-31, 1718949691214, 3, ReconfigureStart, 0, 1, 1	80	2:1:1:1:2:35493000:4:1
80	gs04r2b04, 224, 0, 2444128, 1, 2024-06-21-08-01-31, 1718949691538, 4, ReconfigureEnd, 0, 1, 1	81	2:1:1:1:2:35817000:4:0
81	gs04r2b04, 224, 0, 2444128, 1, 2024-06-21-08-01-31, 1718949691538, 5, SendStart, 0, 1, 1	82	2:1:1:1:2:35817000:6:5
82	gs04r2b04, 224, 0, 2447227, 1, 2024-06-21-08-01-31, 1718949691645, 3, ReconfigureStart, 0, 2, 2	83	2:1:1:1:3:35924000:4:1
83	gs04r2b04, 224, 0, 2447227, 1, 2024-06-21-08-01-31, 1718949691646, 9, ReceiveStart, 0, 2, 2	84	2:1:1:1:3:35925000:10:9
84	gs04r2b05, 224, 0, 2515996, 1, 2024-06-21-08-01-31, 1718949691646, 3, ReconfigureStart, 1, 2, 2	85	2:2:1:2:3:35925000:4:1
85	gs04r2b05, 224, 0, 2515996, 1, 2024-06-21-08-01-31, 1718949691648, 9, ReceiveStart, 1, 2, 2	86	2:2:1:2:3:35927000:10:9
86	gs04r2b04, 224, 0, 2447227, 1, 2024-06-21-08-02-26, 1718949746052, 10, ReceiveEnd, 0, 2, 2	87	2:1:1:1:3:90331000:10:0
87	gs04r2b04, 224, 0, 2447227, 1, 2024-06-21-08-02-26, 1718949746054, 4, ReconfigureEnd, 0, 2, 2	88	2:1:1:1:3:90333000:4:0
88	gs04r2b04, 224, 0, 2444128, 1, 2024-06-21-08-02-27, 1718949747534, 6, SendEnd, 0, 1, 1	89	2:1:1:1:2:91813000:6:0
89	gs04r2b05, 224, 0, 2515996, 1, 2024-06-21-08-02-27, 1718949747534, 10, ReceiveEnd, 1, 2, 2	90	2:2:1:2:3:91813000:10:0
90	gs04r2b05, 224, 0, 2515996, 1, 2024-06-21-08-02-27, 1718949747537, 4, ReconfigureEnd, 1, 2, 2	91	2:2:1:2:3:91816000:4:0
91	gs04r2b05, 224, 0, 2515996, 1, 2024-06-21-08-02-27, 1718949747551, 7, IterationStart, 1, 2, 2	92	2:2:1:2:3:91830000:8:7
92	gs04r2b04, 224, 0, 2447227, 1, 2024-06-21-08-02-27, 1718949747553, 7, IterationStart, 0, 2, 2	93	2:1:1:1:3:91832000:8:7
93	gs04r2b05, 224, 0, 2515996, 1, 2024-06-21-08-02-35, 1718949755552, 8, IterationEnd, 1, 2, 2	94	2:2:1:2:3:99831000:8:0
94	gs04r2b05, 224, 0, 2515996, 1, 2024-06-21-08-02-35, 1718949755553, 3, ReconfigureStart, 1, 2, 2	95	2:2:1:2:3:99832000:4:1
95	gs04r2b04, 224, 0, 2447227, 1, 2024-06-21-08-02-35, 1718949755570, 8, IterationEnd, 0, 2, 2	96	2:1:1:1:3:99849000:8:0

Figure 4.22 Comparison between registers of the CSV file and the PRV file.

There is the following relevant information:

- Line N of the trace file meets with line N+1 of the Paraver file.
- The thread N in the Paraver file meets with the Iteration + 1 in the trace file.
- Node id and tasks id in the Paraver file meet with rank + 1 in the trace file
- In the Paraver trace file all event values correspond with events of type END. The events of type START and type END, are differentiated by the last field, the event value.
- The timestamps of the registers of both files are the same, but in the case of the PRV trace file they start from a timestamp of 0.

The next test to execute consists of merging two CSV trace files in a unique PRV trace file and repeating the validation procedures. To ease the process, the test can be done using two times the same CSV file.

```
C:\Users\USER\OneDrive\Projects_TFM\dmrtraceparser>dmrtraceparser example.csv example.csv
File 1: example.csv
File processed
File 2: example.csv
File processed
Paraver trace file already generated
C:\Users\USER\OneDrive\Projects_TFM\dmrtraceparser>
```

Figure 4.23 Merge of two CSV trace files in a unique PRV trace file.

As it has been converted two times to the same file, the total time, and the number of nodes and cpus, in the header, must be the same as obtained when a unique file was converted. Where it is expected a change in the header is in the number of applications sections, due to it being defined that two files are considered different applications.

```

1 #Paraver (2024/06/22 at 13:49):576862000:8(1,1,1,1,1,1,1,1):2:8(17:1,17:2,17:3,17:4,17:5,17:6,17:7,17:8)
2
3 2:1:1:1:1:0:0:0
4 2:1:1:1:1:0:2:1
5 2:6:1:6:1:0:0:0
6 2:7:1:7:1:0:0:0
7 2:2:1:2:1:0:0:0
8 2:5:1:5:1:0:0:0
9 2:3:1:3:1:0:0:0
10 2:3:1:3:1:9000:2:1

```

Figure 4.24 Paraver trace file header after merging two times the same file.

The number of applications has been increased from 1 to 2, and the structure of the applications has been duplicated, as it was expected.

The last test to do is to analyze the point of the PRV trace file when the two CSV trace files are joined.

```

615 2:3:1:3:17:575981000:4:0
616 2:3:1:3:16:576050000:6:0
617 2:6:1:6:17:576050000:10:0
618 2:6:1:6:17:576052000:4:0
619 2:4:1:4:17:576848000:10:0
620 2:2:1:2:16:576848000:6:0
621 2:4:1:4:17:576862000:4:0
622 2:1:2:1:1:0:0:0
623 2:1:2:1:1:0:2:1
624 2:6:2:6:1:0:0:0
625 2:7:2:7:1:0:0:0
626 2:2:2:2:1:0:0:0
627 2:5:2:5:1:0:0:0
628 2:3:2:3:1:0:0:0
629 2:3:2:3:1:9000:2:1
630 2:6:2:6:1:9000:2:1

```

Figure 4.25 Point of the PRV trace file where two CSV trace files are joined.

The next step consists of loading the PRV trace files in Paraver and executing several visualization operations. Paraver detects if a trace file has a wrong format and aborts the file loading process.

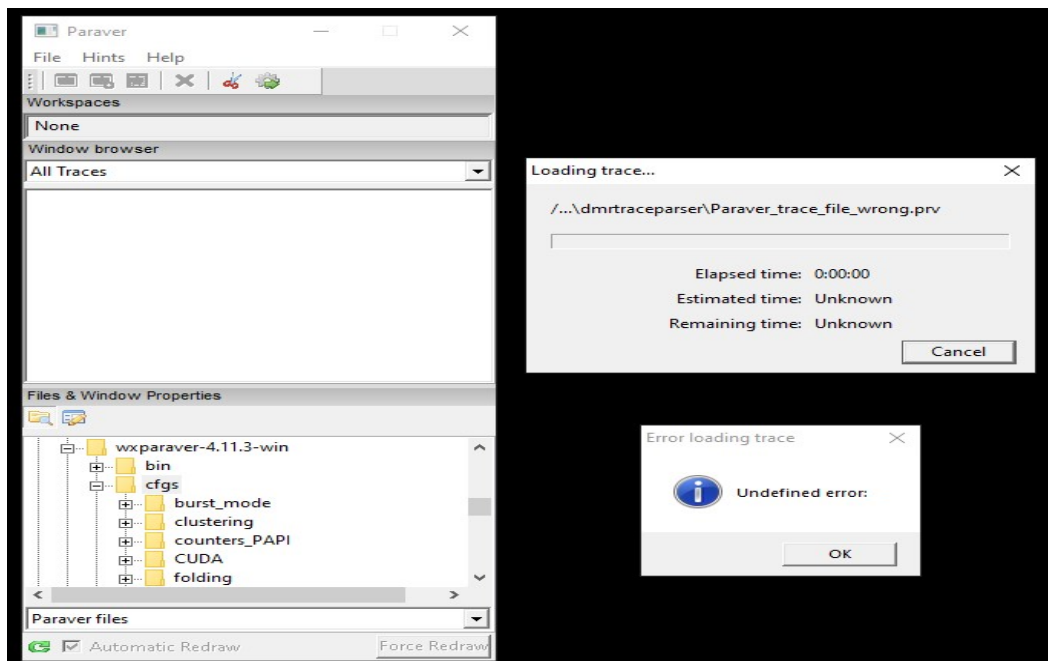


Figure 4.26. Paraver error message loading a wrong trace file.

The first test is to load the PRV trace file obtained by converting a unique CSV trace file, creating a visualization window, and showing the events.

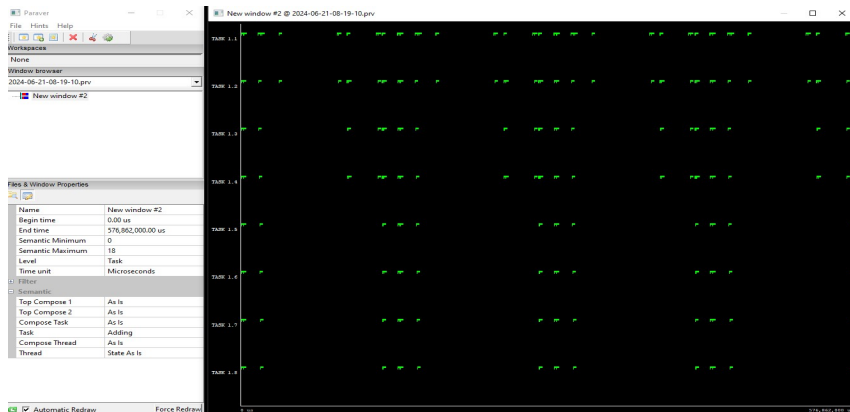


Figure 4.27 Generated trace file loaded in Paraver.

Paraver has uploaded the trace file without errors and the distribution of the events is coherent with the traces recorded.

The next step is generating the states by using the semantics of Paraver.

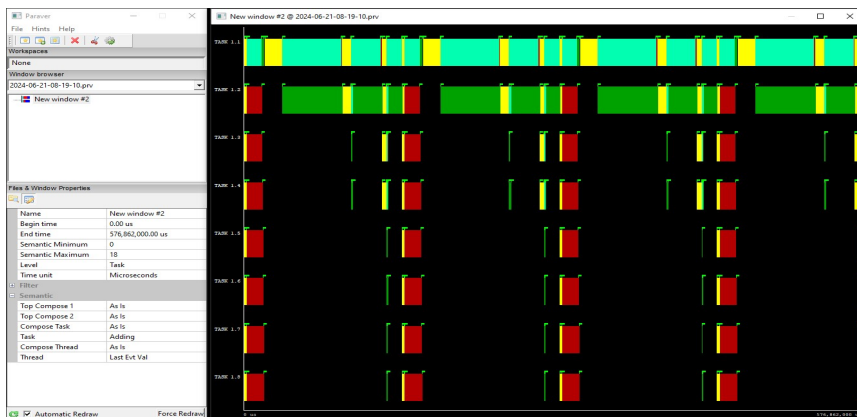


Figure 4.28 Events converted to states in Paraver.

To analyze the resulting traces, they must be divided into several sections.

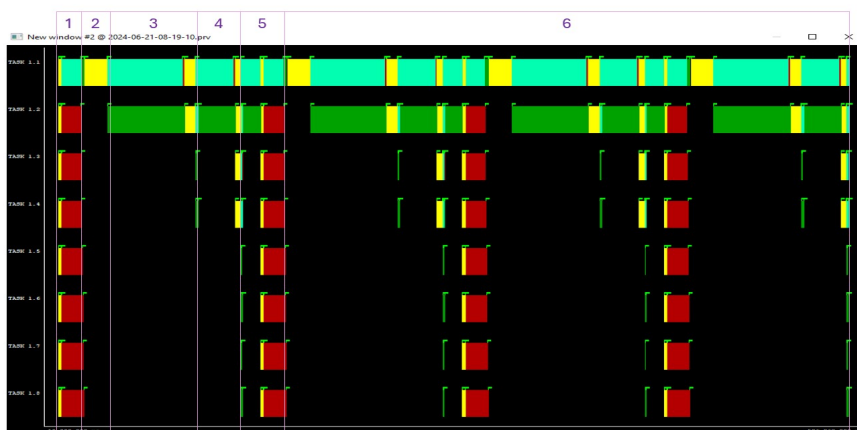


Figure 4.29 Timeline of the instance execution divided in sections.

