Universidade Estadual de Campinas
Instituto de Computação

Vitoria Dias Moreira Pinho

Parallel Code Profiling for the OpenMP Cluster Distributed Task-based Runtime

Perfilamento de Código Paralelo para o Runtime Distribuído de Tarefas OpenMP Cluster

CAMPINAS
2023

Vitoria Dias Moreira Pinho

Parallel Code Profiling for the OpenMP Cluster Distributed Task-based Runtime

Perfilamento de Código Paralelo para o Runtime Distribuído de Tarefas OpenMP Cluster

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestra em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Hervé Cédric Yviquel
Co-supervisor/Coorientador: Prof. Dr. Guido Costa Souza de Araújo

Este trabalho corresponde à versão final da Dissertação defendida por Vitoria Dias Moreira Pinho e orientada pelo Prof. Dr. Hervé Cédric Yviquel.

CAMPINAS
2023

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

P655p
Pinho, Vitoria Dias Moreira, 1997-
Parallel code profiling for the OpenMP Cluster distributed task-based runtime / Vitoria Dias Moreira Pinho. – Campinas, SP : [s.n.], 2023.

Orientador: Hervé Cédric Yviquel.
Coorientador: Guido Costa Souza de Araújo.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Programação paralela (Computação). 2. OpenMP (Programação paralela). 3. Computação de alto desempenho. I. Yviquel, Hervé Cédric. II. Araújo, Guido Costa Souza de, 1962-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações Complementares

Título em outro idioma: Perfilamento de código paralelo para o runtime distribuído de tarefas OpenMP Cluster
Palavras-chave em inglês:
Parallel programming (Computer science)
OpenMP (Parallel programming)
High performance computing
Área de concentração: Ciência da Computação
Titulação: Mestra em Ciência da Computação
Banca examinadora:
Hervé Cédric Yviquel [Orientador]
Carla Osthoff Ferreira de Barros
Luiz Fernando Bittencourt
Data de defesa: 05-05-2023
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0001-9619-3713
- Currículo Lattes do autor: http://lattes.cnpq.br/5243670807170633

Universidade Estadual de Campinas
Instituto de Computação

Vitoria Dias Moreira Pinho

Parallel Code Profiling for the OpenMP Cluster Distributed Task-based Runtime

Perfilamento de Código Paralelo para o Runtime Distribuído de Tarefas OpenMP Cluster

Banca Examinadora:

- Prof. Dr. Hervé Cédric Yviquel
  Instituto de Computação - Universidade Estadual de Campinas
- Prof. Dr. Luiz Fernando Bittencurt
  Instituto de Computação - Universidade Estadual de Campinas
- Dra. Carla Osthoff Ferreira de Barros
  Computação de Alto Desempenho - Laboratório Nacional de Computação Científica

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 05 de maio de 2023

Acknowledgements

This work was possible thanks to the help of several people. I especially thank my mother, family, and friends for all their support and encouragement. I am grateful to my advisor, Professor Dr. Hervé Yviquel for his guidance, and all the help in the development of the research and the dissertation. I have learned a lot from him and his mentorship. Thanks to my co-advisor, Professor Guido Araujo for all the support and monitoring of the project. I would also like to thank the Researcher Dr. Marcio Machado Pereira for his valuable support in the project and for revising the dissertation. Thanks to the entire OMPC project team, especially Carla Cardoso, who developed the applications that I used during the project and provided me with all the necessary support for me to use them. Also, to Gustavo Leite who helped with the code review and to Guilherme Valarini who answered any questions I had regarding the OMPC profile. I am grateful to Petrobras and FUNCAMP for their financial support that has enabled me to complete this research. I would like to acknowledge my professors, who have taught me throughout my academic journey, and the Institute of Computing at UNICAMP, for providing me with the necessary resources to carry out my research. I would also like to express my gratitude to the examining board of my Qualification Exam, composed of Professor Dr. Luiz Fernando Bittencourt and Professor Dr. Lucas Francisco Wanner, for all the valuable advices. And finally, I would like to thank the examining board of my master's degree for accepting participation.

Resumo

O desenvolvimento de programas paralelos é amplamente utilizado em aplicações científicas, desde que esse tipo de aplicação demanda um processamento de alto desempenho. OMPC é um runtime distribuído baseado em tarefas que permite o uso de um modelo de programação de tarefas OpenMP para paralelizar o código. Ao contrário das tarefas clássicas do OpenMP, o OMPC distribui a computação entre computadores heterogêneos, permitindo explorar ambos os níveis de paralelismo na thread e no processador, lidando com as comunicações MPI por conta própria. O desenvolvimento do código é mais fácil, já que o usuário não precisa programar MPI. Pode ser desafiador depurar e entender o código paralelo, pois a execução do código ocorre ao mesmo tempo em diferentes núcleos do processador ou até mesmo diferentes computadores. As ferramentas de perfilamento de código ajudam os usuários com essas questões, uma vez que fornecem informações sobre gerenciamento de memória, transferências de dados e dos eventos executados. O usuário pode usar os resultados de perfilamento para realizar possíveis melhorias de desempenho ou detectar erros em aplicações paralelas. Entretanto, a maioria dessas ferramentas não é projetada para atender a aplicativos baseados em tarefas, e os usuários destes runtimes, como o OMPC, enfrentam mais dificuldades. A proposta deste trabalho foi desenvolver uma ferramenta de perfilamento para atender às necessidades de runtimes baseados em tarefas. Este estudo examinou quais métricas e funcionalidades já eram oferecidas pelas ferramentas existentes e quais poderiam ser utilizadas e aprimoradas para aplicativos baseados em tarefas distribuídas.

Abstract

Parallel program development is widely used in scientific applications since this type of application demands high-performance processing. OMPC is a task-based distributed runtime that permits using an OpenMP task programming model to parallelize the code. Unlike OpenMP classic tasks, OMPC distributes computation across heterogeneous computers, permitting explore both parallelism levels at thread and processor, handling MPI communications on its own. Code development is easier since the user does not need to program MPI. Debugging and understanding parallel code can be challenging since the execution of the code occurs at the same time in different cores or even computers. Profiling tools help users address this as provide information about memory management, data transfers, and executed events. The user can use the profile results to perform possible parallel performance improvements or detect application errors. However, most profiling tools are not designed to suit task-based applications, and users of runtimes such as OMPC face more difficulties. The purpose of this work was to develop a profile tool to attend to the needs of task-based runtimes. This study examined which metrics and features were already offered by the existing profiling tools and which could be utilized and improved for distributed task-based applications.

List of Figures

2.1 Task Graph Example. 17
2.2 Critical Path Example. 18
2.3 OMPC Overview. 21

3.1 Profiling Models. 23
3.2 Profiling Workflow. 23
3.3 Call-graph and Flat Profile. 24

4.1 Steps to generate profile files. 32
4.2 OMPC Bench usage 33
4.3 Visualization Tool components 33
4.4 Developer timeline version 36
4.5 User timeline version 36
4.6 OMPC Events Timeline 39
4.7 OMPC Data Events 39
4.8 Arrows of user version timeline. 41
4.9 Profile symbol usage. 42
4.10 Simple Task Graph. 42
4.11 Scheduling Task Grap. 43
4.12 Computation and Parallel Time. 45
4.13 Communication Time. 45
4.14 Graph Metric Example. 47

5.1 Timeline SGEMM - All Events. 50
5.2 Timeline SGEMM - First and Last Events. 51
5.3 Timeline SGEMM - Zoomed. 51
5.4 Timeline SGEMM - Dependencies. 52
5.5 Timeline SGEMM - Head Process Begin. 53
5.6 Timeline SGEMM - Head Process Execution. 54
5.7 Timeline SGEMM - Head Process End. 54
5.8 Random Spread Graph. 61
5.9 Random Nearest Graph. 61

List of Tables

3.1 Profiling Tools Overall Information. 27
3.2 Profiling GUI Tools. 27
3.3 Profile Tools Metrics 28
3.4 Profile Tools Task Analysis. 29

5.1 SGEMM 32k - Time and Efficiency Metrics 50
5.2 SGEMM - Quantitative and Time Metrics 55
5.3 SGEMM - Efficiency Metrics 55
5.4 SPOTRF - Quantitative and Time Metrics 56
5.5 SPOTRF - Efficiency Metrics 56
5.6 SPOTRF EXP 1 - Task Metrics 58
5.7 SPOTRF EXP 2 - Task Metrics 58
5.8 Graph Metric - Random Nearest. 59
5.9 Graph Metric - Random Spread. 60

Contents

1 Introduction **12**
1.1 Motivation . . . 14
1.2 Objectives . . . 14
1.3 Thesis Structure . . . 14

2 Parallel programming models **15**
2.1 Parallelism Models . . . 15
2.1.1 Loop parallelism . . . 15
2.1.2 Task parallelism . . . 16
2.2 Implementation of Parallelism Models . . . 18
2.2.1 OMPC . . . 19

3 Profiling **22**
3.1 MPI+X Profiling . . . 24
3.2 Task Profiling . . . 26
3.3 Comparative analysis of profiling tools . . . 26

4 OMPC Profiling tools **30**
4.1 OMPC Profile Workflow . . . 31
4.2 Visualization Tool Overview . . . 32
4.3 OMPC Timelines . . . 35
4.3.1 OMPC Threads . . . 37
4.3.2 OMPC Events . . . 37
4.3.3 OMPC Arguments . . . 38
4.3.4 OMPC Dependencies . . . 40
4.3.5 Profile Symbols . . . 40
4.4 Task Graph . . . 41
4.5 Profile metrics . . . 43
4.5.1 OMPC . . . 43
4.5.2 Task . . . 45
4.5.3 Graph . . . 46

5 Experiments **48**
5.1 Experimental setup . . . 48
5.2 OMPC Plasma - Timeline Analysis . . . 49
5.3 OMPC Plasma - General Metrics . . . 53
5.4 OMPC Plasma - Task Metrics . . . 57
5.5 OMPC Task Bench - Graph Metrics . . . 59

6 Conclusion 62
6.1 Lessons learned 63
6.2 Future works 63

Bibliography 65

Chapter 1

Introduction

The development of parallel programs has increased significantly due to the high processing demand of scientific applications. These applications perform massive computations on large amounts of data and need efficient parallelization to execute in a timely manner. Compute is distributed across multiple cores and computers, using heterogeneous hardware and devices to accelerate processing. There are several parallelization paradigms with two widely used libraries to address it: OpenMP [14] and MPI [17], compatible with C, C++, and Fortran languages. The Message Passing Interface (MPI) defines the syntax and semantics of library routines used to write programs that can communicate by message-passing. It permits exploring the parallelism between processes through defined routines. MPI is extensively used for programming distribution in computer clusters, employing message passing to transfer data and computation among nodes. On the other hand, OpenMP is an API for shared-memory parallel programming and allows distributing computation across threads. Despite these two libraries being designed for different proposals, it is common to use a hybrid MPI+OpenMP implementation to explore both levels of parallelism (by threads or processes).

Loop-based parallelism is a common parallelization approach that consists of dividing loop iterations among computer threads. Although it is usually easier to program, data dependencies across iterations can make it difficult to implement. In this case, a more suitable approach is task parallelism since it permits defining pieces of code to be executed in parallel, called tasks, with data dependencies between them. Regardless of the limitation of being designed to work only at the thread level, OpenMP allows the development of both methods of parallelization through a work-sharing loop construct or a task construct. The work-sharing loop construct divides the iterations of an associated loop across threads. Alternatively, the task construct distributed pieces of code defined by a structured block among the computer threads and presents the possibility of defining data dependencies between them. Despite providing a more flexible way to parallelize the code, task construct adds more complexity to writing the code. OpenMP provides pragmas[1] to perform parallel operations in C and C++ programs. In particular, programmers can describe the dependencies between tasks, permitting the use of more complex parallelism patterns.

---

[1] Preprocessor instructions, also known as compiler directives.

Indeed, a hybrid MPI+OpenMP implementation allows a full parallel exploration through thread and process levels, but it increases the complexity of code development. In such applications, users need to beware of communication between computers, choose the best workload division, often with different hardware, and face problems with data access and network. Furthermore, the design of a parallel system must attempt some problems with data management, synchronization, load balancing, and race conditions, among others. OpenMP Cluster [35] (OMPC) is a distributed task-based runtime developed to address these issues. It permits the implementation of parallel programs with computation distributed via OpenMP task directives. However, this runtime distributes tasks across the computer nodes of clusters, not just in a single computer, like the classic OpenMP task directive. Despite the programmer using only OpenMP to develop the code, the OMPC runtime automatically handles the load-balancing and the communication between the cluster nodes through MPI.

Debugging and improving parallel codes is difficult since each piece of code running in parallel produces its own results and can share variables or memory with another process or thread. Profiling tools provide a better understanding of code behavior, facilitating the implementation of these programs and helping to debug the code, evaluate the performance, detect bottlenecks and overheads, and others. These tools can provide a time and memory analysis. Time analysis shows how much time is spent on each function of the code or in memory transfers, how long the processes or threads were idled (without doing computations while waiting to receive data, for example), and so on. As a complement, memory analysis can provide the size of transferred messages, the total number of times the user accesses such data or memory region, the percentage of cache hits and fails, and others. Also, the profiling tool can present the results in several ways: a table with profiling statistics, and a timeline with profiling events[2] across the time, a call graph of the execution, and so on. There are several types of profilers such as statistical profilers (using operating system interrupts to sample the execution and usually do not need to instrument the binary or source code) and instrumentation profilers (adding instructions to the source code to perform some measurement).

The most used profiling tools by the industry focus on Hybrid MPI+X programs, featuring process communication and thread duration, where X represents another parallel library like OpenMP. Such tools do not supply features specific for task programming, in most cases, even if there is a monitoring of the start and end of a task, they hardly show their dependencies as well, for example. It is difficult to profile task-based applications with these tools, so programmers would highly benefit if more specific task information were provided.

During the development of the project, an article was published on the OMPC [35], of which I was a co-author, and a poster with a short-paper called "Profiling Analysis for a Distributed Task-Based Runtime" was accepted in the CARLA 2022 [4], of which I was the main author.

---

[2] In profiling tools, events are relevant information that occurs at a certain moment of execution.

1.1 Motivation

Due to the lack of analysis available in profiling tools focused on task programming, the motivation of this study is to develop profile tool features to address the needs of task-based runtimes, as metrics and additional information on the timeline. Despite the chosen runtime is the OMPC, the developed concepts could be applied to other task-based runtimes.

1.2 Objectives

This study aims to provide new profile features related to applications based on task parallelism and running on HPC clusters, such as metrics and information on the timeline. To achieve this goal, it was necessary to research the parallel code profiling needs of distributed tasks-based programs and how to improve the existing tools. This work proposes to answer the following research question: what metrics and features can be helpful for programmers to understand the behavior of parallel applications based on tasks to improve their performance? Metrics of parallel efficiency, such as load balance, communication, and computation efficiencies are offered by some tools and also can be helpful to this type of application. Furthermore, the timeline view of events, commonly provided by the existing profiling tools, can be improved by adding task-related events. Also, it is possible to perform some analysis in the task graph to extract useful parallel information, such as the critical path (i.e. the task execution path that takes longer to run).

1.3 Thesis Structure

This thesis is structured as follows: Chapter 1 presents an introduction and motivation for carrying out the study; Chapter 2 describes the programming models as well as their implementation with emphasis on the OMPC runtime. Chapter 3 provides an overview of profiling and existing tools. In Chapter 4, the OMPC profiling tools developed during the study are presented. In Chapter 5, the results of the profiling metrics developed are presented. Finally, Chapter 6 concludes the study.