Figure 4-29 shows different regions of the timeline of the execution of an isolated instance of the application. The yellow sections are computation periods, and the red and the green periods are communications for sending and receiving computing results and data to compute, respectively. The light blue periods are periods, where send and receive communications, occur simultaneously but in different processes. The description of each section is the following:

1. The application starts with 8 nodes. The nodes compute the data and perform a reconfiguration. After the reconfiguration, the computing nodes send the results.
2. After the reconfiguration, only 1 node continues assigned to the application. The node receives the data to compute, performs the computation, and after, performs a reconfiguration. After processing the reconfiguration, the node sends the results.
3. After the reconfiguration, 2 nodes are assigned to the application. The nodes receive the data to compute, perform the computation, and after, perform a reconfiguration. After processing the reconfiguration, the nodes send the results.
4. After the reconfiguration, 4 nodes are assigned to the application. The nodes receive the data to compute, perform the computation, and after, perform a reconfiguration. After processing the reconfiguration, the nodes send the results.
5. After the reconfiguration, 8 nodes are assigned to the application. The nodes receive the data to compute, perform the computation, and after, perform a reconfiguration. After processing the reconfiguration, the nodes send the results.
6. The execution repeats the explained cycle several times.

The final validation steps consist of testing the creation of Paraver trace files by merging two CSV trace files. To perform the tests easily, it has been used two times the same trace file. The result expected is just a duplication of the applications with a repetition of the timelines of the events and states.

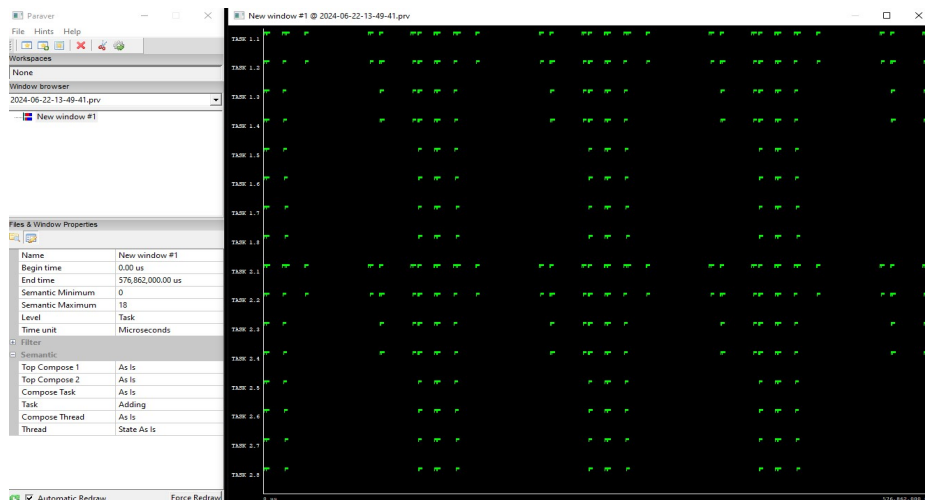


Figure 4.30 Paraver trace file from two trace files.

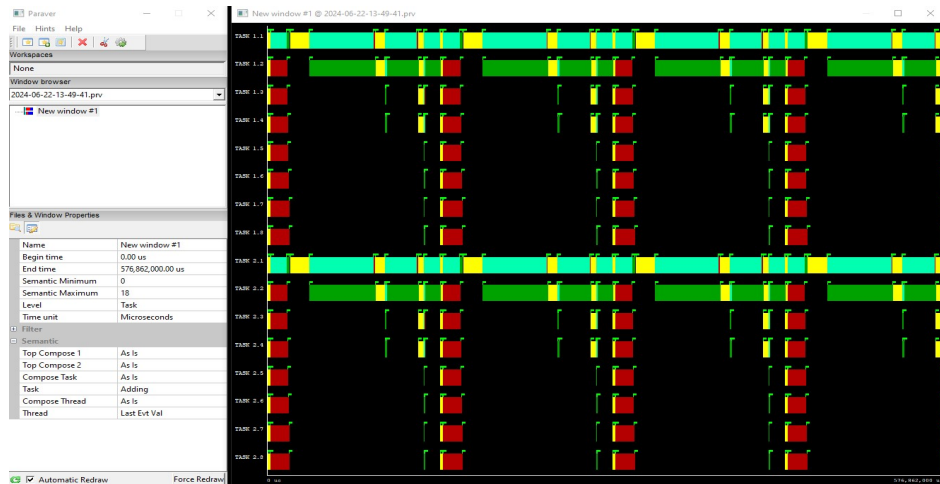


Figure 4.31 Paraver file from two trace files converted to states.

The conversion of two CSV trace files to a unique PRV trace file seems to be right, and Paraver can upload the resultant trace file without errors and create the states from the trace events.

After the battery of tests done over the output files generated by the trace converting application, it can be concluded that the application runs fine and converts the CSV trace files to PRV trace files in the right way.

To close the test and validation section, it should be noted that the final validation of the trace conversion application will be done when it is used to trace real malleable applications in the last part of this master thesis.

4.8. Trace converter command line arguments

The trace converter is a command line application. It can be executed with different types of arguments:

- `dmrtraceparser [--iters] [--noiters] [filename.csv] [foldername]`

Each argument has a different effect on the behaviour of the conversion application:

- `--iters`. The field “iteration” of each DMRTRACE record is used to fill the field “thread” of each Paraver trace record.
- `--noiters`. The field “thread” of each Paraver trace record is always “1”. It does not allow Paraver to show simultaneous states in different threads.
- `[filename.csv]`. One or several trace files to convert to Paraver.
- `[foldername]`. If a folder name is used, the trace converter merges and converts all the CSV files inside the folder into a unique Paraver file.
- If the arguments `-iters` and `-noiters` are not used, the trace converter uses a counter of reconfigurations.
- If files or folders are not specified, the converter merges and converts all the CSV files inside the execution folder.

To ease the use of the converter a common practice consists of creating script files to call the converter using the right arguments.

The usual way of using the conversion application is not to use the command line arguments `-iters` and `-noiters`. In this way, the converter creates Paraver trace files where the thread field is a counter of reconfigurations. This conversion strategy allows Paraver the simultaneous state visualization without creating too many threads to visualize.

The using of the argument `-iters` with trace files with many iterations, can cause a collapse in Paraver. It is due to excessive thread to visualize.

The using of the argument `-noiters`, causes the impossibility of visualizing simultaneous states in Paraver. It is due to the thread identifier of all the events is the same and equal to "1".

4.9. License Agreement

The trace converting application is an open-source software application that complements the malleability trace generation library. For this reason, the license agreement adopted is GPLv3, the same adopted for the trace generation library and other software applications developed at the BSC.

To avoid future misunderstandings, the license agreement is included in the header of each source code file of the application.

More information about the GPLv3 license agreement is available at the following link.

- GPLv3 (<https://www.gnu.org/licenses/gpl-3.0.html>)

4.10. Summary of the trace converter application section

This section started by explaining how the Paraver trace file format is defined and with the definition of the trace format converter application requirements. Afterwards, the application design and the application development were explained in detail.

Finally, it was explained how the application has been tested and validated by using a trace file from the execution of a simple malleable parallel application, generating a Paraver trace file, comparing the content of both files and finally loading the resulting trace file into Paraver and executing several visualization operations.

As it has been explained at the final of the test and validation section the application will be fully validated when is used to study several real applications in the last part of this document.

5. Study of several malleable applications

In this section, the trace library and the trace conversion application will be used to study the behaviour of several malleable parallel applications. First, a common study procedure will be defined, and after that, the procedure will be applied in the study of several applications.

The main goals of this section are, to show how to apply the traceability library to record relevant events of a malleable application, how to convert the DMR traces to the Paraver trace format, and how to use Paraver for monitoring the application execution and extract some relevant metrics.

5.1. Application study procedure

The first step for the study is defining a common procedure valid for the study of any application. The goal is to have a set of basic steps useful to study the behaviour of any malleable application.

The defined procedure is shown in Table 6.

Application Study Procedure
Making the application malleable, according to the DMR procedures
Defining a policy of resource reconfiguration
Launching the application
Getting the DMR trace files generated during the execution of the application
Converting the generated DMR trace files into a Paraver trace file
Opening Paraver and load the trace
Creating a visualization window
Activating the event flag visualisation in the window
Select the last event val in the thread field in the semantics properties window
Select the level task in the window properties
Visual analysis of the application
Creating a new histogram from the visualisation window
Setting the appropriate histogram visualisation options
Setting the appropriate trace objects to get different levels of detail of metrics
Statistical analysis of the application

Table 6. Malleable application study procedure.

To test the applications, it is necessary to define a resource reconfiguration policy. There are several ways of testing a malleable application:

- Launching a unique instance of the application and changing the available resources to force the application to change its resources, when a reconfiguration is performed.

- Launching multiple instances of the application and forcing them to share the available resources.

To select a reconfiguration policy, it is necessary to modify a source code file and recompile it. To ease the task of defining the policy from scratch, there are several pre-defined policies in the source code. So that, it is possible to select one of the predefined policies just by commenting and uncommenting source code and recompiling it.

The test of the applications is going to be performed using a unique instance and selecting the policy that expands the resources from 1 to 8 in powers of 2 and shrinks them from 8 to 1.

The source code of this policy is placed in a slurm subfolder of the DMR folder installation folder of the user:

- `~/dmr/slurm-spawn/src/plugins/select/linear/select-linear.c`

The code that must be selected and recompiled is shown in Figure 5-1.

```

4084 /*****
4085 /*****
4086 /*****      TALP      *****/
4087 /*****
4088 /*****
4089 > /****
4127 /*****
4128 /*****
4129 /*****      RANDOM      *****/
4130 /*****
4131 /*****
4132 > /****
4174 /*****
4175 /*****
4176 /*****
4177 /*****      expand step by step up to 8, and then shrink to 1 *****/
4178 /*****
4179 /*****
4180
4181 extern int select_p_select_jobinfo_get(select_jobinfo_t *jobinfo,
4182 enum select_jobdata_type data_type, void *data) {
4183     if (data_type == SELECT_JOBDATA_INFO) {
4184         hostlist_t hl = slurm_hostlist_create(jobinfo->hostlist);
4185         char *host = slurm_hostlist_shift(hl);
4186         strcpy((char *) data, host);
4187         slurm_hostlist_destroy(hl);
4188         return SLURM_SUCCESS;
4189     } else if (data_type == SELECT_JOBDATA_ACTION) {
4190         jobinfo->action = 0;
4191         jobinfo->resultantNodes = jobinfo->currentNodes;
4192         if (jobinfo->currentNodes < 8){
4193             jobinfo->action = 1;
4194             jobinfo->resultantNodes = jobinfo->currentNodes * 2;
4195         } else if (jobinfo->currentNodes == 8) {}
4196         jobinfo->action = 2;
4197         jobinfo->resultantNodes = 1;
4198     }
4199     }
4200     return SLURM_SUCCESS;
4201 }
4202
4203 /*****
4204 /*****
4205 /*****      PhD      *****/
4206 /*****
4207 /*****
4208 /*****
4209 > /****
4273

```

Figure 5.1 Reconfiguration policy step up to 8.

The code of the policies TALP, RANDOM and PhD must be commented to avoid being compiled.

This source code file has a link in the DMR folder, and the code can be compiled by executing the Makefile, also used to compile the DMR library and the DMRTRACE library.

5.2. Paraver procedures for application studies

This section is a revision of common Paraver procedures, to avoid overloading the study of different applications with redundant information. Some basic procedures, such as loading a trace or converting events to states, have been explained in previous sections, this section will be a short remembering of those procedures and some other useful ones.

To study the job execution by using Paraver, it is necessary to convert the trace into the Paraver trace file format. The DMRTRACEPARSER application does this operation in different ways, one of them is passing the trace files as arguments of the application.