Chapter 2

Parallel programming models

Developers face several difficulties to parallelize their applications, mainly related to which programming model is more suitable to extract the best performance. The parallel programming model defines the work distribution across different computers, devices, processes, or threads. Typically, loops are well-known parallelization hotspots, however, the decision about the workload division directly affects the performance. Commonly models to divide the work are loop parallelism, task parallelism, or a combination of both, and it is important to select the most appropriate combination of libraries and runtimes for the chosen model. Section 2.1 describe these models and Section 2.2 presents some libraries and runtimes used to implement them.

2.1 Parallelism Models

2.1.1 Loop parallelism

The loop parallelism consists in dividing the loop by sections of iterations and then distributing these sections among computer threads. For example, it is possible to define that two threads will process the work, with one of them processing the odd iterations and the other the remaining. Or, simply dividing equally the iterations across the available threads. The advantage of this approach is that it is usually simple to implement, although its limitation is the work division if the iterations have data dependencies on each other. For example, calculating matrix multiplication is a good application of this method, since it is possible to build a loop in which the iterations do not depend on each other. On the other hand, it is difficult to use it for a Fibonacci numbers generator, since each iteration depends on two previous iterations. The OpenMP standard specifies a compiler directive to easily parallelize loops and also permits users to choose how the iterations will be divided. Listing 2.1.1 shows an example of a square matrix multiplication algorithm using the loop parallelism of OpenMP. Parallel for compiler directive created the threads that consume and execute each iteration of the loop from line 2.

```
1 #pragma omp parallel for
2 for(int i=0; i<SIZE; i++)
3    for(int j=0; j<SIZE; j++)
4       for(int k=0; k<SIZE; k++)
```

```
        C[i][j] += A[i][k] * B[k][j];
```

Listing 2.1: Square matrix multiplication algorithm using the loop parallelism of OpenMP, written in C language. The product of matrices A and B is stored in matrix C. The directive divides the loop iterations across the computer threads. Iterations are not dependent on each other.

2.1.2 Task parallelism

Task parallelism consists in defining pieces of codes, called tasks, that will be executed in parallel. This approach is more flexible since it is possible to define dependencies between different tasks and runtime can schedule tasks satisfying these dependencies. The OpenMP standard specifies another compiler directive to define tasks with dependencies on each other, regardless of the limitation that the execution of tasks is distributed only at the thread level. Listing 2.1.2 presents an example of calculating distance in a binary tree using OpenMP tasks. The `calculate_distance_from_root` function calls `calculate_distance` for the root node, with distance starting at 0. Parallel compiler directive created the threads that consume and execute each task generated in the `calculate_distance` function.

```
void calculate_distance(node * n, int parent_distance){
  n->distance = parent_distance + 1;

  if(n->left != NULL)
    #pragma omp task
    calculate_distance(n->left, n->distance);
  if(n->right != NULL)
    #pragma omp task
    calculate_distance(n->right, n->distance);
}

void calculate_distace_from_root(node * root){
  #pragma omp parallel
  #pragma omp single
  calculate_distance(root, 0);
}
```

Listing 2.2: Recursive function for calculating distance in binary tree using OpenMP task parallelism, written in C language. The distance of the child nodes is calculated through the task directive. In this case, it was not necessary to define any dependencies.

Due to difficulties in ensuring that dependencies are created properly and understanding task execution, it is common to create a graph representation. Nodes represent tasks and edges represent dependencies. Figure 2.1 shows an example of a task graph and how it can be executed using two threads. Note that the order of execution must consider the task's graph dependencies, i.e. task 5 can only be executed after task 2 and task 1 must be executed before tasks 3 and 4. As in this example, other information can be added to the graph, such as task time or the size of the transferred data. In addition, it is possible to extract useful information for code optimization, such as the critical path, described in the following section.

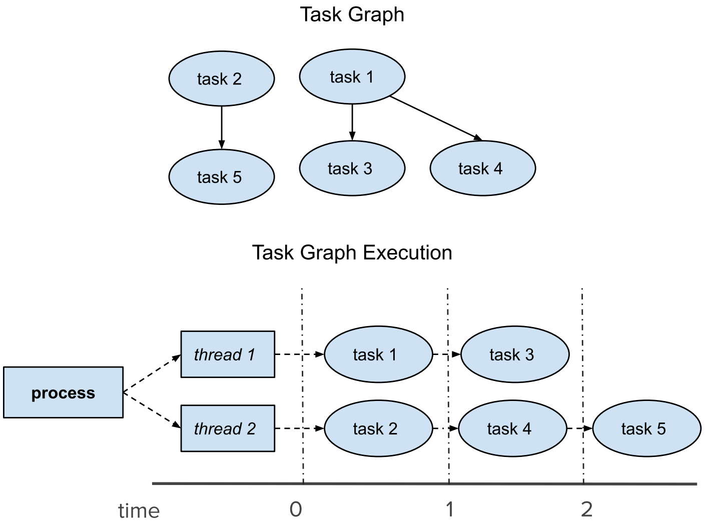


Figure 2.1: The top side of the Figure shows a task graph with its dependencies. An example of a possible execution schedule for this graph is shown at the bottom, considering one process running two threads. Note that the duration of tasks is the same in this representation, which does not occur in real executions.

Critical path

The critical path is the most time-consuming path in a task graph, therefore it is the path that dominates the parallel time consumption. Determining the critical path of a task application is important because its reduction will also decrease the parallel time and consequently optimize the application. However, it may not be possible to reduce the critical path, indicating that it is impossible to improve the application performance. In this case, increasing the number of computer cores will not lower the application time. Moreover, even if the reduction is possible it may not affect application performance significantly, since other paths may have similar times. Figure 2.2 presents an example of critical path calculation from a graph obtained from the Task Bench [31] application profiled using OmpTracing [28], a profiling tool for OpenMP programs developed in our previous work.

This work [34] presents an algorithm that uses program activity graphs (PAGs) to calculate the critical path of distributed programs. It uses the Charlotte distributed operating system, in which the basic communication primitives are messages of send and receive, and extract PAGs from the IPS performance measurement tool. Another similar method is present in this study [12] that extracts performance indicators from a critical path analysis of Scalasca's parallel trace replay. Scalasca [16] is a performance analysis tool that includes call-path profiling support. Both studies extract the critical path of distributed programs and do not address task model programming.

The determination of the critical path in many studies focuses on calculating it before its execution to schedule tasks or extract it from the program execution log. In this dissertation, the critical path calculation is similar to this second approach, centering on obtaining it from the task graph after the program has finished executing.

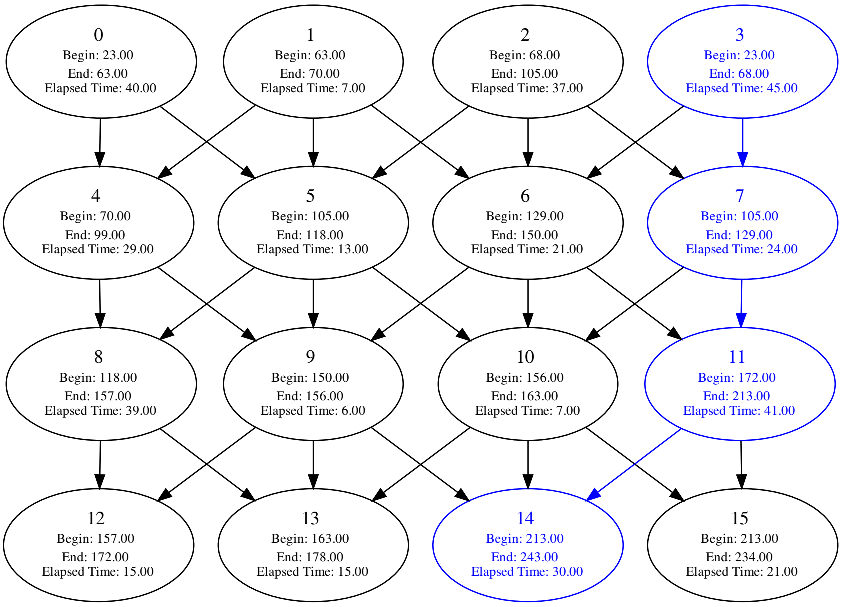


Figure 2.2: A graph from the Task Bench application obtained through OmpTracing. The graph shows the task's duration, start, and end time. Tasks and dependencies highlighted in blue correspond to the critical path. In this case, the critical path is calculated using just the elapsed time.

2.2 Implementation of Parallelism Models

There are several libraries that can be used to implement the programming models described. MPI is widely used combined with another library, which is typically represented as an MPI+X implementation, where X is another parallel library like CUDA, OpenMP, OpenCL, OpenACC, etc. These different libraries allow parallelism exploration on different heterogeneous devices and computers.

Some runtimes propose their own implementation of the task-based parallelism model, relying on different programming languages or libraries. As examples, there are Legion [8], StarPU [7], Charm++ [18] and OMPC [35]. Legion is a C++ runtime that includes task and data parallelism, organized around logical regions to perform computations. StarPU is a runtime system for heterogeneous platforms with CUDA or OpenCL accelerators. Charm++ is a task-based runtime that provides a separation between sequential and parallel objects. OMPC is a runtime that allows the distribution of tasks using OpenMP directives, hiding the MPI implementation complexity from the user. OMPC's advantage over these other runtimes is that it relies on the OpenMP standard, which is one of the most popular parallel programming models. The following section further describes the OMPC runtime.

2.2.1 OMPC

OpenMP Cluster (OMPC) is a task-based distributed runtime that allows cluster programming through OpenMP directives. It is built on top of LLVM [22] as a new device plug-in built around the original OpenMP runtime. In a classic OpenMP implementation, the directives are designed for shared memory programming, that is, at the thread level, or for use with devices, such as GPUs. Applications that run in a distributed environment commonly associate OpenMP with MPI, distributing the application on different devices through MPI messages. However, the user has to program at a low level how the data will be distributed and communicated: should program a matching reception message for every send message and implement the load-balancing by hand. So, the code can end up inefficient and error-prone, and writing it becomes much more complex when compared to just using OpenMP directives. OMPC, on the other hand, allows distributed programming by creating tasks using the OpenMP target directive, without having to deal with scheduling and data communication. It uses an MPI-based event system that transfers tasks and data to the nodes, and a HEFT-based[33] algorithm for scheduling tasks. In addition, it has a fault tolerance mechanism.

In the classic OpenMP programming model, tasks are created by a control thread and executed by the worker threads using a work-stealing algorithm to schedule tasks. When a worker thread finishes executing all tasks that are attributed to him, it can steal tasks from other worker threads. This approach is reasonable in multi-threaded applications since the shared memory makes communication between threads fast. However, in applications running in distributed environments, transferring data between nodes can be highly costly causing large overheads that significantly impact application performance. On the other hand, the OMPC schedule algorithm was developed in order to reduce the communication between the nodes. It adopted the HEFT algorithm to static schedule tasks to worker nodes, that is, tasks are dispatched to their worker nodes for execution only after the task graph was entirely constructed. Hence, tasks can be allocated in such a way as to reduce communications between nodes.

The way to program with OMPC is the same as the classic OpenMP target programming which was introduced in the specification of OpenMP v4.0 to program accelerators, especially GPUs. So the programming model is the same: target directives are used to determine tasks and map clauses for sending and receiving data. The nowait clause eliminates the implicit barrier at the end of a target construct, permitting that a parent task can continue its execution even if a target task is not yet finished. OMPC was designed so that programmers can use a second level of parallelism in each cluster node with any combination of OpenMP or CUDA. The Listing 2.2.1 presents an example of a matrix multiplication function written with two parallelism levels. The first level of parallelism divides the multiplication into blocks represented by the loops in lines 4, 5, and 7, with the creation of the task in line 10. In this line, dependencies and data mapping are also defined. In line 14, the second level of parallelism is defined, iterating over the blocks. Alternatively, the multiplication of the inner blocks can be implemented using a call to the BLAS [10] or cuBLAS [5] libraries for a more efficient implementation.

```
1 void BlockMatMul(BlockMatrix &A, BlockMatrix &B, BlockMatrix &C) {
```

```
#pragma omp parallel
#pragma omp single
for (int i = 0; i < N / BS; ++i)
 for (int j = 0; j < N / BS; ++j) {
  float *BlockC = C.GetBlock(i, j);
  for (int k = 0; k < N / BS; ++k) {
   float *BlockA = A.GetBlock(i, k);
   float *BlockB = B.GetBlock(k,j);
   #pragma omp target depend(in: *BlockA, *BlockB) \
                      depend(inout: *BlockC) \
                      map(to: BlockA[:BS*BS], BlockB[:BS*BS]) \
                      map(tofrom: BlockC[:BS*BS]) nowait
   #pragma omp parallel for
   for(int l = 0; l < BS; l++)
    for(int m = 0; m < BS; m++)
     for(int n = 0; n < BS; ++n)
      BlockC[l + m * BS] += BlockA[l + n * BS] * BlockB[n + m * BS];
  }
 }
}
```

Listing 2.3: Example of a block matrix multiplication, extracted from the OMPC documentation. Matrix A is multiplied by B and stored in C. N represents the matrix size and BS represents the block size.

OMPC uses two types of processes: a head and several workers. Each process belongs to a cluster node and has its own set of threads. The head process is responsible for offloading tasks and data, while worker processes execute these tasks, written with OpenMP target directive. Figure 2.3 shows the relation between OpenMP code and processes/nodes. The OpenMP program runs in the head process (1), which creates the tasks graph (2) and defines which tasks are executed in each worker process, considering data dependencies and order of execution. The OpenMP target map directive defines what data is sent to the worker processes, for which the runtime creates MPI Send/Receive messages. Tasks created by OpenMP's target directive, or target nowait, are executed in worker processes, created by the runtime in the head process via MPI. If the user has defined a second level of parallelism in the tasks, it is possible to use GPU devices with OpenCL or CUDA. The head process is responsible for coordinating all execution and sending/receiving data from worker processes. Hence, after completing the computation of the task, the head process receives the results through the MPI messages. In addition, it can also order the sending of data between workers (3), defined as the forward operation. I.e., when a task depends on a result that was produced by a different worker process.