- `dmrtraceparser.exe file1.csv file2.csv ... fileN.csv`

A trace file can be opened in Paraver by using the menu file.

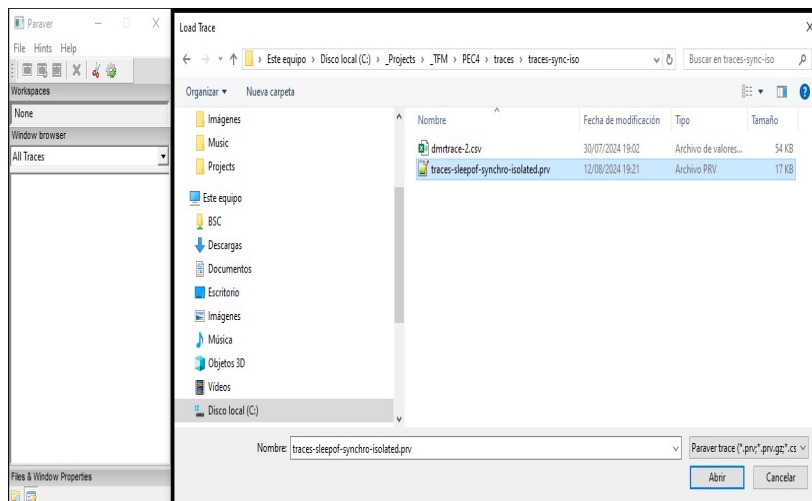


Figure 5.2 Opening a trace file in Paraver.

A timeline window can be created by clicking over an icon. After that, the event flags can be activated by using the window contextual menu.

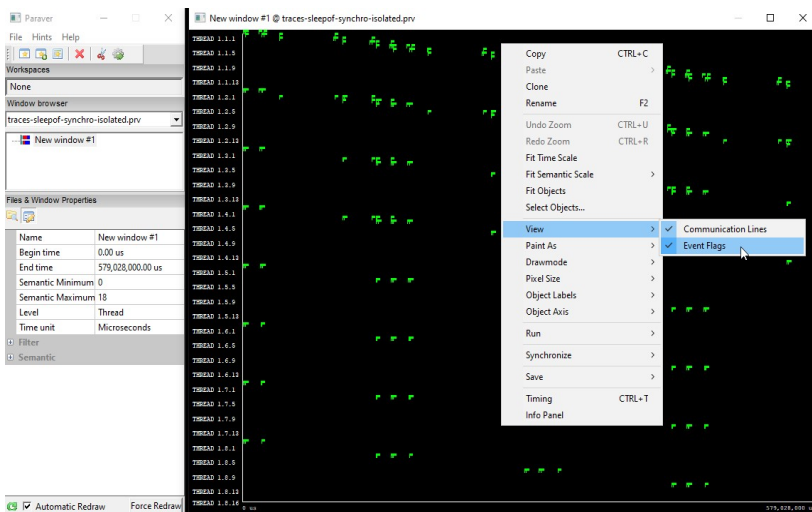


Figure 5.3 Opening a timeline window and activating the event flags.

For the right visualization, it is needed to convert the events to states. It can be done by using the properties of the timeline window, in the semantics section by selecting the option “Last Evt Val” in the thread field. The result will be the visualization of the timeline of the states of the application.

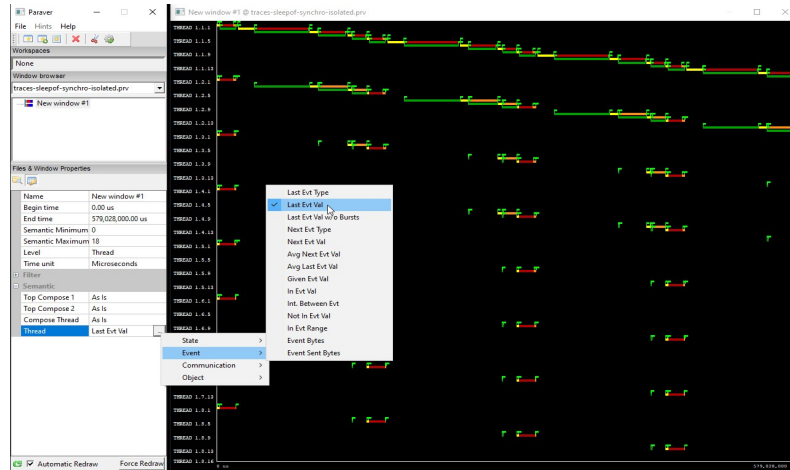


Figure 5.4 States of the job timeline.

By default, the visualization window is loaded showing information at thread level and in a time scale of microseconds. Microseconds is too much resolution for measuring times in malleable applications, so that is more useful working with milliseconds by changing the property “time unit” in the window properties.

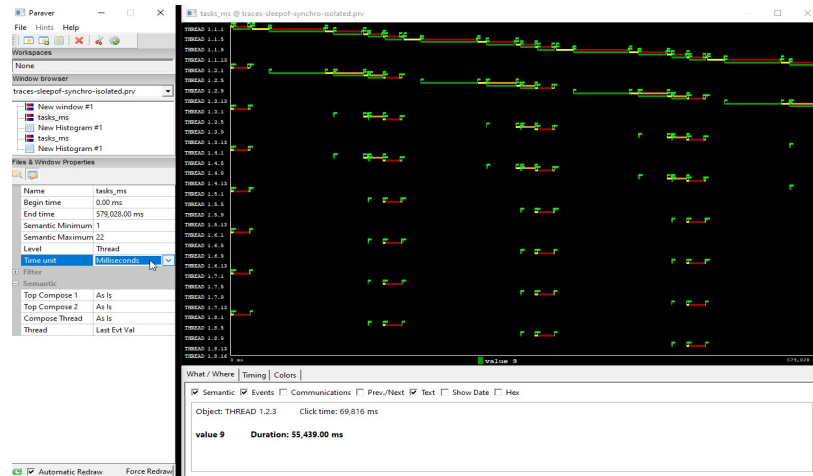


Figure 5.5 Visualization of the time scale in milliseconds.

At the thread level, Paraver shows in the Y axis, the application, the rank, and the thread, and in the X axis, the time. The application level is always 1 because the trace corresponds with the execution of just one instance of the application. The task level corresponds with the ranks, from 1 to 8, changing after each reconfiguration, following the previously selected resource availability policy. The thread level corresponds with the iteration where there are reconfigurations, and therefore send and receive communications.

Paraver shows in red (value 5) the send communications, in yellow (value 7) the computing time, in green (value 9) the receive communications, and in orange (value 13) the detaching process. The DMRTRACEPARSER application puts the codes of these colors in the event value field of each register of the generated Paraver trace files.

Paraver allows users to change the visualization level. The level can be changed from thread to task, application, cpu, and others. Those levels of visualization hide part of the information but give clearer information when lower levels of visualization show too much information.

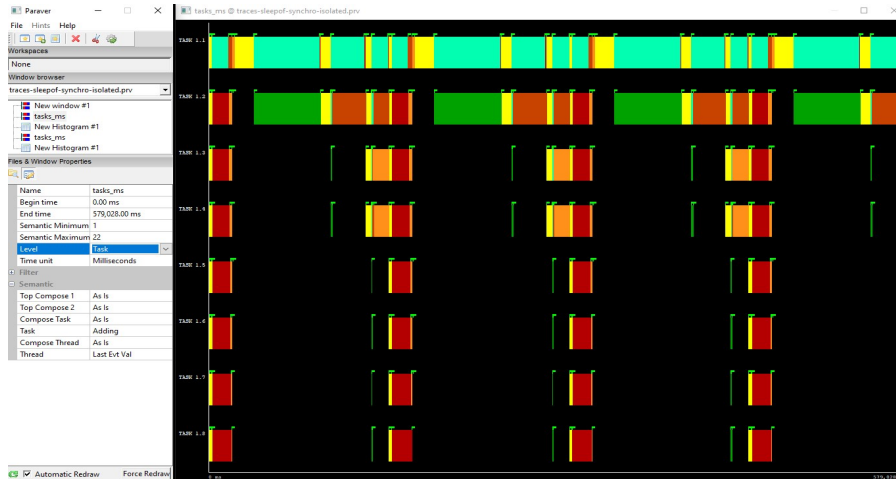


Figure 5.6 Visualization of the job execution by using the task level.

The visualization window allows operations of filtering, zooming and scrolling, to examine trace details, and activating the info panel, by using the context menu, is possible to get times of states by double-clicking over them. By using those features is also possible to analyze in detail part of the trace.

Additionally, Paraver allows the extraction of statistics of the traces. It is possible to create a statistics view by clicking over the icon “new histogram” and selecting a timeline window.

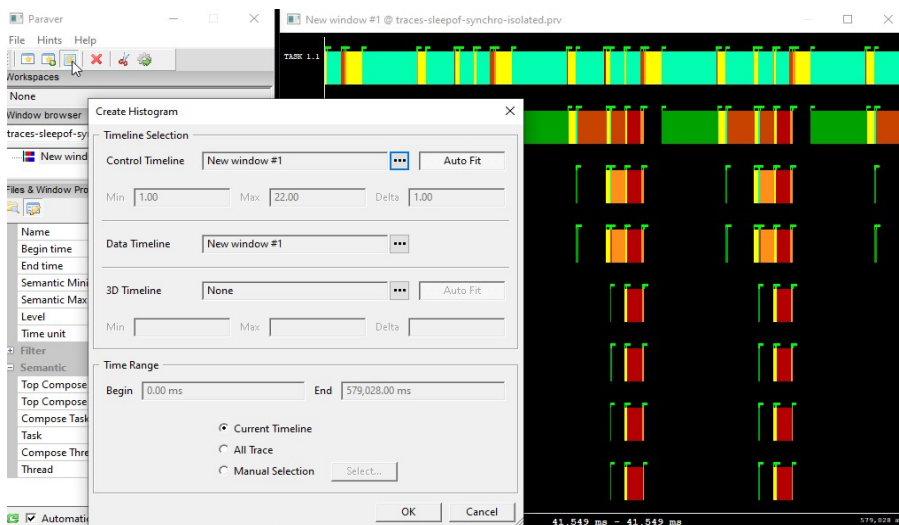


Figure 5.7 Creation of a histogram window for statistics.

After creating a new histogram, some adjustments in the screen and the properties window must be done. It can visualize the percentages of computation times (state 7), and communications (states 5 and 9). Some basic adjustments can be made to improve and customize data visualization:

- Activating the lens icon in the histogram screen, to change the visualization from color histogram to color and data histogram.
- Activating or deactivating colors.
- Selecting a horizontal or vertical view.
- Showing or hiding empty columns, to show or hide all the possible states, even these states won't be in the trace file.
- Enabling or disabling header colors, to show or hide the color representation of all possible states. Those colors are the same ones used in the timeline visualization window.
- Enabling or disabling totals, to show or hide the overall statistics of the selected objects.
- Change the statistics configuration in the property window, for instance to percentage of time.

After configuring the histogram view in the right way, the statistics windows show useful information to make an objective evaluation of a job execution.

A useful information to determine the efficiency of the execution of an application can be the percentage of time dedicated to computation and communications.

	1	3	5	7	9	11	13	14	20	22
TASK 1.1	0.01 %	0.85 %	2.91 %	20.08 %	0.00 %	0.00 %	1.41 %	72.81 %	-	1.95 %
TASK 1.2	0.01 %	0.70 %	11.41 %	10.32 %	44.07 %	0.00 %	1.96 %	1.96 %	0.00 %	29.57 %
TASK 1.3	0.03 %	1.61 %	43.76 %	14.89 %	1.84 %	0.01 %	36.32 %	1.41 %	-	0.14 %
TASK 1.4	0.03 %	1.59 %	43.56 %	14.61 %	3.76 %	0.00 %	33.00 %	1.71 %	-	1.74 %
TASK 1.5	0.06 %	1.46 %	80.05 %	10.50 %	1.74 %	-	6.21 %	-	-	-
TASK 1.6	0.05 %	1.45 %	80.71 %	10.37 %	3.17 %	-	4.25 %	-	-	-
TASK 1.7	0.05 %	1.46 %	84.24 %	10.51 %	1.43 %	-	2.31 %	-	-	-
TASK 1.8	0.06 %	1.42 %	85.08 %	10.35 %	3.07 %	-	0.01 %	-	-	-
Total	0.29 %	10.53 %	431.72 %	101.63 %	59.08 %	0.01 %	85.46 %	77.89 %	0.00 %	33.40 %
Average	0.04 %	1.32 %	53.96 %	12.70 %	7.39 %	0.00 %	10.68 %	19.47 %	0.00 %	8.35 %
Maximum	0.06 %	1.61 %	85.08 %	20.08 %	44.07 %	0.01 %	36.32 %	72.81 %	0.00 %	29.57 %
Minimum	0.01 %	0.70 %	2.91 %	10.32 %	0.00 %	0.00 %	0.01 %	1.41 %	0.00 %	0.14 %
StDev	0.02 %	0.32 %	31.45 %	3.34 %	13.91 %	0.00 %	13.98 %	30.79 %	0 %	12.27 %
Avg/Max	0.60	0.82	0.63	0.63	0.17	0.46	0.29	0.27	1	0.28

Figure 5.8 Statistics window in Paraver.