The amount of threads that each process has is defined by the user through environment variables, and different amounts can be defined for different types of threads. The head process creates a control thread and several worker threads, while the worker process creates data and executes event handler threads and a gate thread. The differences between these types of threads are described in Section 4.3.1. The runtime was designed so that any OpenMP code was compatible. Due to its flexibility and ease of use, it is an excellent alternative for exploring task scheduling and for studying metrics and profiling

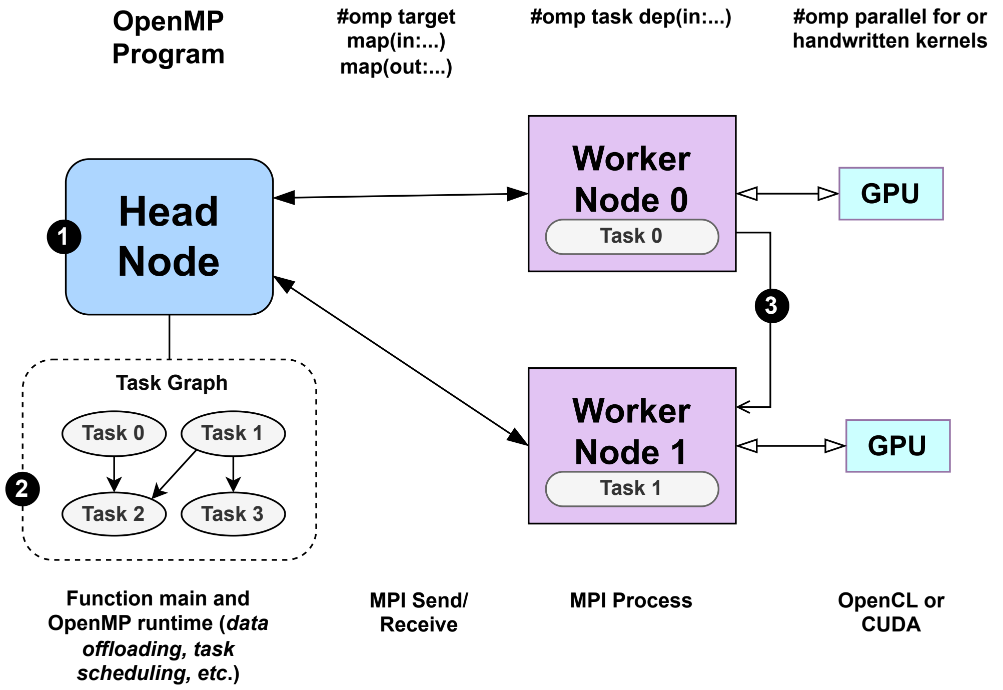


Figure 2.3: Overview of OMPC processes and their link to OpenMP code.

features.

Chapter 3

Profiling

Understanding the behavior of a parallel application may not be an easy task, as the execution is divided across the computer's cores. In distributed environments, it is even more complicated because there is heterogeneity of resources, as hardware and clock differences. It is difficult to verify the correctness of the code, what data each thread computes, and how the memory is accessed. Profiling tools were developed to address these issues, helping to debug the application, analyze performance, and detect bottlenecks and overheads.

There are several models of monitoring code, including event-based, statistical, or instrumentation profiling. Event-based tools achieve profile information when certain events start or end. The monitored events depend on how the application was written and the runtime used. In this model, the order in which events occur is also monitored. Statistical and instrumentation profiling, on the other hand, obtain information through operating system interrupts and by adding profiling code to source code, respectively. Note that a profile tool can implement more than one model. Figure 3.1 presents a comparison of behavior between these three models. The event-based profile monitors events that vary depending on the application and runtime. For example, common events to be monitored are the start and end of parallel regions, the creation, and execution of tasks, etc. Statistical profiles collect samples of the execution state at each periodic operating system interrupt. The samples collect depend on the tool and the type of operating system used. Instrumentation profiles add instructions to the program manually (the user himself chooses the instrumentation points) or automatically.

Despite the behavioral differences, the profiling workflow is quite similar and consists of: 1 - Writing the parallel code; 2 - Instrument the code (optionally, in profilers that support it); 3 - Compile the code with support for the chosen profile tool (optionally, since in entirely statistical profiles, this step is usually unnecessary); 4 - Execute the application (in statistical profiles the application is executed through the tool); 5 - Collect profiling results; 6 - Visualize and analyze these results in order to find optimization points (in this step it is possible to use a tool to post-process the files); 7 - Modify the code to improve performance; 8 - Go back to step 2 to check if there were improvements. Figure 3.2 summarizes this process in a diagram.

In addition, the tool can also have different levels of profiling, such as per process (typically monitoring MPI calls), threads, or events/tasks. Due to these different types and levels of analysis, the profile results can be really extensive and presented in many

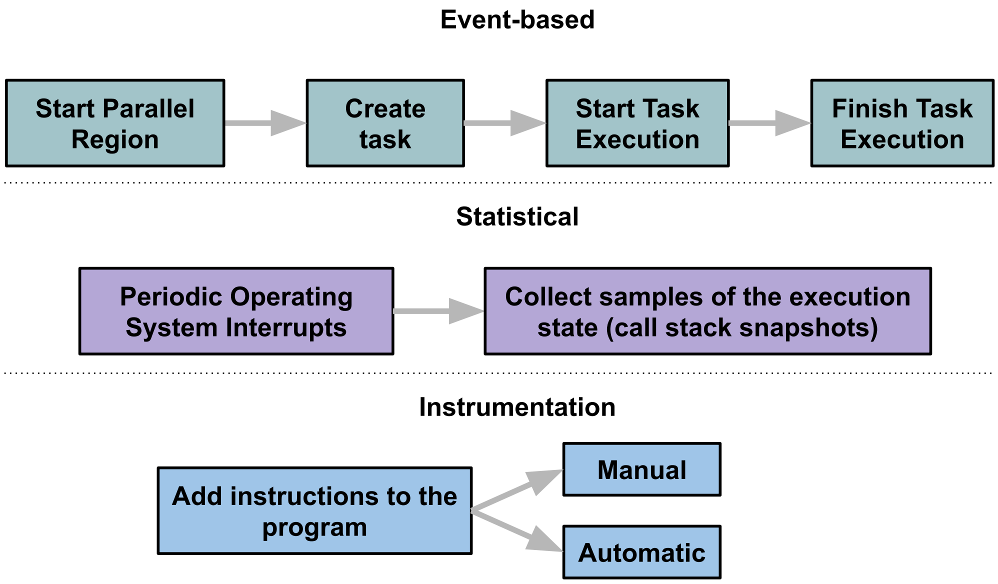

Event-based
Start Parallel Region
Create task
Start Task Execution
Finish Task Execution
Statistical
Periodic Operating System Interrupts
Collect samples of the execution state (call stack snapshots)
Instrumentation
Add instructions to the program
Manual
Automatic

Figure 3.1: The behavior of each profiling model: event-based, statistical or instrumentation profile. The event-base profile only presents examples of common events to be monitored, as each tool will present its own set of events.

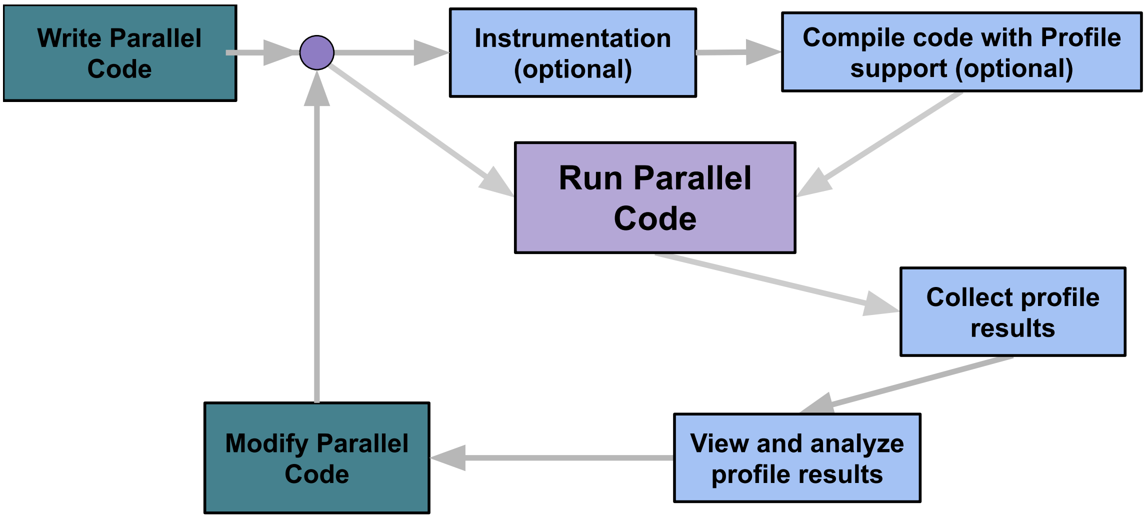
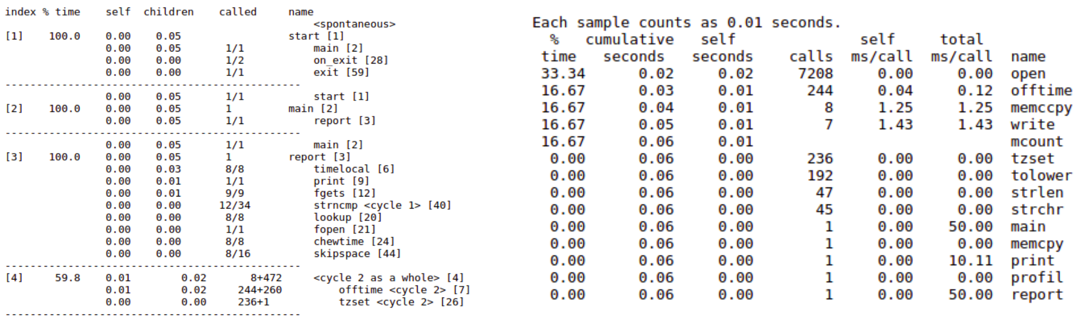

Write Parallel Code
Instrumentation (optional)
Compile code with Profile support (optional)
Run Parallel Code
Collect profile results
View and analyze profile results
Modify Parallel Code

Figure 3.2: The profiling workflow.



```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                0.00    0.05       1/1           start [1]
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1           report [3]
-----------------------------------------------
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1         report [3]
                0.00    0.03       8/8           timelocal [6]
                0.00    0.01       1/1           print [9]
                0.00    0.01       9/9           fgets [12]
                0.00    0.00      12/34          strncmp <cycle 1> [40]
                0.00    0.00       8/8           lookup [20]
                0.00    0.00       1/1           fopen [21]
                0.00    0.00       8/8           chewtime [24]
                0.00    0.00       8/16          skipspace [44]
-----------------------------------------------
[4]     59.8    0.01        0.02         8+472     <cycle 2 as a whole> [4]
                0.01        0.02       244+260         offtime <cycle 2> [7]
                0.00        0.00       236+1           tzset <cycle 2> [26]
-----------------------------------------------
```

call-graph

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
```

flat profile

Figure 3.3: Call-graph and Flat profile obtained from GVProf documentation.

different ways. Some of the more common ones include a call graph or flat profile, both of which show the cumulative time of function calls, a task graph (nodes represent tasks and edges represent data dependencies), a timeline (events as colored bars, representing the duration and moment that occurred), and statistical graphs. Figure 3.3 shows an example of a call-graph and flat profile extracted from the documentation of the GVProf [36] tool. The call graph shows the time spent on functions and their calls. On the other hand, a flat profile only shows the time spent on each function. Figure 2.2 shows an example of a task graph obtained through OmpTracing and in Section 4.3 there are several examples of OMPC timelines.

Profiling tools can provide analysis of time, hardware (energy measurements, frequency monitoring, etc), memory (total memory usage, memory leak, data redundancies, etc), and metrics (load balance efficiency, communication efficiency, and others). This can lead to another difficulty: how to interpret these results and modify the code to achieve a performance gain. Also, monitoring a profiling tool adds overhead to the application and can produce large traces, impacting post-analysis scalability. In the following sections, some profiling tools and a comparative analysis of them are presented.

3.1 MPI+X Profiling

Due to the complexity of hybrid MPI+OpenMP programs, using multiple ways of performance analysis is quite common. Several profiling tools are used to assist with this analysis, each of them providing specific efficiency metrics. The Performance Optimisation and Productivity (POP) Centre of Excellence in HPC [11] divides these metrics into two categories: parallel efficiency and computation efficiency. The first one reflects how well parallelized the code is, split into load balance efficiency and communication efficiency. The second reflects how well the computation scales as the number of processes/threads increases, split into instruction scaling and instructions per cycle scaling. The POP group developed PyPOP, a python package that calculates these and other metrics from Extrae[26] traces, designed to be used with Jupyter notebooks.

Score-P [21] is a joint performance measurement infrastructure for Periscope [9], Scalasca [16], TAU [30], and Vampir [20], intended to be used with C/C++ and Fortran codes. It was designed to collect data such as times, communication metrics, or hardware counters, of OpenMP or MPI applications. The user needs to instrument the application to collect measurement data, stored in the OTF2, CUBE4, or TAU snapshot formats or queried via the online access interface. POP Metrics, present previously, can be automatically calculated using Scalasca.

Extrae is a performance analysis tool that generates trace files for Paraver[27] tool. It supports MPI, OpenMP, pthreads, OmpSs, and CUDA libraries. Extrae allows users to manually instrument the application to extract information about a specific part of the code or to use sampling mechanisms to achieve performance data. The monitors added by Extrae provide performance and counter metrics of multiple components of the system, collected using PAPI and the PMAPI interfaces, and accurately measure time through the timestamp mechanism. Paraver is part of the CEPBA-Tools toolkit and provides a trace view of performance data achieved with Extrae. It offers a wide set of time functions, a filter module, and a mechanism to combine two timelines, displaying a large number of metrics related to workload, application, task, and thread. Paraver provides two main displays, a timeline that represents the application processes behavior over time, and a statistics display with numerical analysis of the user-selected region. It shows information about the system, node, and CPU, such as time-dependent values (semantic values) and communication lines. Also, a set of building blocks (filter and semantic modules) are available to be combined, permitting transform the trace in the visualization process.

HPCToolkit [6] is a very complete tool for profiling hybrid MPI+OpenMP that provides measurement, analysis, attribution, and presentation of application performance. It provides two presentation tools to analyze the profiling results, the hpcviewer, and hpctraceview. The hpcviewer presents some performance metrics mapped to a program's source code, such as the costs of an execution's dynamic calling contexts, for example. On the other hand, the hpctraceview presents a diagram that shows the parallel execution over time. The programmer can use these analyses to detect bottlenecks and increase parallel performance.

TAU [30] is another very complete and well-known profiling tool for performance analysis in hybrid MPI+OpenMP programs. This tool provides several analysis metrics and profiling variants such as callpath profiling (distribution of performance along the event calling paths), calldepth profiling (distribution of performance across program parts from a top-down hierarchical perspective), and phase profiling (performance data relative to execution state). Also, it provides a tracing view of the events that characterize the execution.

Timemory [25] is a modular performance analysis for HPC, written in C ++ 14. It is a modular system that uses template metaprogramming for user-defined performance measurements and analyses, available for code written in C, C++, Python, and Fortran. Timemory supports interoperability with CUDA, MPI, UPC ++, and several forms of multi-threading. It provides a common instrumentation interface that permits multiplexing analysis tools and performance measurement. Timemory aims to supply the needs for composite components, enabling the creation of a set of complementary features with different capabilities that can be used in specific use-case scenarios. It allows several ways of instrumentation profiling and the authors are currently developing statistical profiling.

3.2 Task Profiling

Charm++ [18] is a task-based runtime, an portable parallel programming language based on C++ that provides a separation between sequential and parallel objects. Projections[19] is a very complete parallel performance analysis framework used to profile Charm++ applications. It has an event tracing component that allows the user to control the information generated and a Java-based visualization and analysis component with several view options. Projections is simple to use, the user only needs to link the application with the trace generation module. The performance views available are quite diverse: graphs (data breaking by intervals), a timeline of processors, usage profile (percentage-wise what each processor spends its time), communication properties and time, histograms (to examine performance property distribution), performance counters, animations, and others.

Legion [8] programming model is a C++ runtime that includes task and data parallelism. This runtime is organized around logical regions to perform computations, with independent data, tasks, and functions. Legion includes a task-level profiler called Legion Prof, and it is compiled by default. Legion Prof provides logs in a compressed binary format using ZLIB and performance timelines. Each processor is related to one timeline that shows operations performed. The timeline provides a utilization graph of the memories and processors during the run and other additional information.

StarVZ [29] is a performance analysis tool for hybrid CPU/GPU task-based applications. It is designed to attend StarPU [7] applications, providing an in-depth performance analysis to address problems of task dynamic scheduling and task irregularities. StarPU is a runtime system for heterogeneous platforms with CUDA or OpenCL accelerators. StarVZ Framework is publicly available as an R package and exploits the application structure, runtime system, and hardware information. It provides original application-oriented panels in two visualization groups, runtime-oriented and application-oriented panels. The panel's organization incorporates a space-time diagram with application tasks and computing resource information. The relevant data is displayed by filtering and aggregating the trace information, in addition to some statistics computation. Also, the StarVZ workflow provides scalability support.

3.3 Comparative analysis of profiling tools

The profiling tools presented in Section 3.1 and Section 3.2 have some features in common and they are utilized in different contexts according to user application characteristics. Table 3.1 shows the libraries and programming languages supported by each of them, the PT column is a name abbreviation for the tool used in the next tables. Tools with a greater number of programming languages and supported libraries have the advantage of being able to be used in several different contexts, however, they may have higher overhead and not be suitable for more specific analyses. On the other hand, less versatile tools can provide more detailed analysis for specific contexts.

Some profile tools just generate analysis files and use other extensions or tools to visualize results. This is the case of Extrae, for example, which generates trace files supported by the Paraver GUI tool. Moreover, some of them are built up using other

Profile tool	PT	Libraries Supported	Programming Languages
PyPop	PP	-	-
Score-P	SP	MPI, OpenMP, pthreads, CUDA, OpenCL, OpenACC	C, C++, Fortran
Extrae	EX	MPI, OpenMP, pthreads, CUDA, OpenCL, OmpSs	C, C++, Java, Python
HPCToolkit	HT	MPI, OpenMP, CUDA, OpenCL	C, C++, Fortran
TAU	TAU	MPI, OpenMP	C, C++, Java, Python, Fortran
Timemory	TM	MPI, OpenMP, CUDA, UPC++	C, C++, Python, Fortran
Projections	PJ	Charm++	C++
Legion Prof	LP	Legion	C++
StarVZ	SVZ	StarPU, Cuda, OpenCL	C++
OMPC Profile	OMPC	OpenMP, Cuda, OpenCL	C, C++

Table 3.1: Libraries and programming languages supported by each profiling tool. The PT column is the tool abbreviation name. PyPop analyzes traces of Extrae, so it does not have to support libraries and programming languages.

ones, such as TAU, which uses Score-P, and it can use other tools to improve its analysis, such as PyPop. All of this information is presented in Table 3.2.

PT	GUI tools	Other Tools
PP	Jupyter	no
SP	Scalasca, TAU, Vampir, Cube, Extrae-P	no
EX	Paraver	PyPOP
HT	hpcviewer, hpctraceview	no
TAU	Paraprof	no
TM	timemory-plotter	TAU, Caliper, etc
PJ	no*	no
LP	Browser	no
SVZ	no*	no
OMPC	Chrome Trace	OMPC Bench

Table 3.2: GUI tools used for each profiling tool and his additional analysis tools. In the case of Projections and StarVZ the visualization is integrated in the tool itself.

Table 3.3 summarizes the profile metrics available for each tool. The metrics are divided into four categories: time, communication, memory, and hardware. Time metrics are related to the execution time of the whole application or some parts of them. Communication metrics are related to the communication between processes or threads. Memory metrics give memory usage information of hardware components. Finally, Hardware metrics provide additional information such as hardware counters and frequency.

Table 3.4 shows the specific features for task analysis available for each profiling tool.

PT	**Time metrics**	**Communication metrics**	**Memory metrics**	**Hardware metrics**
PP	parallel efficiency, global efficiency, etc	communication efficiency	no	IPC scaling, frequency scaling, etc
SP	execution time*	collective communications, P2P, etc	total memory	PAPI, perf hardware counters
EX	timeline*	number of communications	no	PAPI
HT	cputime, realtime, wallclock, etc	no	memory leak, IO	PAPI, perf hardware counters
TAU	cputime, realtime, wallclock, etc	MPI communication matrix	memory allocations, IO	PAPI
TM	cputime, realtime, wallclock, etc	no	peak RSS, IO, etc	PAPI
PJ	timeline*	communication CPU overhead, number of messages sent, etc	memory usage	no