5.3. Basic study of the sleepOf application

The sleepOf application is the application that comes with the DMR library. The application executes a sleep function in multiple ranks. The sleep function suspends the execution of instructions in each rank for some seconds.

The application executes multiple iterations and after each iteration checks if must reconfigure its resources. The sleep time depends on the number of resources assigned to the application. The sleep time is defined in the function compute of the application.

```
98 void compute(int i, int world_size, int world_rank)
99 {
100     int durStep = DURATION / world_size;
101     sleep(durStep);
102 }
```

Figure 5.9 Function compute of the sleepOf application.

The sleep duration will be a constant (16) divided by the number of ranks of the application. The computation time depends on the number of resources.

The first step to studying the application is submitting a job. The job is submitted via a sbatch file:

- sbatch mnv_submission.sbatch

The job will end after a predefined number of iterations will be completed.

After the job execution, a CSV trace file is generated. It contains all the events created during the execution of the job. The CSV trace file is converted to a Paraver trace file by using the DMRTRACEPARSER application, to be loaded in Paraver. Figure 5-10 shows the thread and task views of the sleepOf application execution in Paraver.

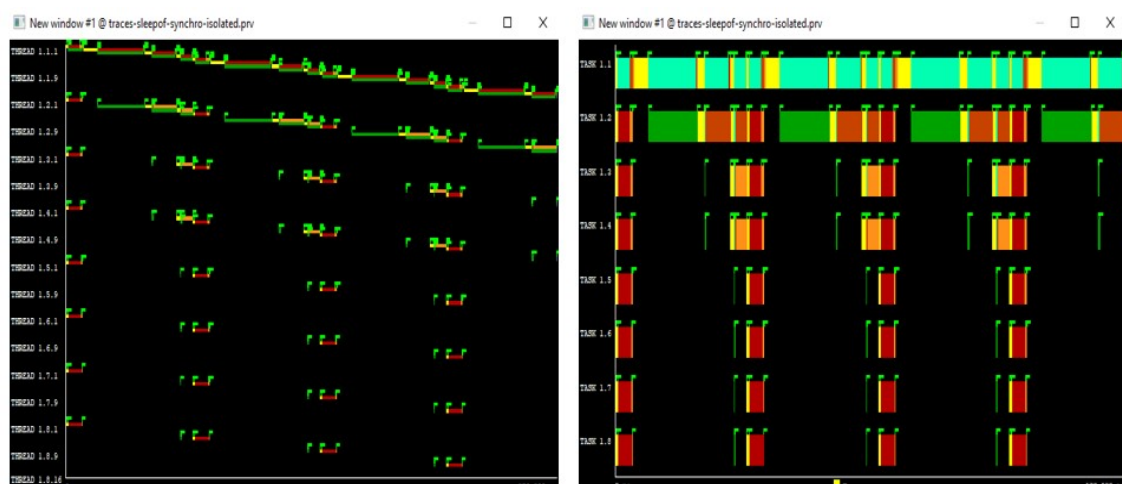


Figure 5.10 Thread and Task views of the sleepOf application.

The thread and the task views can show a complete cycle of reconfigurations from 1 to 8 resources. The comparison of both visualizations allows to decide the more useful view.

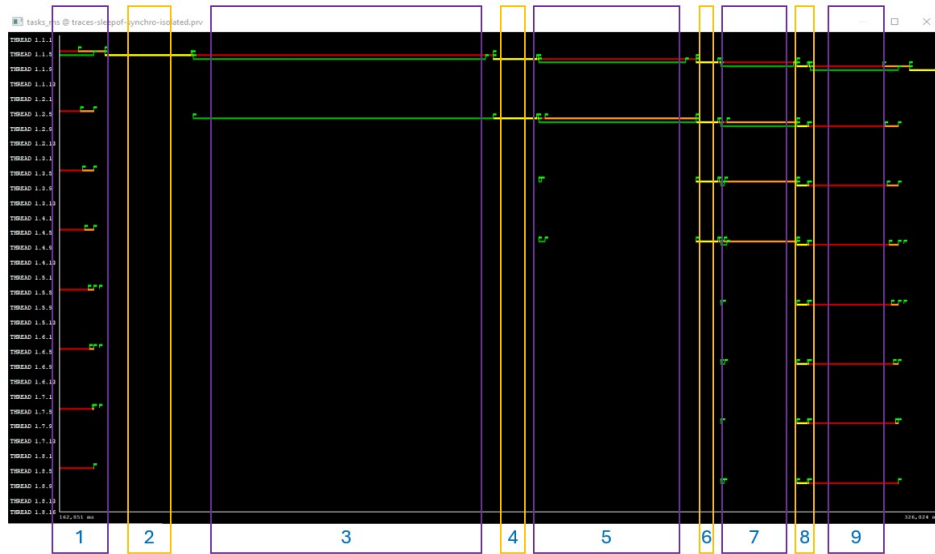


Figure 5.11 Reconfigurations cycle from 1 to 8 resources in the thread view.

Changing to the tasks view, the information is compacted at task level. Paraver creates new states and colors to represent the processes executed in the same rank at the same time but in different threads.

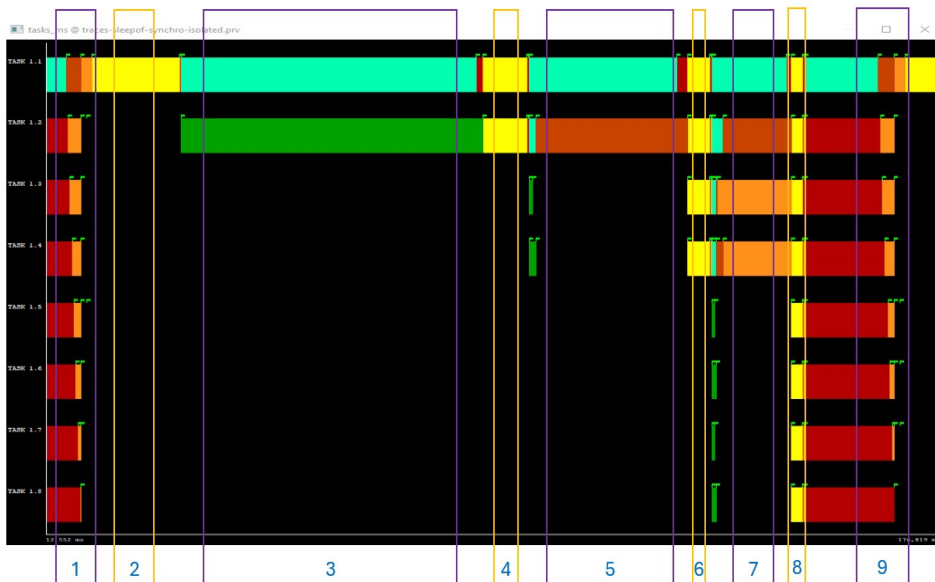


Figure 5.12 Reconfigurations cycle from 1 to 8 resources in task view.

After comparing the threads and the task view, it seems easier to use the task view, even is a compacted view with less information.

In the task view, Paraver has created a new state, state 14, represented in light blue, which is the composition of the states sending and receiving processed simultaneously in the same rank but in different threads.

The complete cycle of reconfigurations from 1 to 8 resources contains several reconfigurations. Following the resource availability policy, the application expands its resources in powers of 2 from 1 to 8 resources. After a complete cycle, the application shrinks its resources to 1 to repeat the cycle. The has been divided into several sections to be explained:

1. Computation with 8 resources has finished, all the ranks send results to rank 0 and rank 0 sends data to compute to itself.
2. Rank 0 computes.
3. Reconfiguration, rank 0 sends results to itself and sends data to compute to ranks 0 and 1.
4. Ranks 0 and 1 compute.
5. Reconfiguration, ranks 0 and 1 send results to rank 0 and rank 0 sends data to compute to ranks 0 to 3.
6. Ranks 0 to 3 compute.
7. Reconfiguration,, ranks 0 to 3 send results to rank 0 and rank 0 sends data to compute to rank 0 to 7.
8. Ranks 0 to 7 compute.
9. Reconfiguration, ranks 0 to 7 send results to rank 0 and rank 0 sends data to compute to itself, starting a new cycle of reconfigurations.

One of the analyses to make over the sleepOf application consists of checking the computing time. It should depend on the number of resources allocated for the application. The computing time must follow the formula $ct = 16 / nr$, where ct is the computing time and nr is the number of resources.

The computation time can be obtained from the task visualization window by activating the info panel and double-clicking over the different compute states (yellow). Paraver will show the state time in the info panel.

The following window shows the different execution times of each different states of the application sleepOf.



Figure 5.13 Computation times depend on resources.

Figure 5-13 shows that the computation times are proportional to the assigned resources. The computation time changes from 16 seconds, when just one resource is used by the application, to 2 seconds, when 8 resources are used.

Reviewing the job execution timeline, there is an important difference between the time necessary to receive data in the ranks 0 and 1 against the needed time to receive data in the rest of the ranks. In the case of rank 0 the difference also exists in the time to send data.



Figure 5.14 Unbalanced communication times.

To find an explanation for that difference, the source code of the sleepOf application has been reviewed and a possible improvement has been identified. The functions `recv_shrink()` and `send_expand()` use synchronous communications which could cause rank 0 and rank 1 to wait until they finish. To confirm this hypothesis, a change in communications has been made, the synchronous communications have been changed by asynchronously.

```

53 void recv_shrink(double **data, int size)
54 {
55     int my_rank, comm_size, parent_size, src, factor, inIPart, i;
56     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
57     MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
58     MPI_Comm_remote_size(MPI_COMM_WORLD, &parent_size);
59     factor = parent_size / comm_size;
60     int localSize = size / comm_size;
61     int remoteSize = size / parent_size;
62     MPI_Request request[10];
63     *data = (double **)malloc(localSize * sizeof(double));
64     for (i = 0; i < factor; i++)
65     {
66         src = my_rank * factor + i;
67         inIPart = remoteSize * i;
68         MPI_Recv(*data + inIPart, remoteSize, MPI_DOUBLE, src, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
69     }
70 }
71
72 void send_expand(double *data, int size)
73 {
74     int my_rank, comm_size, intercomm_size, factor, dst, inIPart;
75     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
76     MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
77     MPI_Comm_remote_size(MPI_COMM_WORLD, &intercomm_size);
78     factor = intercomm_size / comm_size;
79     int remoteSize = size / intercomm_size;
80     MPI_Request request[10];
81     for (int i = 0; i < factor; i++)
82     {
83         dst = my_rank * factor + i;
84         inIPart = remoteSize * i;
85         MPI_Send(data + inIPart, remoteSize, MPI_DOUBLE, dst, 0, MPI_COMM_WORLD);
86     }
87 }
88
89 void recv_shrink(double **data, int size)
90 {
91     int my_rank, comm_size, parent_size, src, factor, inIPart, i;
92     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
93     MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
94     MPI_Comm_remote_size(MPI_COMM_WORLD, &parent_size);
95     factor = parent_size / comm_size;
96     int localSize = size / comm_size;
97     int remoteSize = size / parent_size;
98     MPI_Request request[10];
99     *data = (double **)malloc(localSize * sizeof(double));
100    for (i = 0; i < factor; i++)
101    {
102        src = my_rank * factor + i;
103        inIPart = remoteSize * i;
104        MPI_Irecv(*data + inIPart, remoteSize, MPI_DOUBLE, src, 0, MPI_COMM_WORLD, &request[i]);
105    }
106    for (int i = 0; i < factor; i++)
107    {
108        MPI_Wait(&request[i], MPI_STATUS_IGNORE);
109    }
110 }
111
112 void send_expand(double *data, int size)
113 {
114     int my_rank, comm_size, intercomm_size, factor, dst, inIPart;
115     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
116     MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
117     MPI_Comm_remote_size(MPI_COMM_WORLD, &intercomm_size);
118     factor = intercomm_size / comm_size;
119     int remoteSize = size / intercomm_size;
120     MPI_Request request[10];
121     for (int i = 0; i < factor; i++)
122     {
123         dst = my_rank * factor + i;
124         inIPart = remoteSize * i;
125         MPI_Isend(data + inIPart, remoteSize, MPI_DOUBLE, dst, 0, MPI_COMM_WORLD, &request[i]);
126     }
127     for (int i = 0; i < factor; i++)
128     {
129         MPI_Wait(&request[i], MPI_STATUS_IGNORE);
130     }
131 }

```

Figure 5.15 Parallelization of communications.

After applying the changes in the source code, recompiling the application, and executing it, the traces can be uploaded in Paraver to review if the changes in the source code have produced changes in the communications times.

Communication times can be easily compared by visual analysis.

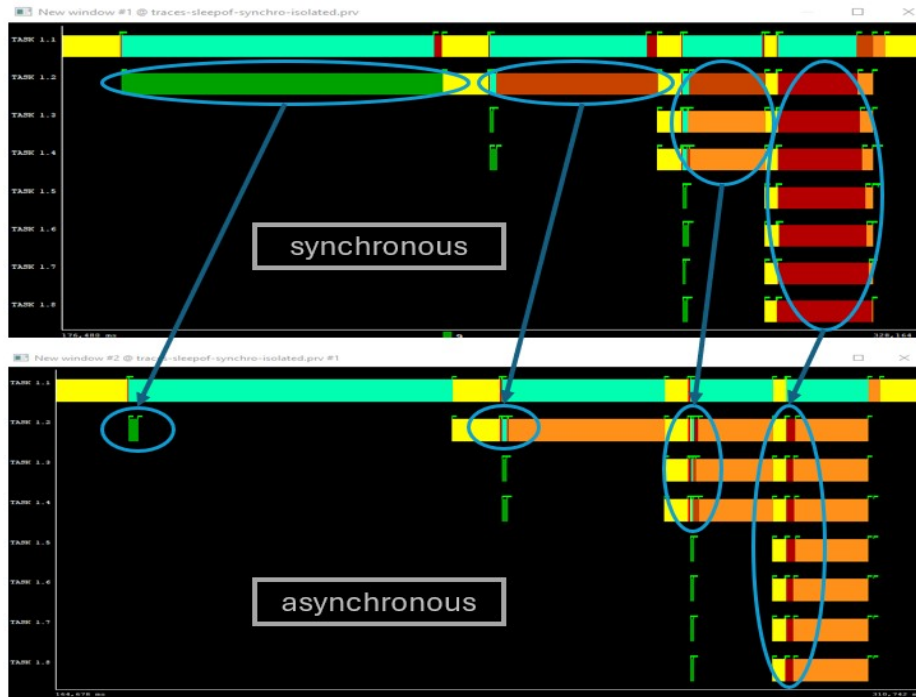


Figure 5.16 Synchronous versus asynchronous states.

The visual comparison shows improvements in the rank 1 data transfers, receiving data for computing. The rest of the ranks are also improved when they send results. However, there are no improvements in the rank 0 data transfers, which causes the overall performance of the application to be the same. With the new approach, the time previously dedicated by the ranks for sending results (red) has been used to waiting for detaching (orange).

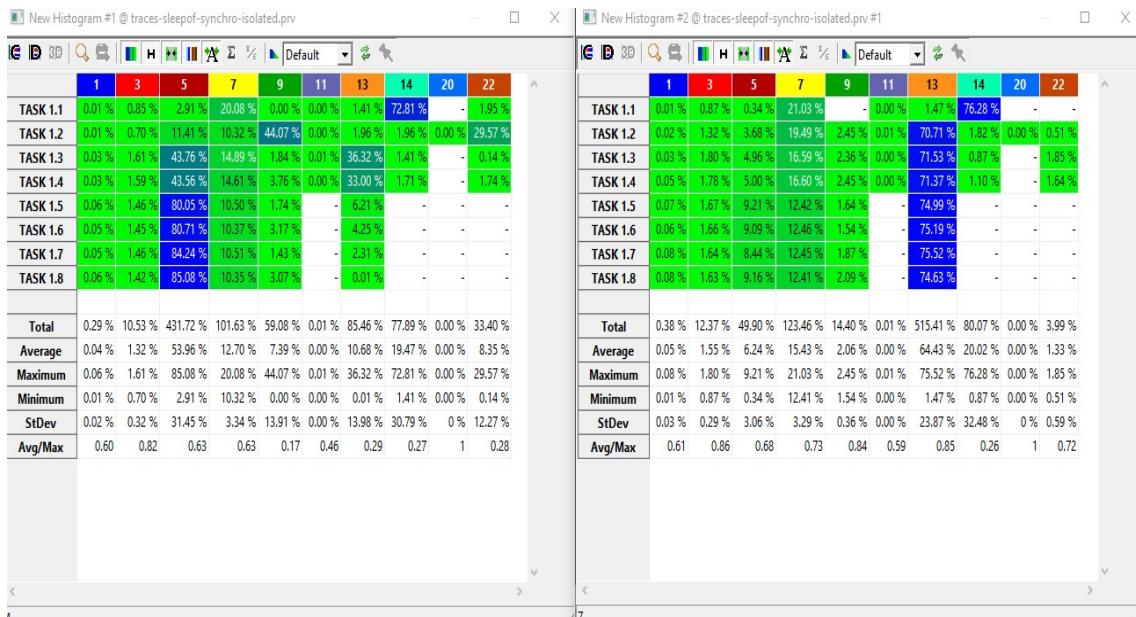


Figure 5.17 Statistics of the synchronous and asynchronous applications.

The histogram windows ease the comparison of both applications. By using statistics, more objective conclusions can be extracted:

- The percentage of computation time is quite similar for both applications.
- The percentage of time sending results (state 5) is very high in the synchronous application, however, the detaching time is less significant.
- The percentage of communications for results sending (state 5) is low in the asynchronous application, however, the detaching time is very significant.

The sleepOf application may be improved to get better communications and detaching times, however, the objective of this section is not to improve the applications under study but just showing the utility of the traceability tools and how to use them. For this reason, no more improvements and tests of the sleepOf application will be part of this project but may be of a future work.

5.4. Study of the Jacobi application

The Jacobi application is one of the first applications converted into a malleable application by using the DMR library. The application solves equation systems using a numeric method based on the parallel Jacobi algorithm, distributing the data and computation in multiple nodes. The application iterates until reaching a predefined error value.

Depending on the selected error value, the application can execute thousands or even millions of iterations, until reaching the stop condition. The malleable version of the Jacobi algorithm allows users to select the number of iterations to check for a reconfiguration of resources.

The following image shows the main part of the function compute(), the main computational operation executed for each rank.

```
183     dt = 0.0;
184     #pragma omp parallel for reduction(max:dt) private(j,incr)
185     for(i=1; i <= size; i++)
186     {
187         for(j=1; j <= COLUMNS; j++)
188         {
189             // using ternary operators improves the execution time a little
190             incr = data[i*(COLUMNS+2)+ j] - data_old[i*(COLUMNS+2)+ j] ;
191             incr = (incr > 0) ? incr : incr * (-1);
192             dt = (incr > dt) ? incr : dt;
193             data_old[i*(COLUMNS+2)+ j] = data[i*(COLUMNS+2)+ j] ;
194         }
195     }
196
197     // find global dt
198     MPI_Reduce(&dt, &dt_global, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
199     MPI_Bcast(&dt_global, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
200
201     return dt_global;
202 }
```

Figure 5.18. Main operations of the Jacobi algorithm.

The following image shows the compute() function wrapped by the DMR macros to make the algorithm malleable. The function DMR_Inhibit_iter() allows the application user to decide the number of iterations to execute between each reconfiguration process.

```

92     DMR_INIT(initialize_data(&Temperature,&Temperature_last, world_size, world_rank, rows), \
93             recv_expand(&Temperature, &Temperature_last, &rows, &it), \
94             recv_shrink(&Temperature, &Temperature_last, &rows, &it));
95
96     DMR_Inhibit_iter(100);           // DMR is only called every 100 cycles
97     DMR_Set_parameters(1, 8, 2, 2);
98     while(dt_global > MAX_TEMP_ERROR && DMR_it <= STEPS)
99     {
100         DMR_COMPUTE(dt_global = compute(Temperature, Temperature_last, world_size, world_rank, rows, &it));
101         DMR_it++;
102         DMR_RECONFIGURATION(send_expand(Temperature_last, rows, it), send_shrink(Temperature_last, rows, it));
103         if((DMR_it % 100) == 0 )
104         {
105             if(world_rank == world_size -1)
106             {
107                 track_progress(DMR_it, rows, Temperature);
108             }
109         }
110     }

```

Figure 5.19 Part of the Jacobi malleable algorithm.

The first step to studying the application is submitting a job. The job is submitted via a sbatch file:

- sbatch mnv_submission.sbatch

After the job execution, a CSV trace file is generated. It contains all the events created during the execution of the job. The CSV trace file is converted to a Paraver trace file by using the DMRTRACEPARSER application, to be loaded in Paraver. Figure 5-20 shows the thread and task views of the Jacobi application execution in Paraver.

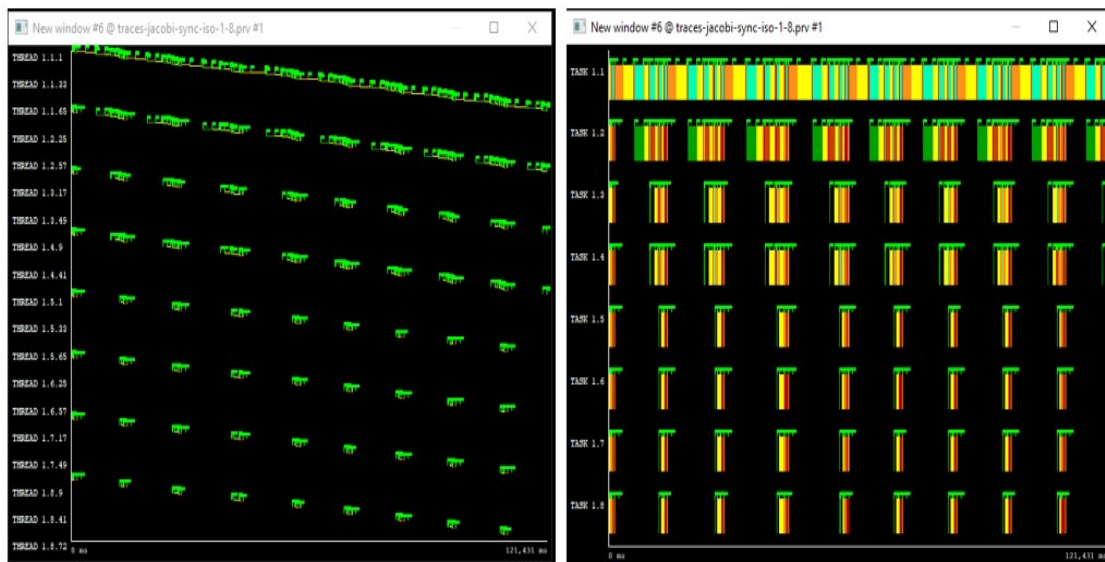


Figure 5.20 Thread and Task views of the Jacobi application.

The thread and task view can show a complete cycle of reconfigurations, from 1 to 8 resources. The comparison of both visualizations allows to decide on the more useful view.