LP	timeline*	no	no	no
SVZ	timeline*	communication tasks	memory usage	no
OMPC	timeline, parallel efficiency, task duration, etc	communication efficiency, communication tasks, etc	no	no

Table 3.3: Profile metrics available for each tool. In the Time metrics column, the timeline fields indicate that the tool only has the timeline and it does not allow change time properties. Score-P does not provide a GUI tool for visualization by itself, so the execution time is shown in numbers as the application result.

Legion Prof and StarVZ are specific for task-based runtimes and give more detailed information about task execution.

Compared to the other tools that offer task monitoring, the developed profiling tool not only presents the tasks in the timeline with their dependencies and additional information, but also offers specific metrics designed to improve the understanding of their behavior and to help with code optimization. Details about the metrics and how they can be used are described in Section4.5

PT	**Task analysis**
PP	no
SP	omp tasks on the timeline (using Vampir GUI tool)
EX	execution time of omp tasks
HT	no
TAU	no
TM	no
PJ	execution time of Charm++ tasks
LP	Legion tasks on the timeline: duration time, state of the task (waiting or ready to be scheduled or executing), data dependencies
SVZ	StarPU tasks on the timeline: duration time, task dependencies, number of tasks
OMPC	OMPC tasks on the timeline: duration time, task dependencies, source location, task type and data. Task and Graph metrics

Table 3.4: Task analysis provided by each profiling tool

Chapter 4

OMPC Profiling tools

OMPC provides an integrated monitoring feature that eliminates the need for additional tools. Known as an OMPC Profile, this feature generates task graphs and trace files, which provide valuable information about application behavior. Trace files can be viewed as timelines organized into event-based sequences and segmented by processes and threads. Each trace file represents a different process with one head process and the remaining as worker processes. The graph files include the OMPC tasks (created via the OpenMP target directive), data tasks (representing the data operations generated from the OpenMP map directive), and their dependencies. Together, these files provide a comprehensive overview of application performance and allow for efficient debugging and optimization.

The OMPC events on timelines are as close as possible to the runtime source code. Due to this, it can be difficult to understand for a user who is not very familiar with the runtime. During the project, several post-processes have been developed to improve the timeline view. Some of these features are merging and filtering traces to a new user-friendly timeline view, and adding new useful information such as the dependencies of task graphs. Improved timelines help users to better understand parallel code and identify optimization points. However, the analysis is limited to user interpretation and is more complicated in large applications with many tasks and threads. In this way, after analyzing particular characteristics of the OMPC runtime, some metrics were developed to guide optimizations, which are extracted from profile files. It is expected that through the metrics and analysis of graphs and traces, users can understand the code and achieve insights on how to improve it, increasing the parallel performance.

As processing these features during runtime would be costly, these functionalities are performed after execution and by a separate library called OMPC Bench [23], a command-line tool of OMPC project written in Python. In addition to the post-processing functions of profiling files, OMPC Bench also allows benchmarking OMPC programs, but this feature will not be detailed here as it is not relevant to this study. All traces from the OMPC project can be viewed through the Chrome Trace [1] browser extension.

OMPC project also provides the OmpTracing [28] tool developed in our previous work, if there is a need to extract specific information from the OpenMP runtime, such as from tasks created with the task directive, or the duration of loops and parallel regions. OmpTracing is a lightweight profiling tool for OpenMP programs and also generates traces and task graph files, similar to the OMPC profile. It is possible to combine the traces of both of them using OMPC Bench.

This chapter describes the main contributions of this work: the development of specific profiling features and metrics for task-based applications. All metrics and timeline operations were developed during this project, done exclusively by me with guidance and advice from other project members. The following sections will present the functionalities developed. Section 4.1 shows how to use OMPC profile and OMPC Bench, Section 4.2 describe features of the visualization tool, Section 4.3 and Section 4.4 presents, respectively, components of OMPC timeline and task graph. Finally, Section 4.5 defines profile metrics.

4.1 OMPC Profile Workflow

When running an OMPC program, the user can configure the runtime to generate trace files or graph separately through environment variables[1]. If the application was compiled with debugging symbols, the timeline will display variable names and source code information such as file name, line, and column. The traces are saved in JSON format, while task graphs are saved in DOT format. Two graph files are generated, one containing more information about the scheduling algorithm. OMPC executes applications in a distributed environment, so the traces are divided by process. Therefore, the number of timelines is equal to the number of processes, with one timeline representing the head process and the remains representing worker processes. Once profile files are generated, users can take advantage of the OMPC Bench tool to carry out various operations such as:

- **Merge traces:** Used to merge traces from the same execution into a single one. The user provides OMPC and optionally OmpTracing traces that are merged in a single trace. It automatically synchronizes and creates a user version trace, however, it is possible to generate a developer version trace instead. Since the application runs distributedly, each trace's timeline starts at a different time due to differences in device clocks. Synchronization is crucial to ensure that all trace files start at the same point. While this form of synchronization is a good approximation for current tests, future studies should aim to develop more accurate synchronization methods.

- **Improve timeline view:** Used to generate a trace in a user-friendly view. The user version trace provides a clearer understanding of the application by renaming events for better comprehension and adding information about variables used, task identifiers, and arrows between related events. Additionally, tasks are colored according to their source code, and if task graph files are available, the task dependencies are included in the timeline. This approach allows users to easily identify which points on the timeline correspond to their source code and the execution order of tasks based on their dependencies.

- **Extract Metrics:** Used to extract information from timelines and graphs, its goal is to provide statistical insights into application behavior, allowing users to understand

---

[1] For more information see OMPC Profile Wiki

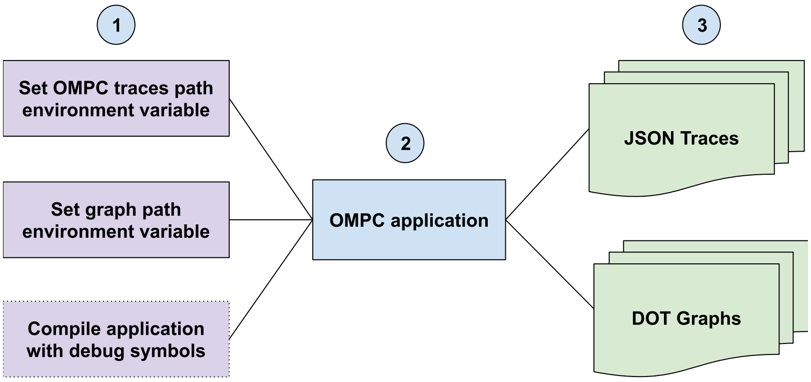

1
Set OMPC traces path environment variable
Set graph path environment variable
Compile application with debug symbols
2
OMPC application
3
JSON Traces
DOT Graphs

Figure 4.1: Steps to generate profile files. The first step (1) consists of setting OMPC traces path and graph path environment variables, and optionally compiling the application with debug symbols. The second step (2) is to execute the OMPC application. In the final step (3) the generated files are collected: traces in JSON format and graphs in DOT format.

> the code's performance and identify areas for improvement. The user provides the OMPC timeline and chooses which metrics to extract: task metrics, graph metrics, or OMPC metrics. These metrics are described in detail in Section 4.5, outlining their functionalities and differences. Once chosen, the resulting metrics are displayed in a table format as a CSV file. Users can also provide multiple applications to compare metric results.

The workflow steps are summarized in Figure 4.1, which shows how to generate profile files, and in Figure 4.2, which shows the use of OMPC Bench. The mentioned filter option is the additional operation to improve the timeline described above. All generated trace files can be viewed as timelines in the Chrome Trace browser extension.

4.2 Visualization Tool Overview

Chrome Trace [1] is a browser extension that loads a JSON file and provides a timeline graphic visualization. It is easy to use and accessible since it can be used just by typing its URL[2] on a Chrome browser. All the traces provided by the OMPC profile, OmpTracing tool, and OMPC Bench follows the Chrome Trace format. The timeline is visualized by clicking to load in the top-left corner and then selecting the trace file or just dragging and drop the file into the window. An example timeline is presented in Figure 4.3. The choice to use this tool was due to not having to implement a new graphical interface. Furthermore, it is very accessible as it only requires the installation of a Chrome browser, which many computers already have installed. The tool's JSON format is simple and allows the addition of several pieces of information, as well as dependencies on the timeline. Finally, since JSON is a well-known format, it is easy to find libraries to process and work with it.

The timeline points indicated in Figure 4.3 by numbers represent as follows:

---

[2] `chrome://tracing`

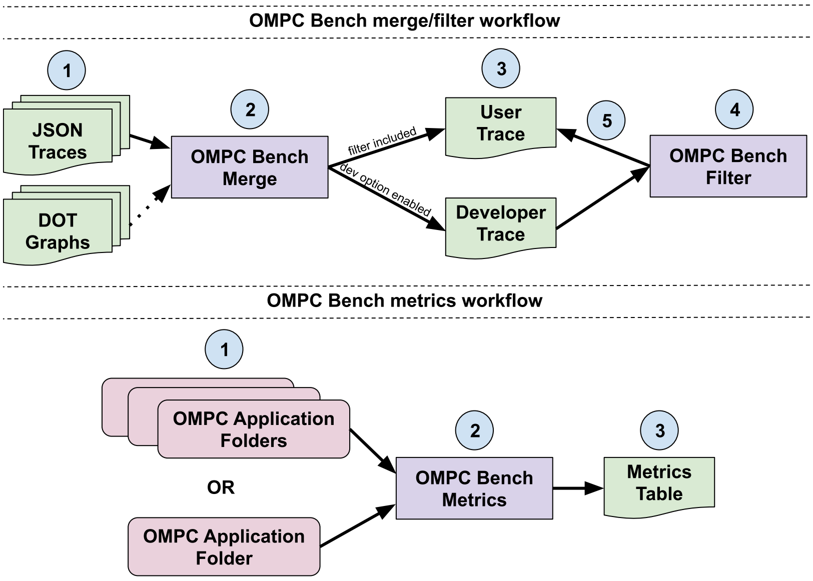
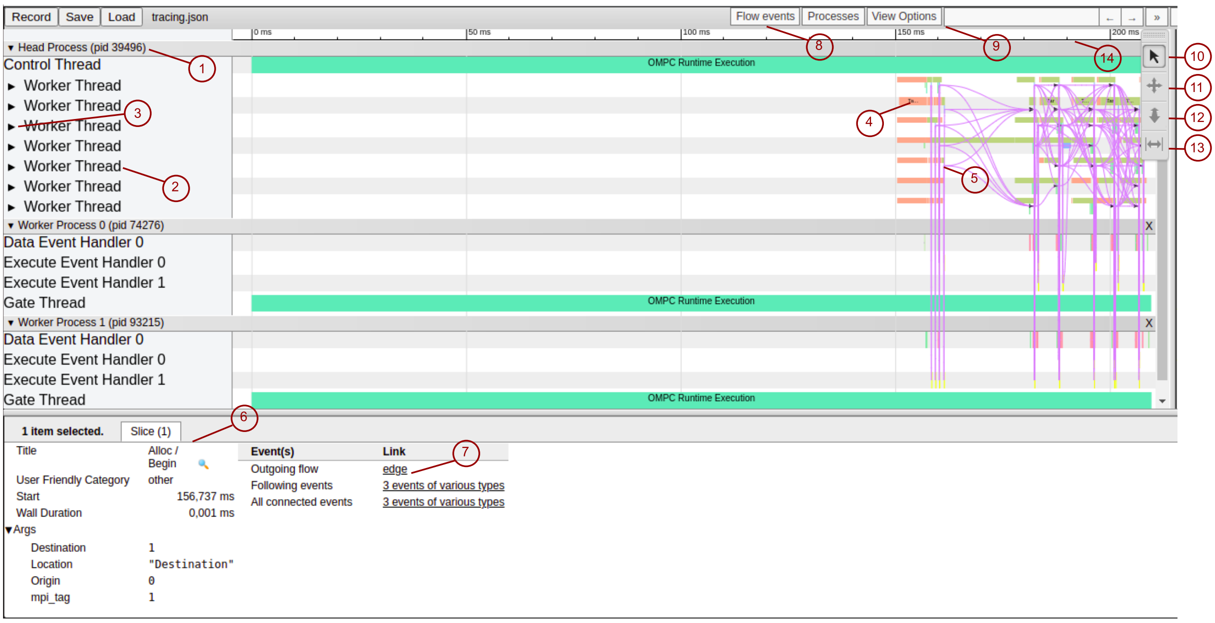


Figure 4.2: OMPC Bench usage. The merge/filter workflow is at the top side of the image and the metrics workflow is at the bottom. The steps to get filtered or merged traces are as follows: (1) Keep all JSON and DOT files of a single application separate in a folder; (2) Run OMPC Bench merge passing the folder path with JSON files prefix. In this step the user can enable the developer timeline option (i.e. no filter applied); (3) If the developer option is disabled, the user trace is generated, otherwise a developer trace is generated ; (4) It is possible to pass the developer trace to OMPC Bench filter and obtain a user trace; (5) User trace generated.



Title	Alloc / Begin
User Friendly Category	other
Start	156,737 ms
Wall Duration	0,001 ms
▾Args	
Destination	1
Location	"Destination"
Origin	0
mpi_tag	1

Event(s)	Link
Outgoing flow	edge
Following events	3 events of various types
All connected events	3 events of various types

Figure 4.3: Example of user timeline with Chrome Trace components indicated by red numbers that represent: (1) Process separation; (2) Thread separation; (3) Event hiding arrows; (4) Timeline events; (5) Relational arrows; (6) Event information; (7) Edge argument; (8) Flow events menu; (9) Search menu; (10) Select tool; (11) Move tool; (12) Zoom tool; (13) Measure Tool; (14) Time indication.

1. **Process separation:** All threads below belong to the referenced process. Each process corresponds to an MPI process created by the OMPC runtime. Usually, a single process is created per node but the user can eventually choose to map them differently for specific cases (e.g. 1 process per socket or 1 process per GPU). This process mapping can be controlled by the user when running OMPC programs just like any other MPI-based programs (using the parameter of *mpirun* or the job scheduler).

2. **Thread separation:** All events on the right belong to the referenced thread. The number of threads created by the OMPC runtime can be controlled by the user through environment variables, they are usually closely related to the number of processor cores available on the machine.

3. **Event hiding arrows:** Used to decrease the height of the timeline, as events from that thread are compressed vertically. It is useful when users need to analyze events that are vertically distant on the timeline. The arrow next to the process name has a similar function but completely hides the threads and events of that process. As an example, all the events of the worker processes have a corresponding event in the head process, so to analyze a certain work process it is possible to hide the other processes and threads that are not related, making the visualization clearer.

4. **Timeline events:** The label indicates what it represents on OMPC. All the colors are chosen by Chrome Trace except for the events named "Task XX", where events of the same color have the same source location and XX is the task id. This event visualization model is quite useful since events with the same name have the same color. So, it is easier for the OMPC user to identify events of the same type.

5. **Relational arrows:** Indicate relations between different events. OMPC timelines use three types of arrows, described in Section 4.3.4. Generally speaking, these arrows help the user to find related events in the timeline (like the corresponding events mentioned above).

6. **Event information:** When an event is selected, by clicking on it, this panel shows some event information. The first lines are information provided by the Chrome Trace tool (as event start, duration, and arrows) and the args section is specific information about this event provided by OMPC. This information helps the user discover the duration of the event and other information specific to the OMPC runtime that if represented as events would make the timeline difficult to analyze.

7. **Edge argument:** Provides information about any arrow from or to this event. If click on it, it will show the two events linked. It is very useful when it is necessary to find the related event but it is far away in the timeline, or very small (common in data communication events).

8. **Flow events menu:** If click on it, is possible to obtain a more clear view of the timeline by hiding the events arrows. The OMPC timeline has many arrows, so

leaving enabled only the category of arrows that will be analyzed at that moment is very practical.

9. **Search menu:** Used to search for events by label or any of its arguments. It is widely used in OMPC timeline analyses, as it is possible to find certain types of events or find corresponding events through their arguments (such as an MPI Tag, for example).
10. **Select tool:** Chrome Trace tool to select events. This feature must be enabled to exhibit event info by clicking on it. Mainly used to display the arguments of a certain event.
11. **Move tool:** Chrome Trace tool to move across the timeline. It is useful when the timeline is zoomed in to a specific point.
12. **Zoom tool:** Chrome Trace tool to zoom the timeline. It is useful to analyze events more precisely and see events that have a fine duration (like communication events). It is possible to zoom a specific event by pressing 'f' on the keyboard.
13. **Measure Tool:** Chrome Trace tool to measure the duration between two events on the timeline. It is useful when the events are in different processes or threads and the user wants to measure the elapsed time between them. For example, in data communication, it is possible to use this tool to calculate its duration.
14. **Time indication:** Time measurement, indicates at what moment of the application the events occurred.

4.3 OMPC Timelines

As mentioned in Section 4.1, the merge operation includes the filter operation by default, however, it is possible to enable the developer option. The developer version timeline simply synchronizes and merges all events from the original traces, generated by OMPC, into a single trace. Figure 4.4 shows a developer timeline example: in this case, all runtime events are preserved with their real names. This version may be useful if the user is more familiar with the OMPC runtime. Figure 4.5 shows a user timeline example. The events are renamed for a user-friendly view and some events are added or compressed (if two events provide similar information it is merged into one with extra information in the argument section). Also, the timeline provides extra information using arrows to relate events. Both figures represent the same execution of the OMPC Plasma matrix multiplication kernel, with 3 nodes (1 head process and 2 worker processes), in which each worker process has 4 threads (1 Data Event Handler, 2 Execute Event Handler and 1 Gate Thread). The Sections 4.3.1, 4.3.2, 4.3.3, 4.3.4, and 4.3.5 describe the user timeline components in detail, including the mentioned threads.

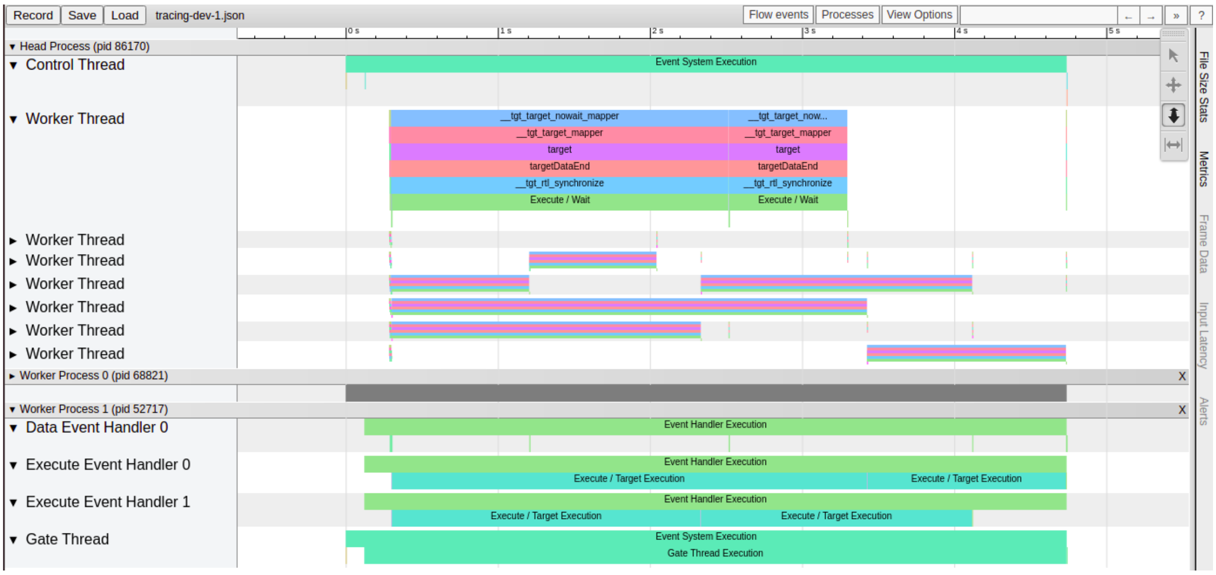


Figure 4.4: Developer timeline version. The event names are related to OMPC runtime code, and all events are displayed.

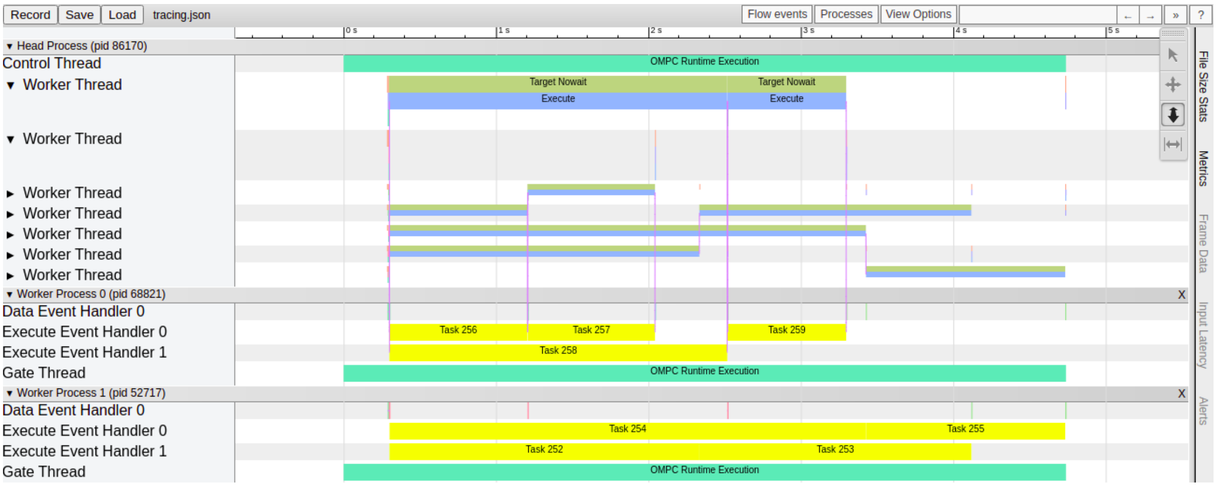


Figure 4.5: User timeline version. The timeline is cleaner as there are fewer events. Arrows to relate events are presented only in the user version. The task events on worker processes are displayed with their identifiers and the colors represent the source location. In this timeline all tasks originate from the same source location, so all of them are yellow-colored.

4.3.1 OMPC Threads

The timeline threads are divided by processes. Each process is a head or worker and the threads that compose them follow the description:

- **Head Process:**
  - *Control Thread:* Responsible for the construction of the task graph.
  - *Worker Thread:* Responsible for data communication and task offloading. Once the control thread finishes building the graph, it helps the worker threads with communication and offloading.
- **Worker Processes:**
  - *Data Event Handler:* Handles data communication events (receiving and submitting data).
  - *Execute Event Handler:* Handle the execution of tasks.
  - *Gate Thread:* handle MPI event notifications from other processes.

4.3.2 OMPC Events

The OMPC Events of the user version indicated in Figure 4.6 represent as follows:

1. **General Events:**
   (a) **OMPC Runtime Execution:** Total duration of application.
   (b) **Variable names:** If the user compiles the application with debug symbols, events that have variable names associated (e.g. Submit) will be nested to a variable name event.
2. **Target Events:**
   (a) **Target Enter (Nowait):** Represents a target enter data map (nowait) region or an entrance in a target data map region (nowait). This region is for mapping data to work processes.
   (b) **Target Exit (Nowait):** Represents a target exit data map (nowait) region or an exit in a target data map region (nowait). This region is for unmapping data to work processes.
   (c) **Target (Nowait):** Target (nowait) region, represents an execution task in the head process. That is from the moment the head process finds a target region, it offloads the task to a work process to execute it, until its finalization.
3. **Task Events:**
   (a) **Task XX:** Total duration of task execution on work processes, where XX represents the task id. The same colors mean the same source location.

   (b) **Execute:** Total duration of task execution on the head process. That is, from the moment the head process orders execution in a work process until its finalization.

4. **Data Events:**

   (a) **Alloc:** When OMPC allocates data in the work processes. In the work processes, this event is divided into a pair of Alloc / Begin and Alloc / End.

   (b) **Submit:** In the head process, represents a data submission to work processes. In the work processes, represent received data (from the head process through *Submit event* or from another worker through the *Forward event*) except in the developer version of the timeline (Forward events always have an associated Submit event representing a data submission). In the work processes, this event is divided into a pair of Submit / Begin and Submit / End.

   (c) **Delete:** The data allocated on the work processes are freed. In the work processes, this event is divided into a pair of Delete / Begin and Delete / End.

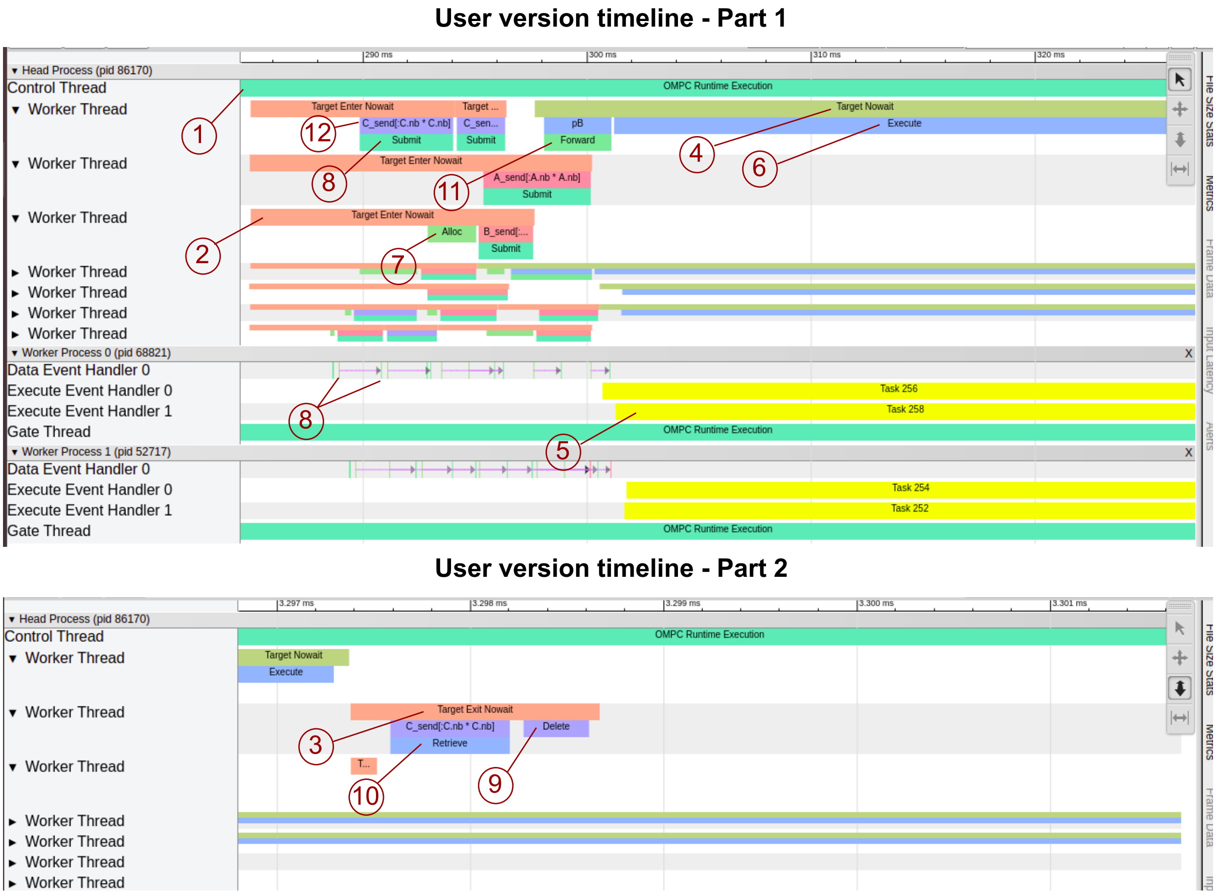
   (d) **Retrieve:** Represents data received in the head process from a work process. In the work processes, this event is divided into a pair of Retrieve / Begin and Retrieve / End.

   (e) **Forward:** Represents the head process sending a message to a worker to forward data to another worker. That is one worker has data that another worker needs, so the head process orders the forward between them. The work process receives the data by the Submit event. In the work processes, this event is divided into a pair of Forward / Begin and Forward / End.

Some events are divided into Begin and End, such as *data events* and *execute events*. The runtime relies on asynchronous calls to the MPI libraries, so it is possible to run multiple events concurrently using the same thread. Note that the events mentioned throughout this dissertation are timeline events, not MPI events. OMPC uses MPI calls in these events, but the user does not have access to such calls and they are not described in the timeline. In timeline improvement, some of these events are merged into a single event in the head process. However, on threads of the work processes, many of them occur at the same time, so to clear the visualization, they remained separate. Most of these events are on the Data Event Handler thread and if they were a single event, there would be an overlap of events. Despite this, it is possible to easily find the corresponding Begin or End event using the timeline dependencies. Figure 4.7 shows an example of how these events are distributed, the arrows link the Begin events to their related End events. The highlighted arrow in the figure shows that between the beginning and end of this event, there is another event, which is common to occur. If all these events were merged, it would make it difficult to visualize the timeline.

4.3.3 OMPC Arguments

Some of the arguments that can appear in OMPC events are listed below:


User version timeline - Part 1

User version timeline - Part 2

Figure 4.6: OMPC Events Timeline. A user version timeline is zoomed in to show different OMPC events. Two parts are zoomed in to display different events. OMPC events indicated by red numbers represent (1) OMPC Runtime Execution; (2) Target Enter (Nowait); (3) Target Exit (Nowait); (4) Target (Nowait); (5) Task XX; (6) Execute; (7) Alloc; (8) Submit; (9) Delete; (10) Retrieve; (11) Forward; (12) Variable names. Event (8) appears two times to show that this type of event is composed of two events (begin and end) on the work processes and a single one on the head process.

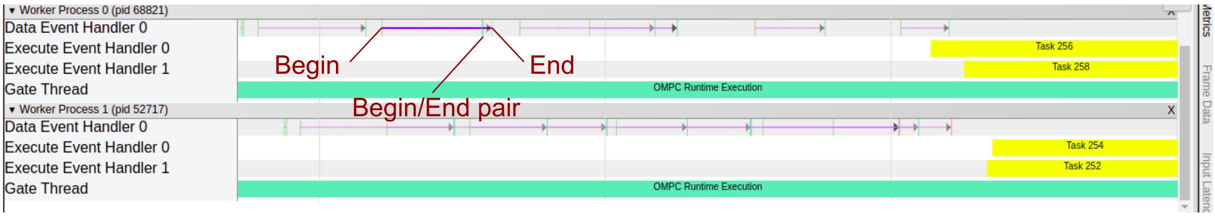


Figure 4.7: OMPC Data Events. In the Data Event Handler thread, data events are distributed in Begin/End pairs, and these pairs are connected by arrows. The highlighted arrow in the image shows that between the start and end of a communication event, there may be other communication events.

- **Origin:** Identifier of the process where the event was created.
- **Destination:** Identifier of the process where the event was executed.
- **Location:** Data and execution events have a pair (origin and destination), and the location indicates which of the pairs the event is on.
- **mpi_tag:** Event id that is the same for the origin and destination pair.
- **task_id:** The identifier of the task that corresponds to the task graph.
- **source_location:** The file, line, and column that the event was executed.
- **identifier:** In target events represents the function that executes the task.

For the origin and destination arguments, timeline identifiers are different from filename identifiers, so 0 represents the head process, 1 represents the work process 0, and so on.