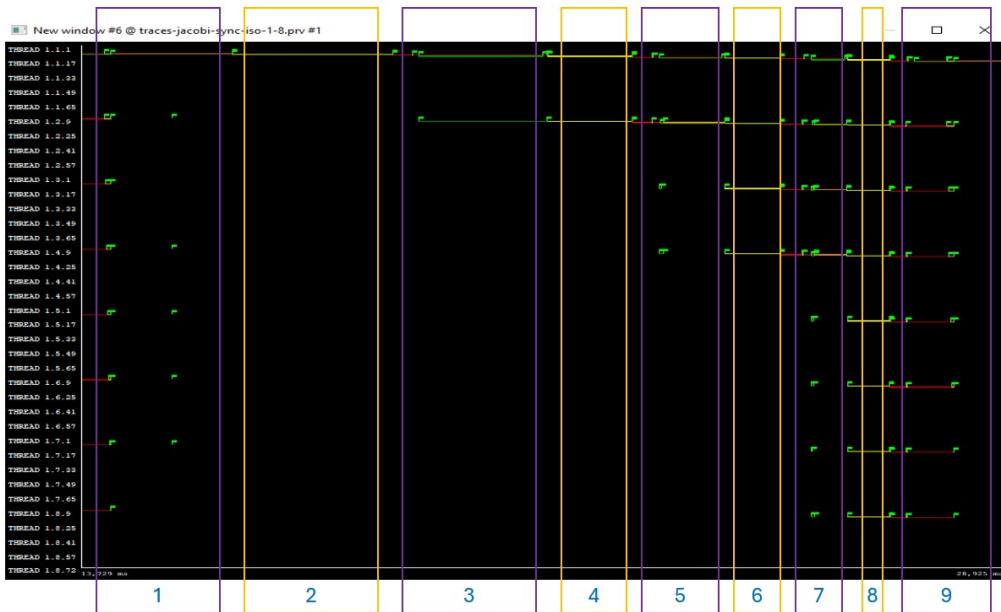


Figure 5.21 Reconfigurations cycle from 1 to 8 resources in the thread view.

Changing to the tasks view, the information is compacted at the task level. Paraver creates new states and colors to represent the processes executed in the same rank, at the same time, but in different threads.

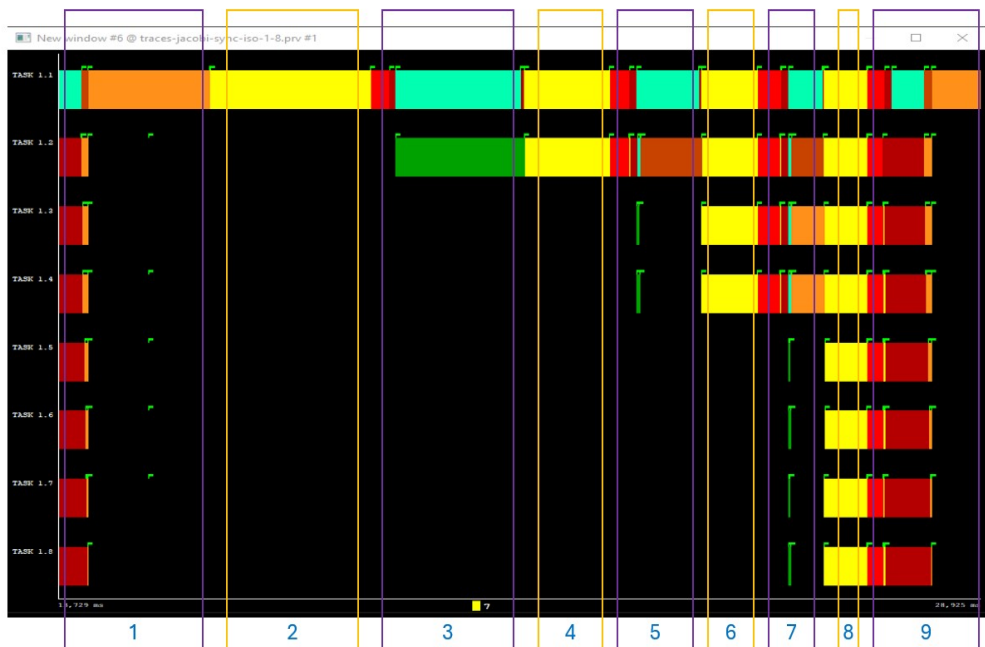


Figure 5.22. Reconfigurations cycle from 1 to 8 resources in task view.

After comparing the threads and the task view, it seems easier to use the task view, even is a compacted view with less information.

The complete cycle of reconfigurations from 1 to 8 resources contains several reconfigurations. Following the resource availability policy, the application expands its resources in powers of 2 from 1 to 8 resources. After a complete cycle, the application shrinks its resources to 1 to repeat the cycle. The has been divided into several sections to be explained.

1. Computation with 8 resources has finished, all the ranks send results to rank 0 and rank 0 sends data to compute to itself.
2. Rank 0 computes.
3. Reconfiguration, rank 0 sends results to itself and sends data to compute to ranks 0 and 1.
4. Ranks 0 and 1 compute.
5. Reconfiguration, ranks 0 and 1 send results to rank 0 and rank 0 sends data to compute to ranks 0 to 3.
6. Ranks 0 to 3 compute.
7. Reconfiguration,, ranks 0 to 3 send results to rank 0 and rank 0 sends data to compute to rank 0 to 7.
8. Ranks 0 to 7 compute.
9. Reconfiguration, ranks 0 to 7 send results to rank 0 and rank 0 sends data to compute to itself, starting a new cycle of reconfigurations.

In the task view, Paraver has created a new state, state 14, represented in light blue, which is the composition of the states sending and receiving processed simultaneously in the same rank but in different threads. The sending and receiving processes can be observed in detail in the thread and the task views, by applying object filtering and zooming over the visualization window.

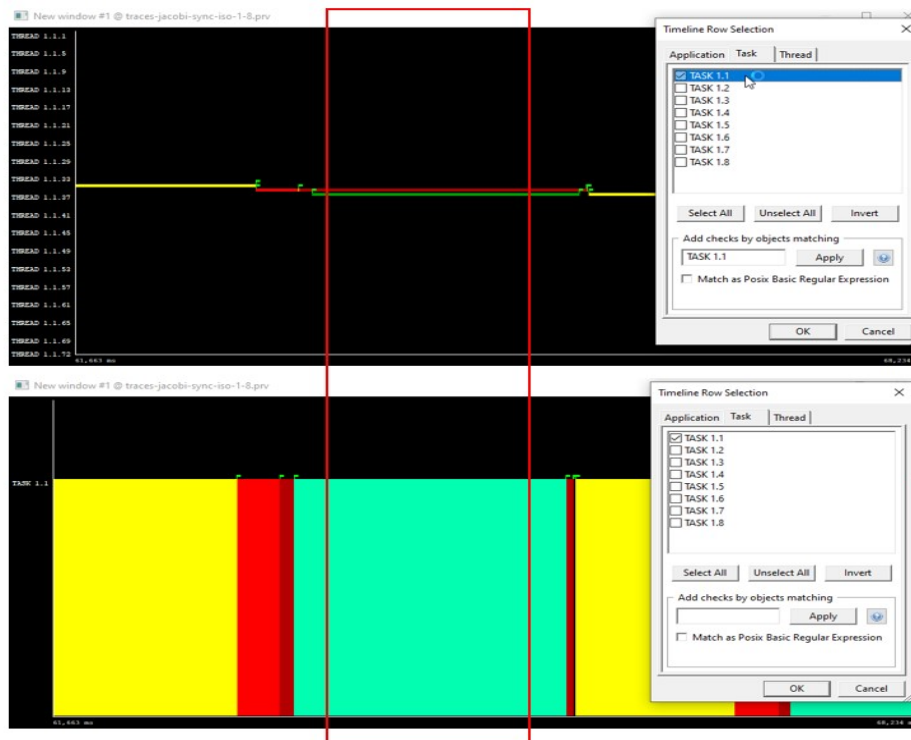


Figure 5.23. Simultaneous send and receive communications.

With a first visual analysis of a complete cycle of reconfigurations, some flags seem to be alone after detaching processes. Those flags seem not to have a relation with any state. The way to analyze those flags is to know their timestamps and review the trace file to know what events have generated them.

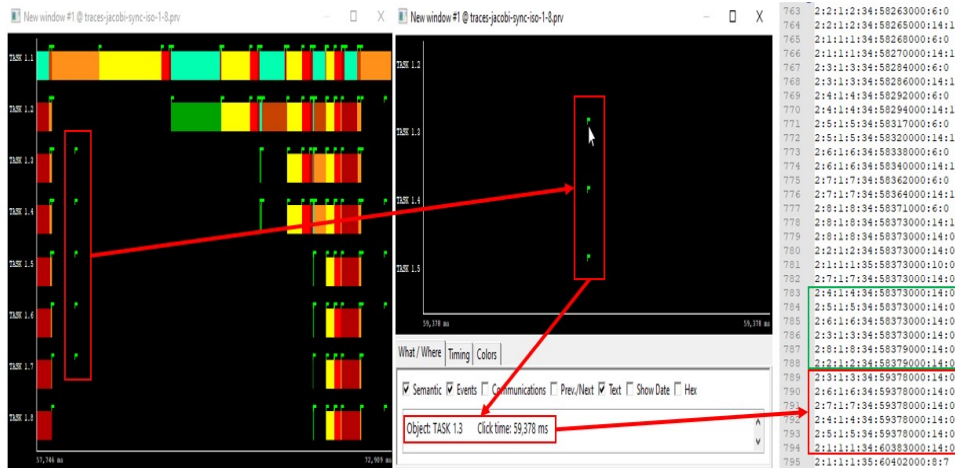


Figure 5.24 Analysis of events without state.

Locating the events in the trace file and analyzing them, was determined that the events 14 (end of detaching) were recorded two times in the trace file. Those events are recorded at several points in the function DMR_Detach().

```

3 void DMR_Detach(int action, int currNodes, char *managementHost, int dependentJobId, pthread_t socketThread)
4 {
5     dmrtrace_write_event(EVENT_DETACHING_START, DMR_comm_rank, DMR_comm_size, DMR_it);
6     int world_size, world_rank;
7     MPI_Comm_size(MPI_COMM_WORLD, &world_size);
8     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
9     char processor_name[MPI_MAX_PROCESSOR_NAME];
10    MPI_Get_processor_name(processor_name, &name_len);
11
12    MPI_Barrier(MPI_COMM_WORLD);
13
14    if (action > 0)
15    {
16        switch (action)
17        {
18            case 1:
19                dmrtrace_write_event(EVENT_DETACHING_END, DMR_comm_rank, DMR_comm_size, DMR_it);
20                exit(0);
21            case 2:
22                dmrtrace_write_event(EVENT_DETACHING_END, DMR_comm_rank, DMR_comm_size, DMR_it);
23            default:
24                break;
25        }
26    }
27 }

```

Figure 5.25 DMR_Detach function traceability.

After examining the source code, no explanation was found at the trace recording level, so that, the question was transferred to the DMR development team, who found and solved the problem in the DMR library.

Analyzing in detail the computing states, it was discovered that the computation times of the iteration were very short and in between of different iterations, there were some milliseconds not assigned to traceability states.



Figure 5.26 Indeterminate times between computations.

After analyzing the possible causes, it was determined that those times corresponded to the trace recording times in the macro DMR_COMPUTE().

```

48 #define DMR_COMPUTE(compute) \
49 { \
50     dmrtrace_write_event(EVENT_ITERATION_START, DMR_comm_rank, DMR_comm_size, DMR_it); \
51     compute; \
52     dmrtrace_write_event(EVENT_ITERATION_END, DMR_comm_rank, DMR_comm_size, DMR_it); \
53 }

```

Figure 5.27 Macro DMR_COMPUTE overloaded by trace recording.

Those times were just some milliseconds, but in algorithms with very short computation times, and a big number of iterations, the recording times represent a big overhead. For this reason, it was decided to record the ITERATION_START and ITERATION_END events not in each iteration but in the ones where a reconfiguration is performed.

Another strange behaviour observed during the visual analysis of a complete reconfiguration cycle of Jacobi was the incoherence between the data transfer times between the ranks 0 and 1 in comparison with the rest of the ranks. This behaviour was also found in the study of the sleepOf application.



Figure 5.28 Unbalanced data transfer times in Jacobi.