4.3.4 OMPC Dependencies

The OMPC timeline has dependencies (arrows) that indicate relations between different events. These dependencies can be disabled to clear the timeline view. The category numbers presented in the Figure 4.8 represents:

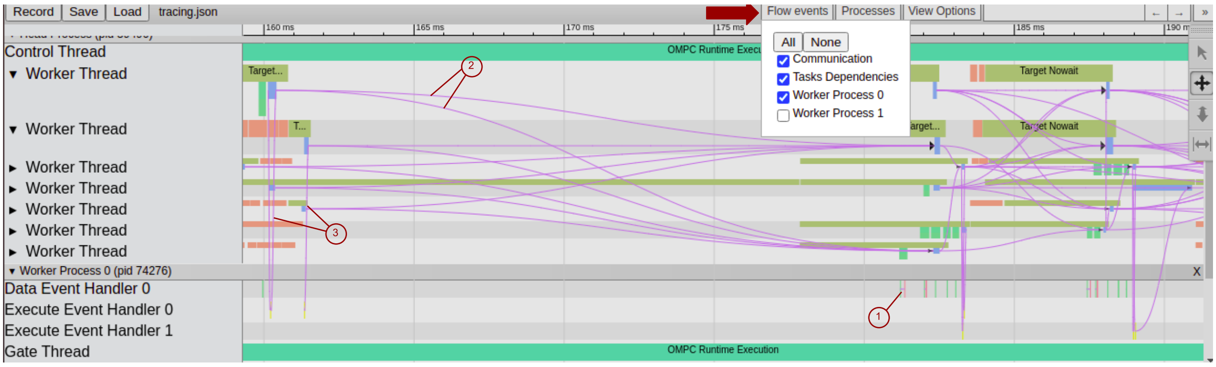
1. **Communication:** Dependencies between event communication pairs (Begin and End) in the work processes.
2. **Tasks:** Data dependencies between tasks in the head process.
3. **Worker Process X:** Dependencies between Execute event in the head process and Task events in the work process X, where X is the id of the work process.

The timeline dependencies are of great help to find related events on the timeline. For example, if it is necessary to find the head process event that generated a worker event, the user would need to search for the mpi tag in the menu and look through the various events that would appear as a result which is the correct event. On the other hand, the dependencies allow just clicking on the event to find all related events. Also, as the head process has multiple threads, without the dependencies it is easier for the user to confuse which event is correct.

4.3.5 Profile Symbols

As mentioned in Subsection 4.3.2, if the application was compiled with debug symbols, by using `-g` flag in GCC/Clang, some events are displayed aligned to a variable name event in the user version timeline. Also, other events present the source location as a timeline argument (the display information when an event is selected). In both cases, the timeline will present the filename, line, and column of the variable or event.

Figure 4.9 shows an example of a variable with its aligned event and how to extract information from them. The `A_send` array was sent from the head process to a work



☑ Communication
☑ Tasks Dependencies
☑ Worker Process 0
☐ Worker Process 1

Figure 4.8: Arrows of user version timeline. The Flow Event menu, indicated by the red flag, shows the categories of user timeline arrows, and it is possible to choose which of them are displayed. The red numbers represent arrows of (1) Communication; (2) Task dependencies; (3) Link between head and work processes events.

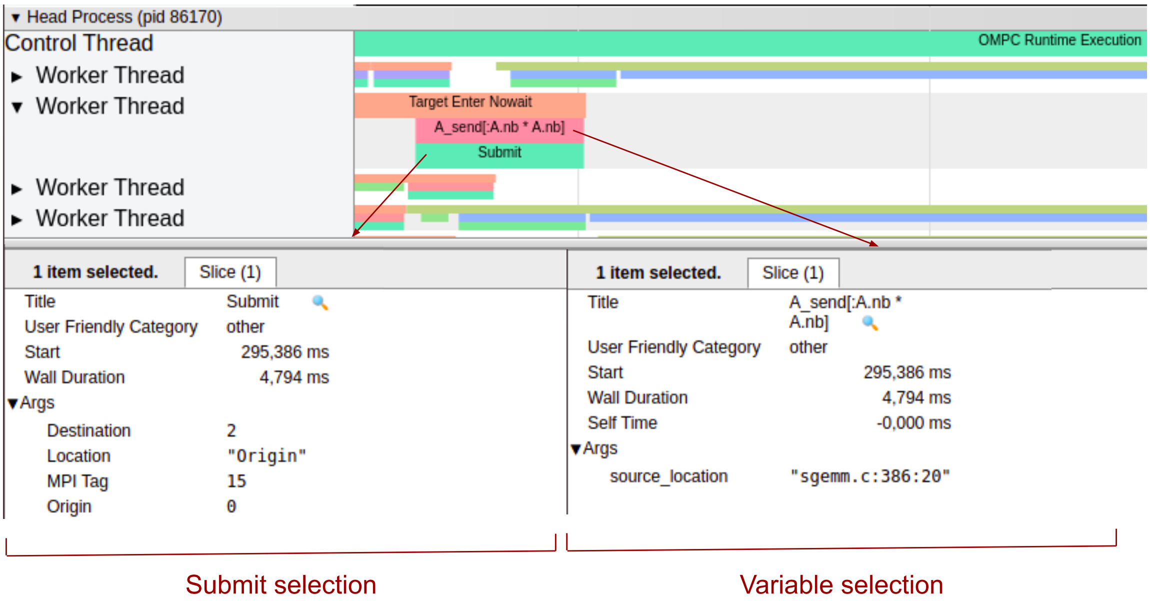
process through the Submit event. When selecting the variable, it is possible to see in which line, column, and source code file this array was sent through the source location argument. Selecting Submit shows that the data was sent to work process 1 through the destination argument (the indicated number is always the worker's identifier added to 1) and also provides its MPI Tag. Knowing the worker identifier and the MPI Tag, it is possible to use the search tool to find the related Submit event, and thus know when the worker received the data.

The timeline with profile symbols is useful to understand how and which data was transferred their duration, and the start point. The user can use this information to reorganize the data division and transfer, and consequently optimizing performance. The source location helps the user link the events appearing in the timeline with its source code.

4.4 Task Graph

OMPC profile creates two graph files after the execution of the application is finished, with tasks represented as oval and octagonal (those that are root or leaf) nodes and data tasks colored in gray. The task identifiers can be used to analyze task execution in the user version timeline. One of those graphs provides more information about the scheduling algorithm (HEFT [33]), with the size of the transferred data as the edge weights and edges colored in blue and red representing, respectively, internode and intranode communication. That is, in intranode communication, the OMPC does not need to transfer the data. Also, in this version of the graph, the source location is added to the tasks. An example of these two task graphs is shown in Figure 4.10 and Figure 4.11, with the mentioned components indicated.

The task graph helps understand the order in which tasks are executed in the timeline, as a task only executes after resolving all its dependencies, and permits critical path analysis. Dependencies only appear in the graph, so to generate a user timeline with them, the graph must be in the same folder as the timelines.



1 item selected. Slice (1)

Title	Submit
User Friendly Category	other
Start	295,386 ms
Wall Duration	4,794 ms
▾Args	
Destination	2
Location	"Origin"
MPI Tag	15
Origin	0

1 item selected. Slice (1)

Title	A_send[:A.nb * A.nb]
User Friendly Category	other
Start	295,386 ms
Wall Duration	4,794 ms
Self Time	-0,000 ms
▾Args	
source_location	"sgemm.c:386:20"

Submit selection

Variable selection

Figure 4.9: Profile symbol usage. The variable event is the one named as `A_send[:A.nb * A.nb]`, which represents an `A_send` array of `A.nb * A.nb` dimensions, and the related nested event is Submit. Information for Submit and Variable events is highlighted at the bottom (Chrome Trace does not allow selecting two events at the same time, so the figure has been edited to display both).

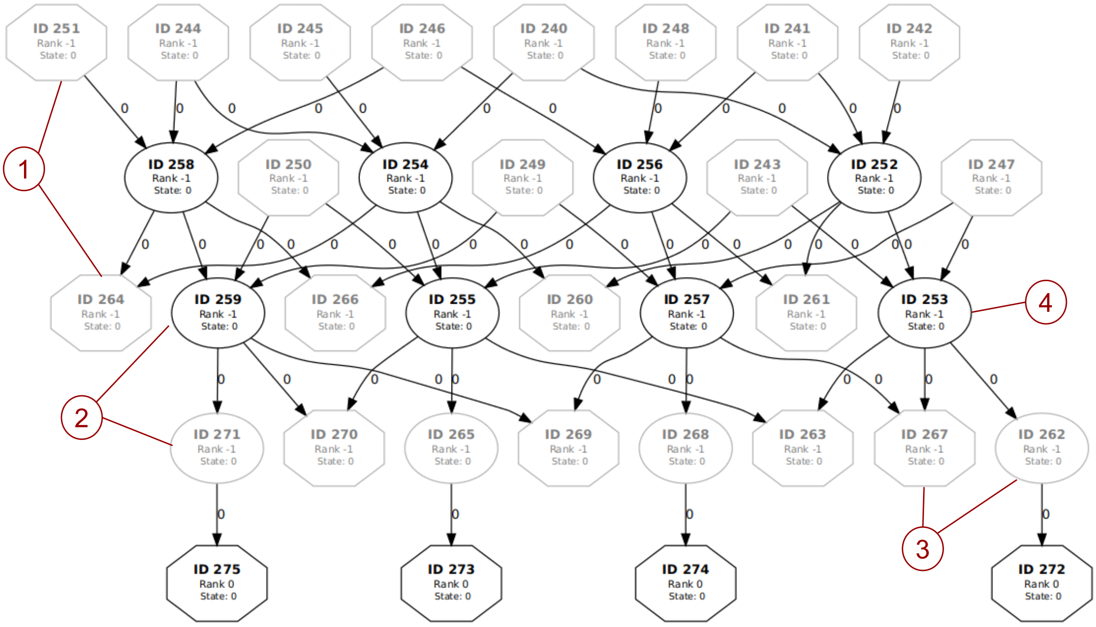


Figure 4.10: Simple Task Graph. This graph is useful when it is necessary to analyze the dependencies pattern and task types. The red numbers indicate (1) Root and leaf nodes as octagonal shape; (2) Non-root and non-leaf nodes as oval shape; (3) Data tasks colored in gray; (4) Execution tasks colored in black.

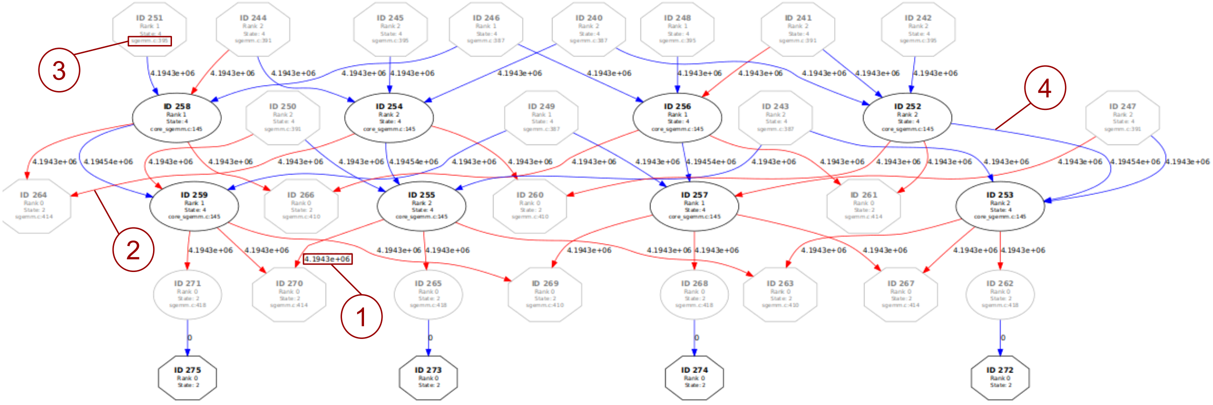


Figure 4.11: Scheduling Task Graph. This graph is useful when it is necessary to analyze the HEFT scheduling algorithm. The red numbers indicate (1) Heft weights (data transferred); (2) Intranode communication; (3) Source location; (4) Internode communication.

4.5 Profile metrics

4.5.1 OMPC

OMPC metrics are created based on POP [11] metrics and other common profile metrics. It was adapted to OMPC runtime and divided into three categories: quantitative metrics, time metrics, and efficiency metrics. The following items describe all of the OMPC metrics, separated by these categories:

- **Quantitative Metrics:** Provide data inherent to the application and are expected to remain the same as long as the same input parameters are maintained.
    - **Worker Process:** The total number of work processes used in the application.
    - **Threads:** The total number of threads used in the application.
    - **Number of tasks:** The total number of tasks used in the application.
- **Time Metrics:** Provide data on the execution time of the program. Except for `Computation Time`, the smaller the value the better.
    - **Total time [h:min:sec:ms]:** Application duration time in HOUR:MINUTE:SECOND:MILLISECOND format.
    - **Parallel Time [h:min:sec:ms]:** Total accumulated parallel time, that is, the sum of the time that each thread of the work processes spent computing tasks.
    - **Computation Time [%]:** Percentage of parallel time in which the threads were computing, that is, the closer to 100 the better.
- **Efficiency Metrics:** measure the efficiency of the program regarding a certain parameter, with values ranging from 0 to 1. They are expected to be as close to 1 as possible, except for the **Serial Region Efficiency** metric. The **Computation Parallel Efficiency**, **Communication Parallel Efficiency**, and **Process Efficiency** are combined metrics, that is, they are the product of two other metrics.

  - **Computation Load Balance Efficiency:** Represents how well distributed the compute time of the tasks was to the compute threads
  - **Computation Efficiency:** Represents how much time the thread with the highest compute efficiency spent computing
  - **Computation Parallel Efficiency:** Represents the balance between Computation Load Balance Efficiency and Computation Load Balance Efficiency. It is expected that both values are reasonable for the computation to be considered efficient because even if a thread is very efficient, poor load balancing suggests that there is a lot of discrepancy between the most efficient and the least efficient.
  - **Communication Load Balance Efficiency:** Analogous to Computation Load Balance Efficiency, but referring to communication threads.
  - **Communication Efficiency:** Analogous to Computation Efficiency, but referring to communication threads
  - **Communication Parallel Efficiency:** Analogous to Computation Parallel Efficiency, but referring to communication threads
  - **Process Load Balance Efficiency:** Analogous to Computation Load Balance Efficiency, but the comparison is between processes and not threads.
  - **Process Communication Efficiency:** Analogous to Communication Efficiency, but the comparison is between processes and not threads.
  - **Process Efficiency:** Analogous to Computation Efficiency but referring to processes.
  - **Serial Region Efficiency:** Represents how much of the program time is spent in serial regions, i.e. for parallel programs it is expected to be a low value

A combined metric is more relevant than the metrics used in its calculation, as it represents a balance between them. For example, the Computation Load Balance Efficiency and Computation Efficiency are computed according to Equation 4.1 and Equation 4.2, respectively.

$$C_{LBeff} = \frac{avg(t_c)}{max(t_c)} \tag{4.1}$$

Equation 4.1: Computation Load Balance Efficiency: $\mathrm{t_c}$ is the computation time.

$$C_{eff} = max(\frac{t_c}{t_{parallel}}) \tag{4.2}$$

Equation 4.2: Computation Efficiency: $\mathrm{t_c}$ is the computation time and $\mathrm{t_{parallel}}$ is the parallel time.

The computation time is the sum of the duration of the target tasks, while the parallel time is the total computation duration on a certain thread. Figure 4.12 illustrates

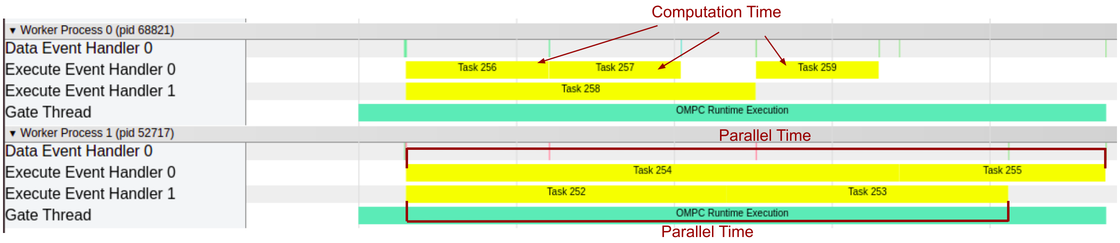


Figure 4.12: Example of calculation of computation and parallel time. The computation time of each thread is the accumulated value of the duration of the tasks. The parallel time is the elapsed time between the start of the first task and the end of the last executed task.

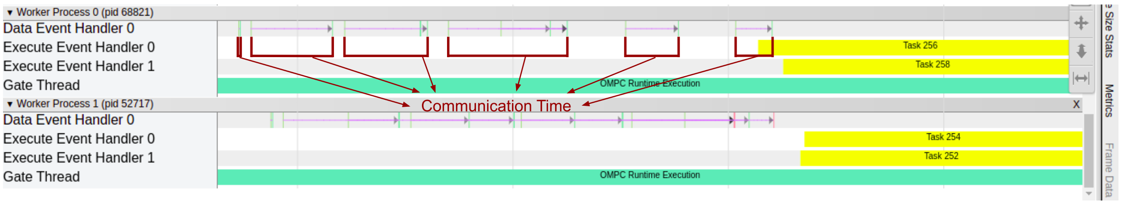


Figure 4.13: Example of calculation of communication time. As the tasks are very small, the image was zoomed in, so only some events are shown. The communication time for a thread is the sum of all time intervals, represented by the lines in red.

the difference between the two. That is, Computation Load Balance Efficiency does not take thread idle time into account, while Computation Efficiency does not take into account the imbalance between different threads since it only considers the maximum. So Communication Parallel Efficiency is more relevant as it considers both.

The calculation of the other metrics is similar. However, in the case of communication time, the tasks are separated into Begin and End events. Then, the algorithm calculates the time interval. This process is illustrated in Figure 4.13. Note that even time intervals of communication tasks tend to have a short duration when compared to the total time: Figure 4.13 is from the same application as Figure 4.12, but it is zoomed in to facilitate visualization.

4.5.2 Task

Task metrics provide statistical information about the execution and data tasks in the work processes of the OMPC application, described as follows:

- **Task Location:** The code location of the task in file-name:line:column format.
- **Individual Metrics:** They are computed using the individual duration of each task. In the case of data tasks, which are divided into Begin and End events, the duration is calculated as the time elapsed from the beginning of the Begin event to the end of the End event (see Figure 4.7). This metric is given both in absolute value (in the HOUR:MINUTE:SECOND:MILLISECOND format) and as a percentage of the total accumulated time.

  - **Avg:** The average duration of the task.
  - **Max:** The maximum duration of the task.
  - **Min:** The minimum duration of the task.

- **Count:** The number of tasks with a specific type and code location.