As was done in the sleepOf application analysis, the source code of the Jacobi application has been reviewed. A possible improvement has been identified in the functions recv_shrink() and send_expand(). The improvement consists of changing the synchronous communications to asynchronous communications and trying to reduce waiting times to perform data transfers in parallel.

```

330 void send_expand(double *data_old, int size, int iterator)
331 {
332     int world_rank, comm_size, intercomm_size, factor, dst, inIPart;
333     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
334     MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
335     MPI_Comm_remote_size(MPI_COMM_WORLD, &intercomm_size);
336
337     factor = intercomm_size / comm_size;
338     size = GLOBAL_ROWS / intercomm_size;
339
340     for (int i = 0; i < factor; i++)
341     {
342         dst = world_rank * factor + i;
343         inIPart = size * (COLUMN+2) * i;
344         if (dst == intercomm_size - 1)
345             size += GLOBAL_ROWS % intercomm_size;
346
347         MPI_Ssend(data_old + inIPart, (size+2)*(COLUMN+2), MPI_DOUBLE, dst, 0, MPI_INTERCOMM);
348         MPI_Ssend(iterator, 1, MPI_INT, dst, 0, MPI_INTERCOMM);
349     }
350 }
351
352
353
354
355
356
357 void send_expand(double *data_old, int size, int iterator)
358 {
359     int world_rank, comm_size, intercomm_size, factor, dst, inIPart;
360     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
361     MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
362     MPI_Comm_remote_size(MPI_COMM_WORLD, &intercomm_size);
363
364     factor = intercomm_size / comm_size;
365     size = GLOBAL_ROWS / intercomm_size;
366
367     MPI_Request request[10];
368     for (int i = 0; i < factor; i++)
369     {
370         dst = world_rank * factor + i;
371         inIPart = size * (COLUMN+2) * i;
372         if (dst == intercomm_size - 1)
373             size += GLOBAL_ROWS % intercomm_size;
374
375         MPI_Isend(data_old + inIPart, (size+2)*(COLUMN+2), MPI_DOUBLE, dst, 0, MPI_INTERCOMM, &request[i]);
376         MPI_Ssend(iterator, 1, MPI_INT, dst, 0, MPI_INTERCOMM);
377     }
378     for (int i = 0; i < factor; i++)
379     {
380         MPI_Wait(&request[i], MPI_STATUS_IGNORE);
381     }
382 }

```

Figure 5.29 Parallelization of the function send_expand() in Jacobi.

Communication times can be easily compared by visual analysis.

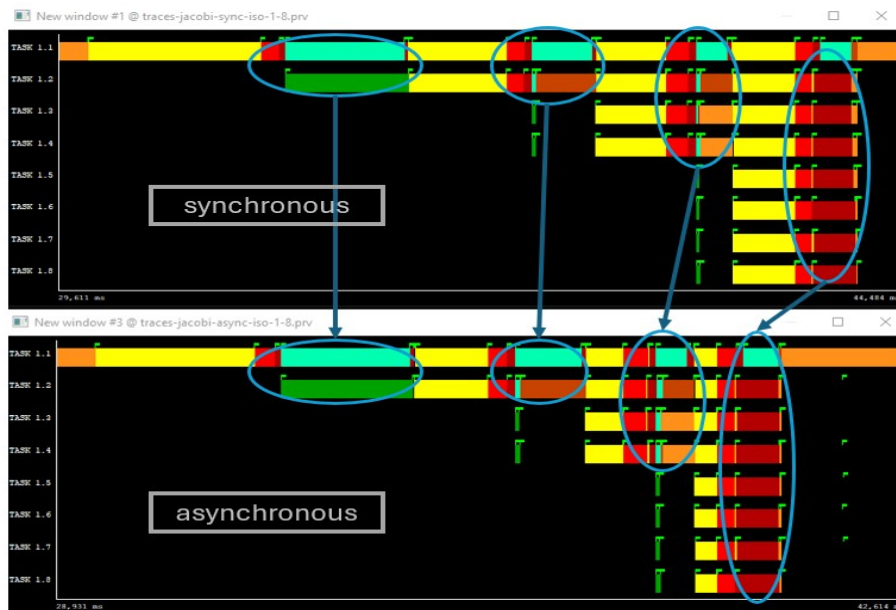


Figure 5.30 Synchronous and asynchronous data transfers.

The visual comparison does not show any improvement in the data transfers with the parallelization, but it seems that it has affected in some way the computing times.

The computing times can be analyzed in more detail by the visualization windows. Double-clicking over the computation states, of each trace, the computation times can be obtained.

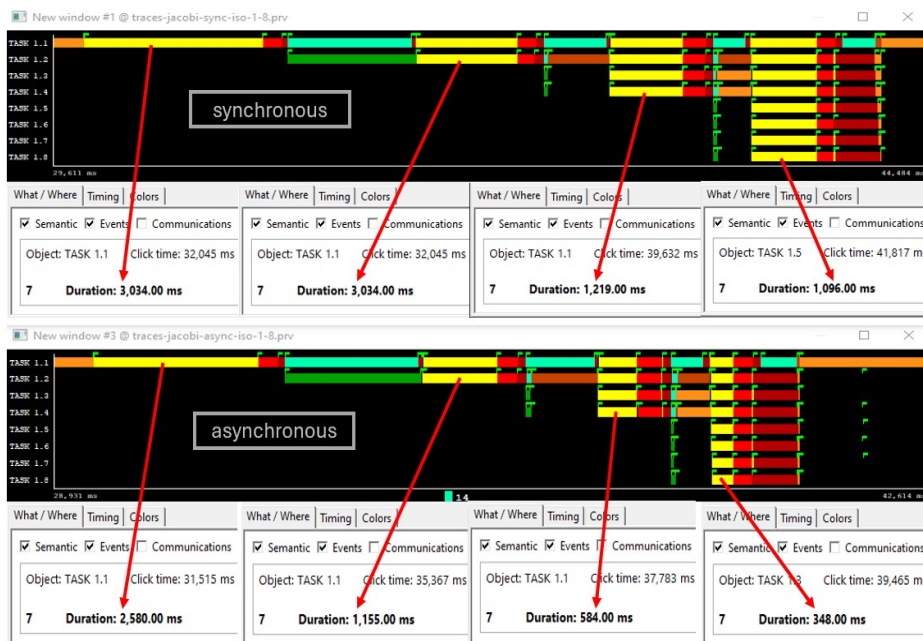


Figure 5.31. Synchronous versus asynchronous computing times.

The visual comparison shows improvements in the computing times when parallel communications are used. The statistic view can be used to obtain more detailed information.

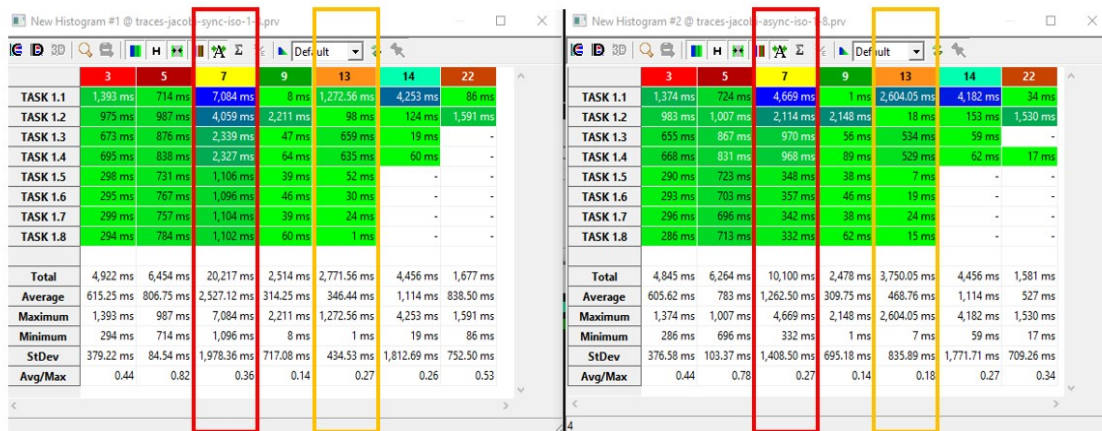


Figure 5.32 Synchronous versus asynchronous computing statistics.

The statistics view shows a clear reduction of the computation times (event 7) but also a time increment in the detaching times (13). The rest of the states in both applications are very similar without notable changes. There are no apparent reasons for those results, the parallel data transfers seem not to have a relation with the computation times. It is not possible to explain those results without reviewing in deep the Jacobi algorithm and its traceability. As a deep study of the Jacobi application is out of the scope of this project, it may be done in future works.

The last study of the Jacobi application consists of analyzing the behaviour of the execution of multiple instances simultaneously sharing and resources. To execute multiple instances at the same time, is necessary to change the resource management policy, enabling the PhD policy in the file “slurm-linear.c”, disabling the rest of the policies and recompiling the DMR library.

```

4084 /*****
4085 /*****
4086 /*****          TALP          *****/
4087 /*****
4088 /*****
4089 > /# ...
4127 /*****
4128 /*****
4129 /*****          RANDOM          *****/
4130 /*****
4131 /*****
4132 > /# ...
4174 /*****
4175 /*****
4176 /*****
4177 /*****          expand step by step up to 8, and then shrink to 1          *****/
4178 /*****
4179 /*****
4180 > /# ...
4204 /*****
4205 /*****
4206 /*****
4207 /*****          PhD          *****/
4208 /*****
4209 /*****
4210 /*****
4211 extern int select_p_select_jobinfo_get(select_jobinfo_t *jobinfo,
4212     enum select_jobdata_type data_type, void *data) {
4213 >     if (data_type == SELECT_JOBDATA_INFO) {...
4222 >     } else if (data_type == SELECT_JOBDATA_ACTION) {...
4471     return SLURM_ERROR;
4472 }

```

Figure 5.33 Resources policy for multiple instance execution.

It is also necessary to define how to share the resources, it is done via the DMR macro DMR_Set_parameters(), which allows to define how to share resources between different instances. Figure 5.33 shows the configuration for the test.

```

99 | // min processors, max processors, without function, powers of 2
100 | DMR_Set_parameters(1, 8, 2, 2);

```

Figure 5.34 Resource sharing configuration via DMR macro.

Multiple instances in parallel can be executed by including a loop in the sbatch file “submission.sbatch”, for example, they can be executed 4 instances.

```

110 | for i in $(seq 1 4);
111 | do
112 |     $SLURM_BIN/sbatch -JdmrJob1 -N1-8 launch.sh &
113 |     sleep 1
114 | done

```

Figure 5.35 Execution of multiple Jacobi instances in parallel.

After the changes, the job can be submitted via the previously used sbatch file.

- sbatch mnv_submission.sbatch

The execution of multiple instances in parallel in the same job generates multiple trace files, in this case 4 trace files.

```

28/07/2024 13:18 31.816 dmrtrace-2.csv
28/07/2024 13:18 29.690 dmrtrace-3.csv
28/07/2024 13:18 33.708 dmrtrace-4.csv
28/07/2024 13:18 35.146 dmrtrace-5.csv
4 archivos 130.360 bytes

```

Figure 5.36 Trace files of 4 instances in parallel.

DMRTRACEPARSER can merge multiple trace files to generate a unique Paraver trace file, in several ways, for instance, passing the files as arguments.

- dmrtraceparser.exe dmrtrace-2.csv dmrtrace-3.csv dmrtrace-4.csv dmrtrace-5.csv

After the Paraver trace file is generated, it can be opened in Paraver. Figure 5.35 shows the thread and task view of the multiple instances.

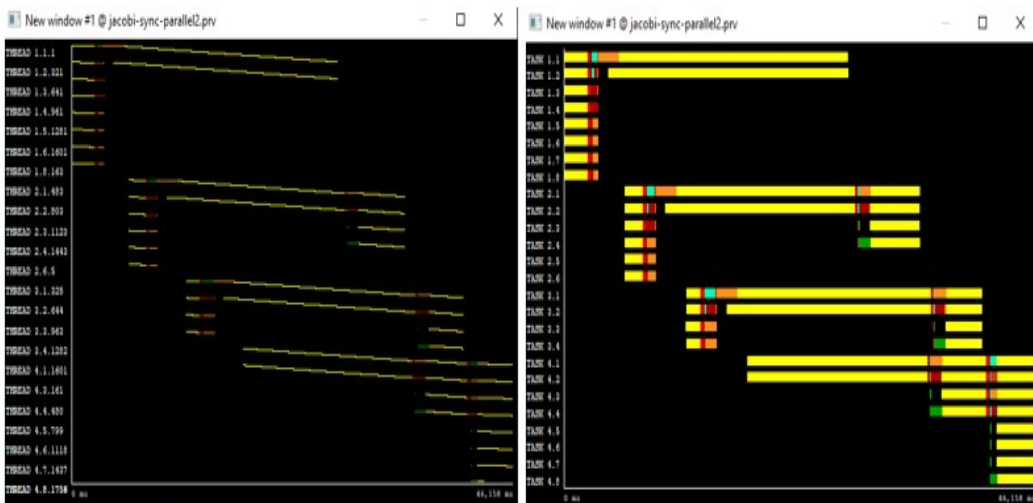


Figure 5.37 Thread and task view of execution of multiple instances of Jacobi.

The best view to analyze the execution of the application is the task view. The view can be divided into sections for the execution analysis.

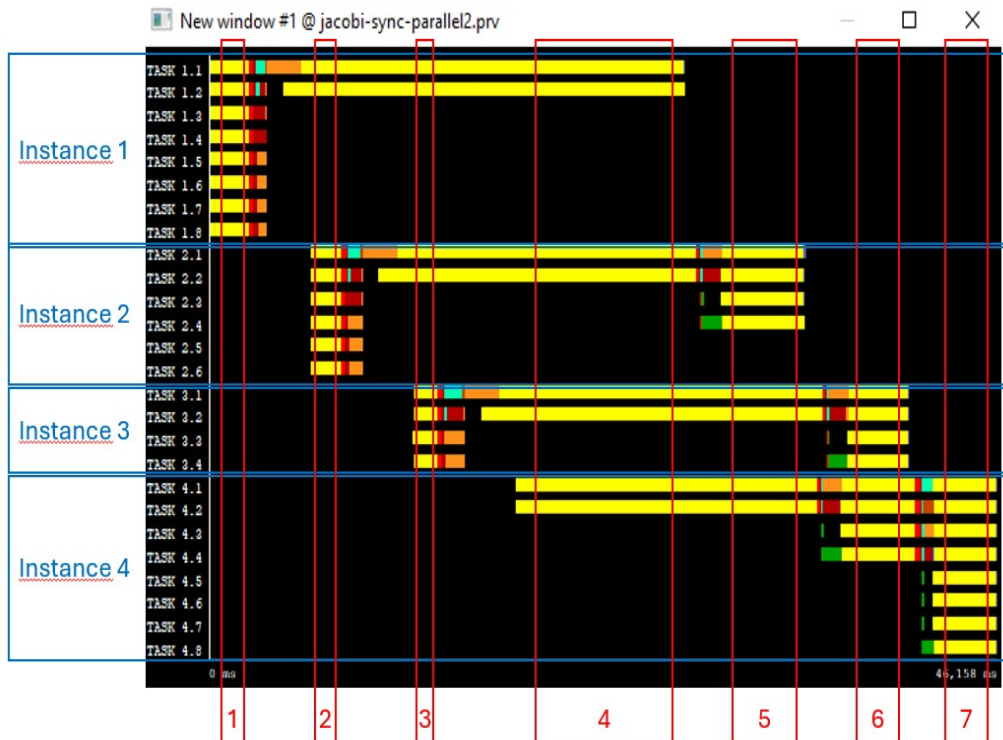


Figure 5.38. Job execution divided into sections.

The different sections in Figure 5.36 show 4 instances running in parallel and sharing 8 resources. The explanation for each section is the following:

1. Instance 1 starts with 8 resources.
2. After the first reconfiguration, instance 1 shrinks to 2 resources and instance 2 starts with 6 resources.
3. After the second reconfiguration, instance 2 shrinks to 2 resources and instance 3 starts with 4 resources.
4. After the third reconfiguration, instance 3 shrinks to 2 resources and instance 4 starts with 6 resources.
5. When instance 1 finishes, instance 2 expands to 4 resources.
6. When the instance 2 finishes, the instances 3 and 4 expand to 4 resources.
7. When instance 3 finishes, instance 4 expands to 8 resources and computes until finishing.

The explained sequence meets with the expected behaviour and the results obtained allow understanding and validation of how DMR manages sharing resources between different instances.

Finally, it is possible to view the statistics of the execution of each instance.

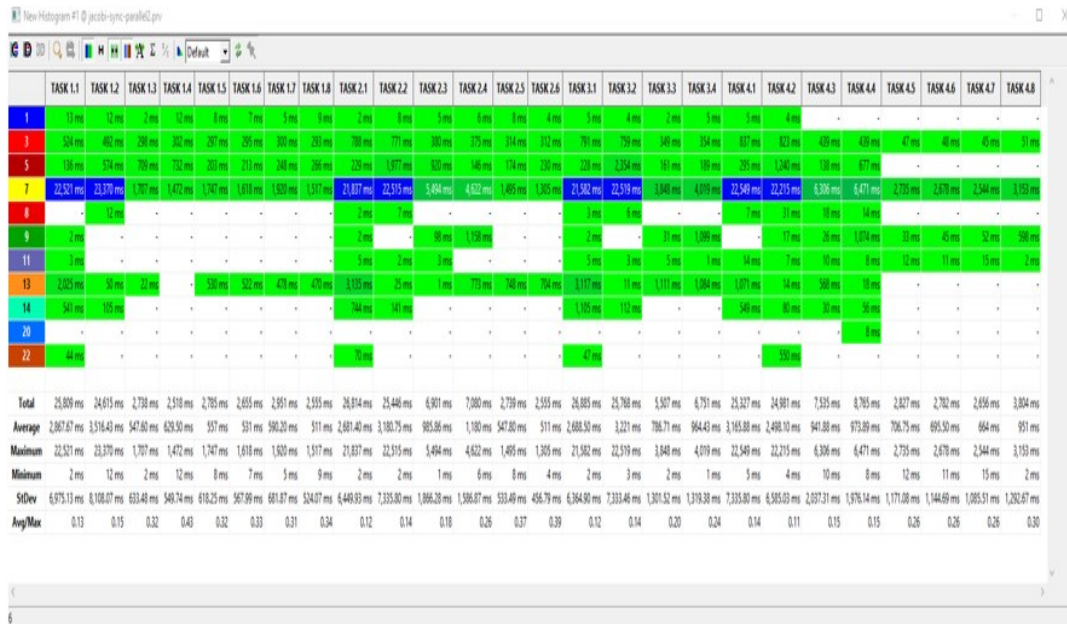


Figure 5.39. Statistics of simultaneous execution of multiple instances.

5.5. Summary of the study of several malleable applications

In this section, traceability tools for malleable applications have been used to study several malleable applications. Common procedures have been defined, and traceability tools have been applied to study several malleable applications.

Visual and statistical study techniques have been applied to analyze the traces of the applications under study. Several issues have been identified and some improvements have been tested. Some of the tested improvements have obtained successful results, but implementing other improvements will require a deeper study of the application source code.

6. Conclusions and future work

When this project was defined, three main objectives were defined:

- Reviewing technical concepts regarding high-performance computing and knowing the BSC's malleability framework.
- Developing a library to provide the BSC's malleability framework with traceability capabilities.
- Developing a software tool for trace analysis.

The project workload was distributed in a percentage relation of about 30% of research in the HPC, especially in the dynamic resources management area, and about 60% of software engineering, especially in visual traceability analysis tools. It was mainly a software and data analysis engineering project.

Due to the project being developed within the framework of collaboration with the BSC, the project requirements have been evolving during the development process, always with the aim of creating a useful deliverable work.

The first part of the project was dedicated to the research activities needed to acquire the knowledge necessary for the trace library development. It was knowledge about HPC, malleability, programming methodologies, tools, etc.

After the development of the trace library and after starting the development of the trace analysis tools, when the first traces were analyzed, and when the real potential of the application was known, it was decided to change the course of the project to get a more useful deliverable for the BSC.

It was decided to abandon the development of the software for trace analysis and focus on developing a tool to be able to convert the traces into the format of the standard HPC analysis tool of the BSC, the Paraver application, and use Paraver to study several malleable applications already available at the BSC.

After that decision to change the project requirements, the project workload was changed notably. The last part of the project changed from the development of a visual analysis software application to a format conversion software utility.

The new project specifications required growing up considerably in the research activity. It was necessary to investigate how to use Paraver, how the Paraver traces are defined, how to convert malleable traces to Paraver traces, and, for the application analysis, how those malleable applications work.

After finishing the project, the main activities developed have been the following ones:

- Reviewing technical concepts regarding high-performance computing and knowing the BSC's malleability framework.

- Developing a library to provide the BSC's malleability framework with traceability capabilities.
- Developing a software tool for converting malleability trace files to Paraver format.
- Study the traces of the execution of several malleable applications.
- Project management and documentation.

The total time dedicated to the project has been about 800 hours, and the workload distribution is more or less the shown in Table 7.

Technical concepts	%	Type
Basics HPC (clusters, slurm, OpenMP, MPI)	5%	Research
Basics BSC (MareNostrum, Confluence, GitLab)	5%	Research
Basics Malleability and DMR	5%	Research

Trace recording library development	%	Type
Development procedures and tools	5%	Research
Trace recording library development and test	10%	Development

Trace converter development	%	Type
Paraver functionalities	5%	Research
Paraver trace files	5%	Research
Trace format converter development	15%	Development

Application analysis	%	Type
Application study general procedures definition	5%	Research
SleepOf application understanding and analysis	5%	Research
Jacobi application understanding and analysis	10%	Research

Summary of management activities	%	Type
Documentation	20%	Management
Project management	5%	Management

Table 7. Workload distribution of the project.

The workload distribution shows that even though this project started as a software engineering project, it finally became a research project with a dedication of about 50% research, 25% development and 25% management and documentation.

The main goals of the project have been reached. At the final of the project, a trace generation library, a trace converter utility, and some trace analysis procedures and examples are available, allowing to trace the execution of malleable applications.

Even though the malleability traceability tools are already functional, there is future work to do in this area. Malleability is an important technology for the improvement of the performance of HPC infrastructures, and it is an active

project at BSC. So far, only some algorithms have been updated and tested to work in a malleable way, and a lot of algorithms are still pending on being malleable. It will be important future work on the malleability project and consequently will imply future work on the traceability project.

Some future improvements of the malleability traceability could be the following:

- Recording more types of events.
- Tracing communications between ranks.
- Traceability at thread level on the user side.
- Mixing malleability traceability with HPC traceability.
- Real-time traceability monitoring.
- Alternative approaches to show iterations in Paraver.
- Direct generation of PRV files during the job execution.
- Configuration files for different types of analysis by using Paraver.
- Automatic analysis and reporting generation.
- Definition of procedures for analysis and validation of algorithms.

As it has been explained the malleability traceability can be an important complementary project for the malleability project at the BSC. Traceability will help to analyse and validate the malleability project results, easing continuous improvement.

7. Glossary

BSC. Barcelona Supercomputing Center.
CNS. Centro Nacional de Supercomputación.
COMPILATION. Process to create an executable from source code.
CSV. Comma Separated Values, plain text file format.
C/C++. High-performance multi-processor and HPC programming language.
DMR. Dynamic Management Resources, malleability library of the BSC.
DMRTRACE. Malleability trace library.
DMRTRACEPARSER. Program to convert CSV to PRV files.
EVENT. Action that occurs during program execution.
GIT. Distributed version control system.
GITLAB. Web-based GIT repository.
HPC. High Performance Computing.
JACOBI. An iterative method to solve systems of equations.
JOB. Unit of work to submit to a cluster for its execution.
JUPYTER NOTEBOOK. Python interactive computing environment.
LINK. Combination of compiled files in a unique executable or library.
MAKEFILE. The file contains instructions to compile and link.
MALLEABILITY. Ability to adjust resources for job executions in HPC.
MARENOSTRUM5. Name of the HPC cluster of the BSC.
MPI. Message passing interface, used for the communication process in HPC.
NANOS++. A process control runtime developed by the BSC.
OPENMP. Open Multi-Processing API for shared memory multiprocessing.
PARAVER. HPC application execution analysis tool developed by the BSC.
PYTHON. High-level multi-processor and scientific programming language.
PRV. Paraver trace file extension.
SLEEPOF. Programming for testing malleability developed by the BSC.
SLURM. Job scheduler for HPC environments.
QUEUE. List of HPC jobs waiting to be processed.
STATE. Status of execution of a job in a concrete time.
TRACE. A set of job execution events is recorded in a file.
VSCODE. Multi-language source code editor based on plugins.

8. Bibliography

- [1] https://upcommons.upc.edu/bitstream/handle/2117/329704/DMRlib_TC_revision.pdf
- [2] <https://www.sciencedirect.com/science/article/abs/pii/S0167819118302229?via%3Dihub>
- [3] <https://tools.bsc.es/doc/1370.pdf>