- **Sum:** The accumulated time of tasks with a specific type and code location. This metric is given both in absolute value (in the HOUR:MINUTE:SECOND:MILLISECOND format) and as a percentage of the total accumulated time.
- **Type:** Task event type, in Section 4.3.2 there are details of the meaning of each event. It can assume the following values according to its classification:
  - **Data Task:** The types of data tasks that metrics handle are: Retrieve, Submit, and Forward.
  - **Execution Task:** There is only one type of execution task, which takes the value "Execute".

4.5.3 Graph

The graph metric is calculated from the critical path of the application. It analyzes the longest time from each root node to the leaf nodes and chooses the longest among them as the critical path. The longest time is defined as the time spent on the task plus the time spent on dependent nodes and dependencies. The critical path is then set to the value 1.0 (100%) and the other longest paths as a percentage of that value. Figure 4.14 shows an example of calculating the critical path and longest times.

As the critical path dominates the parallel time, the objective of this metric is to estimate how much it would be possible to optimize the parallel time if it was optimized. Still, it is possible to use it to check imbalances in the task graph. In the example of Figure 4.14, there are two critical paths, with the same time. Thus, to improve the code it would be necessary to optimize both. If they were improved to the same time as the 0.8 metric path, then code time would be improved by 20%. On the other hand, it would be possible to use the metrics to determine if it is possible to delay the execution of a task that does not belong to the critical path, without delaying the execution of the program, which would be useful for saving computational resources, such as saving energy.

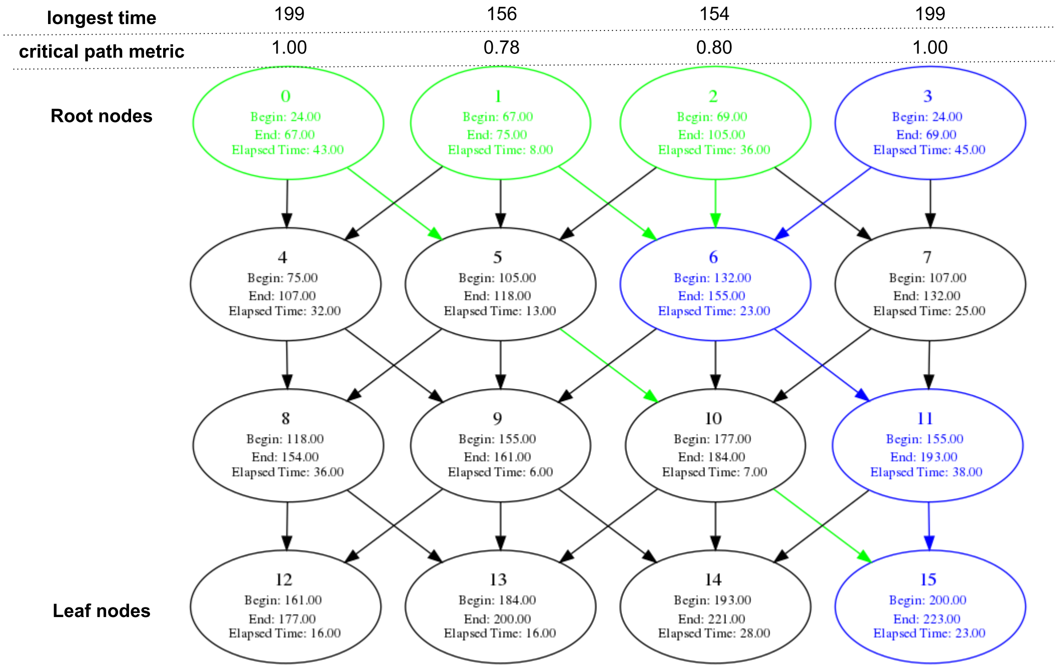


Figure 4.14: Example of a graph from the Task Bench application, with the kernel stencil, with the critical path metrics. For each leaf node (in green) a longer time is calculated. Among the longest times, the one that is greatest corresponds to the critical path, indicated by the blue nodes and edges, with a value of 1.0. In the other nodes, the percentage of this value is applied, converting to two decimal places. The edges in green indicate which would be the longest time path from the other root nodes.

Chapter 5

Experiments

This chapter presents the experiments conducted to evaluate the three sets of metric results. The analysis considers the unique characteristics of each application to interpret the data results. This chapter proposes to demonstrate how metrics can provide useful insights to guide optimizations and reflect the achieved performance. Section 5.1 details information about the applications and resources used. Section 5.2 provides an analysis of the timeline, while Section 5.3, Section 5.4, and Section 5.5 present experiments on general, task, and graph metrics, respectively.

5.1 Experimental setup

The experiments used applications from OMPC PLASMA and OMPC Task Bench in C/C++ programming languages. PLASMA [15] is a parallel linear algebra library with optimized routines for multicore architectures. Task Bench [31] is a parameterized benchmark, designed to explore the performance of parallel programming systems in various application scenarios. OMPC PLASMA [13] and OMPC Task Bench [3] are extensions of PLASMA and Task Bench libraries using OMPC, respectively.

OMPC PLASMA divides the matrix into blocks to parallelize the computation, with its size and block size parameterizable. The application is decomposed in target nowait tasks for each block that call BLAS functions to perform the computation. The Task Bench is a configurable benchmark that measures the performance of several distributed and parallel programming models, including OpenMP. It allows configuring different dependency patterns, as well as the number of tasks, number of interactions, load balancing, memory kernels, etc. OMPC Task Bench permits the use of all these functionalities with the OMPC runtime. The granularity of both applications, OMPC Plasma and OMPC Task Bench, is configurable, so they are suitable applications for performing experiments.

The experiments used Santos Dumont (SDumont) supercomputer [24]. SDumont is located at the National Laboratory for Scientific Computing (LNCC) in Petrópolis - Brazil, and, according to the TOP500 [32], it is among the most powerful supercomputers in the world. SDumont has processing capacity in the order of 5.1 petaflops/s (5,1 x 1015 float-point operations per second) with a total of 36,472 CPU cores, distributed across 1,134 computational nodes, mostly composed exclusively of CPUs with a multi-core architecture. However, there are heterogeneous nodes with accelerators that were not used in this project. SDumont uses Xeon Gold 6252 24C 2.1GHz processor and Mellanox InfiniBand EDR interconnection network between nodes. The nodes used for the experiments use this interconnection with 384Gb of RAM and RedHat Linux 7.6 Operation System.

The experiments were run through Singularity containers from OMPC Docker Hub, with Ubuntu 20.04 LTS Operation System. OMPC Task Bench uses runtime-dev and OMPC Plasma uses plasma-exp containers. As OMPC uses the MPICH implementation (version 4.0.2) with its UCX backend (version 1.8.0) was used. OMPC Plasma uses OpenBlas (version 0.3.20).

5.2 OMPC Plasma - Timeline Analysis

This section shows, through the timelines obtained from the experiment, how to extract useful information from these timelines and relate them to some metrics. The timeline user version rearranges, adds, and selects information from the developer version. Details about its events and characteristics are described in Section 4.3. It is common in a timeline analysis to search for specific events and arguments using the search menu or select arrows that link events.

An experiment was performed using the SGEMM kernel from OMPC Plasma to provide an analysis of the timeline. SGEMM is a block-based matrix multiplication of floating-point values, with single precision. The experiment used a matrix size of 32768 and a block size of 1024, 3 nodes and 17 threads (1 Control Thread, 8 Worker Threads, 2 Data Event Handlers, 2 Gate Threads, and 4 Execute Event Handlers). The generated user version timeline is shown in Figure 5.1.

Note that, without any zoom in the timeline, it is difficult to understand what is happening. The colors can serve as a guide for the user to visualize the repetition of events of the same type. For example, the Execute Event Handler threads of the work processes execute only the same type of task, colored in yellow. In the case of this application, this task represents matrix multiplication of the blocks. Between the execution of events, whether in the head or the work process, there will always be a blank space, which represents the idle time and depending on its duration can be relevant to the performance of the application. Without zooming in, it is almost impossible to visualize these blank spaces. However, even with zoom applied, it is difficult for the user to measure just by looking at whether these spaces are relevant to the application since there are 32768 tasks. The metrics help perform this analysis: Table 5.1 shows the time and efficiency metrics extracted from this application. Not all efficiency metrics were displayed, only those most relevant to the analysis. Through the Computation Time, we can infer that these blank spaces, in the work processes, represent only 7.8% of the total application time, which is a satisfactory result. The Computation Load Balance Efficiency is 0.9935, which can be verified when zooming in on the beginning and end of the timeline, as shown in Figure 5.2, the beginning and end times of the first and last tasks, in each thread, are very close. The first tasks of each thread all start in the interval between 41 and 41.5 ms, that is, with a time difference of less than 0.5 ms. The last tasks finish all with a difference smaller than

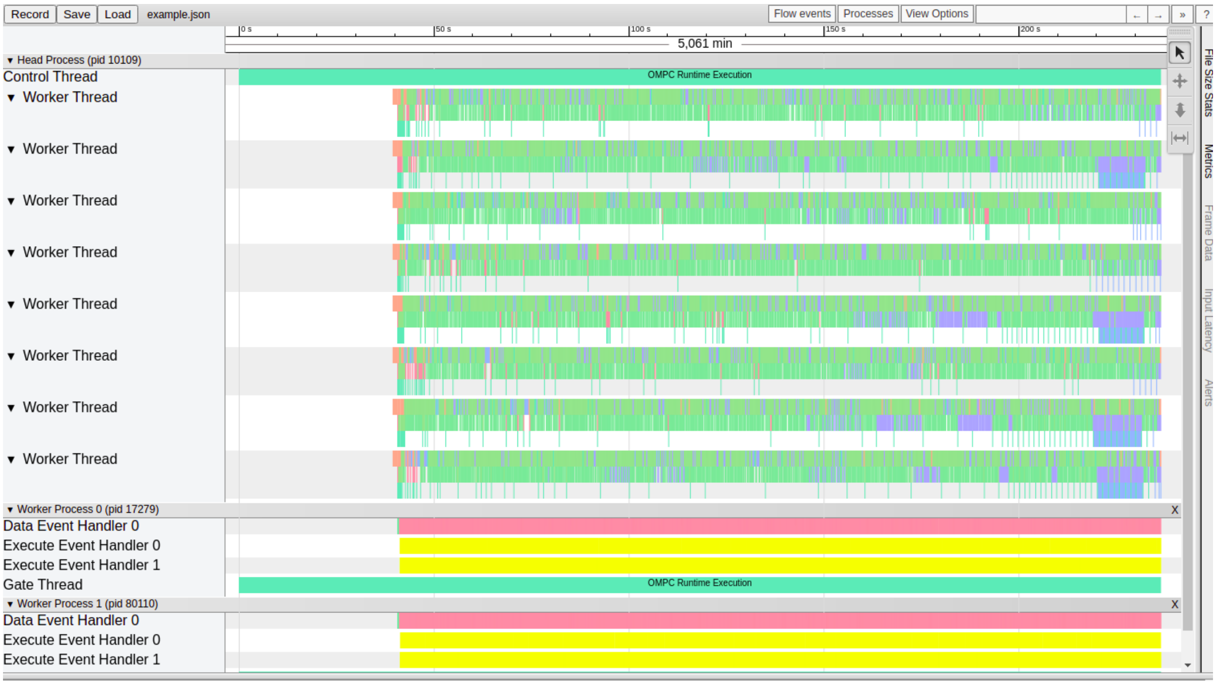


Figure 5.1: Timeline of SGEMM kernel of OMPC Plasma application, with a matrix size of 32768 and a block size of 1024. No zoom has been applied and all events are being shown.

Table 5.1: Time and efficiency metrics results for SGEMM experiment from OMPC Plasma, with a matrix size of 32768 and a block size of 1024.

Efficiency Metrics	**Total time [h:min:sec:ms]**	**Parallel Time [h:min:sec:ms]**	**Computation Time [%]**
	0:3:56:554	0:13:0:323	92.2
Time Metrics	**Computation Load Balance Efficiency**	**Communication Load Balance Efficiency**	**Communication Efficiency**
	0.9935	0.9987	0.0731

0.1 ms.

Regarding communication tasks, the analysis is different. In their case, it is expected a high idle time since these tasks are really fast. If there is a low idle time, it may indicate that there are delays in communication, which could delay the execution of tasks and affect performance. Based on the communication metrics, it is possible to estimate that only 7% of the time is spent in communication. This value is extracted from the Communication Efficiency which is 0.0731, as an estimate for the time spent in communication. The estimate is valid in this case since the communication threads present an excellent load balance, with a Communication Load Balance Efficiency of 0.9987. Note that from the non-zoomed view of the timeline, it is possible to infer that a large portion of the time of the Data Event Handler threads is spent communicating. However, in the zoomed-in Figure 5.3 a large amount of idle time in the communication threads is noticeable, coinciding with the result of the metrics. The arrows between the communication events are enabled, so between the blank spaces there is no communication taking place.

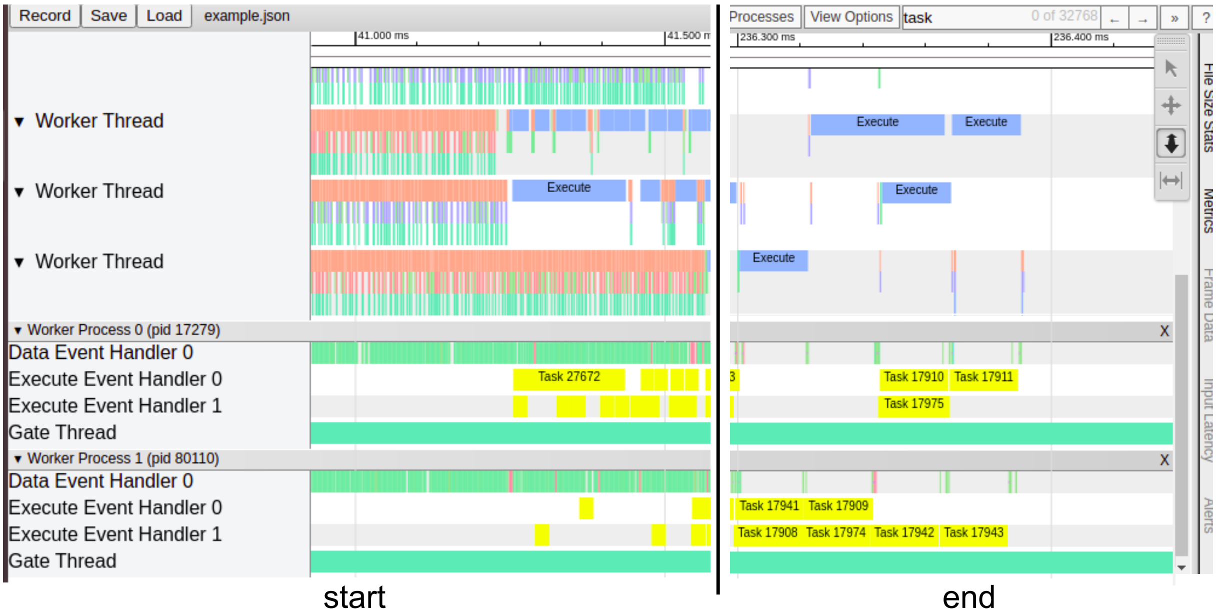
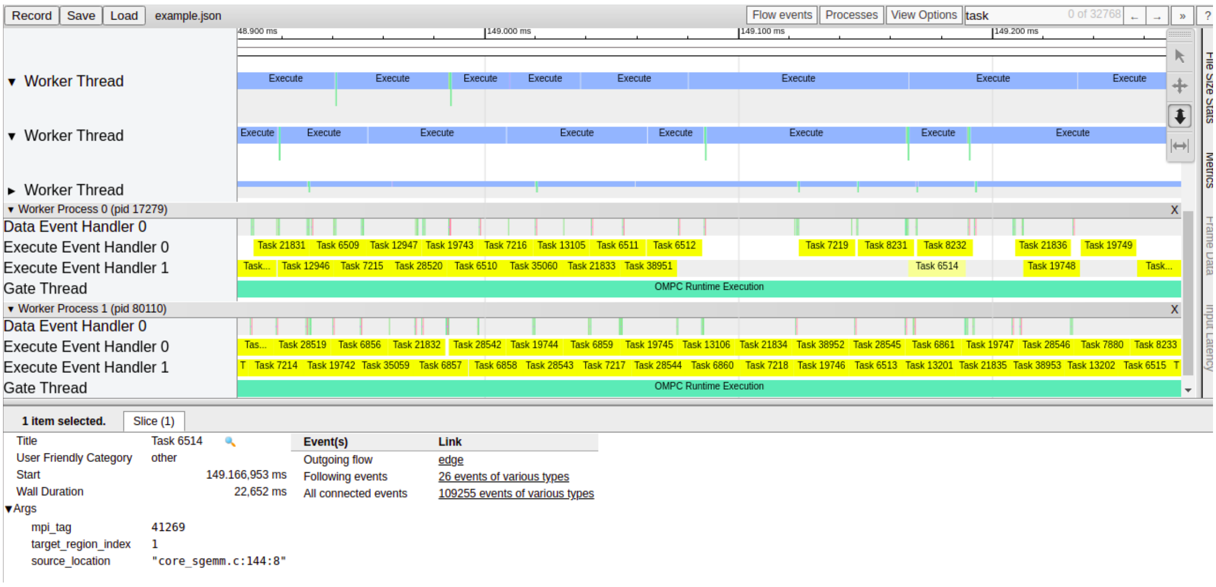


Figure 5.2: Timeline SGEMM with zoom applied at the beginning and end of the application. Where the left side represents the beginning time and the right side the end time, separated by a black line.

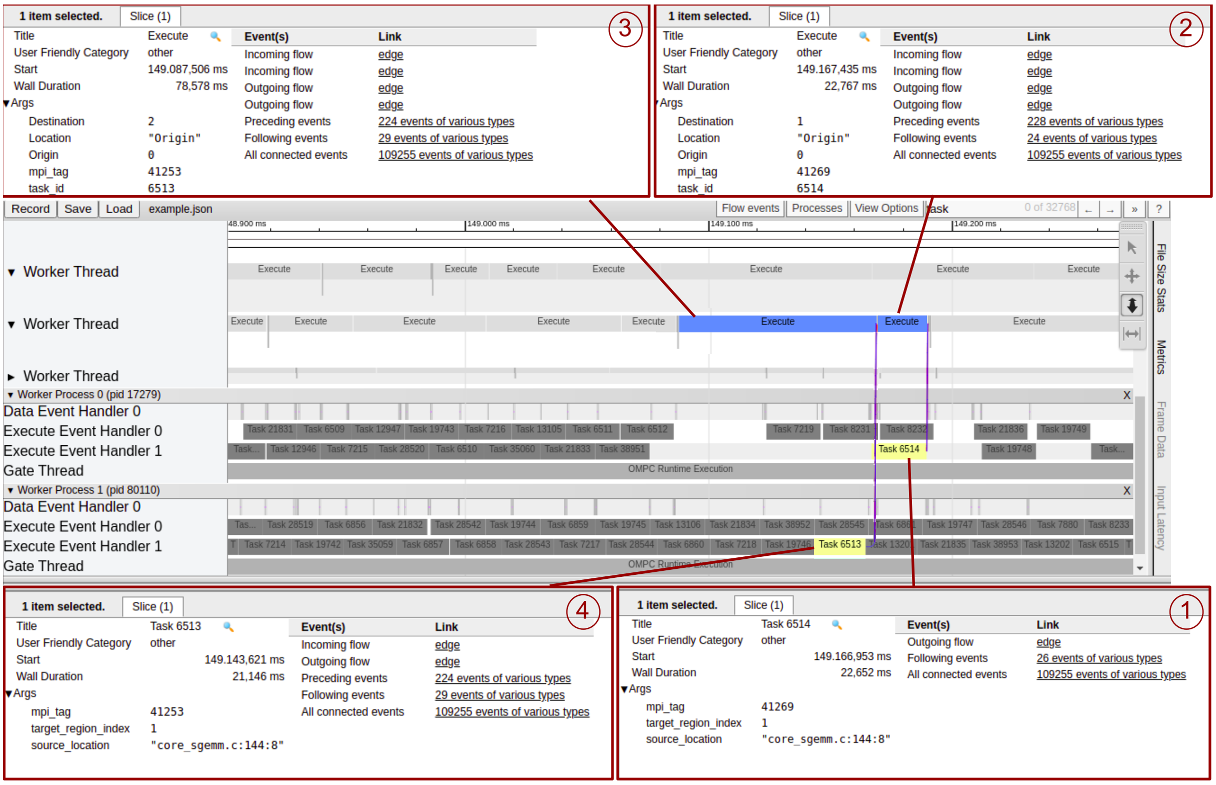


1 item selected. Slice (1)

Title	Task 6514
User Friendly Category	other
Start	149.166,953 ms
Wall Duration	22,652 ms
▾Args	
mpi_tag	41269
target_region_index	1
source_location	`"core_sgemm.c:144:8"`

Event(s)	Link
Outgoing flow	edge
Following events	26 events of various types
All connected events	109255 events of various types

Figure 5.3: Timeline SGEMM zoomed with task 6514 selected.



1 item selected. Slice (1)

		Event(s)	Link
Title	Execute	Incoming flow	edge
User Friendly Category	other	Incoming flow	edge
Start	149.087,506 ms	Outgoing flow	edge
Wall Duration	78,578 ms	Outgoing flow	edge
▾Args		Preceding events	224 events of various types
Destination	2	Following events	29 events of various types
Location	"Origin"	All connected events	109255 events of various types
Origin	0		
mpi_tag	41253		
task_id	6513		

1 item selected. Slice (1)

		Event(s)	Link
Title	Execute	Incoming flow	edge
User Friendly Category	other	Incoming flow	edge
Start	149.167,435 ms	Outgoing flow	edge
Wall Duration	22,767 ms	Outgoing flow	edge
▾Args		Preceding events	228 events of various types
Destination	1	Following events	24 events of various types
Location	"Origin"	All connected events	109255 events of various types
Origin	0		
mpi_tag	41269		
task_id	6514		

1 item selected. Slice (1)

		Event(s)	Link
Title	Task 6513	Incoming flow	edge
User Friendly Category	other	Outgoing flow	edge
Start	149.143,621 ms	Preceding events	224 events of various types
Wall Duration	21,146 ms	Following events	29 events of various types
▾Args		All connected events	109255 events of various types
mpi_tag	41253		
target_region_index	1		
source_location	"core_sgemm.c:144:8"		

1 item selected. Slice (1)

		Event(s)	Link
Title	Task 6514	Outgoing flow	edge
User Friendly Category	other	Following events	26 events of various types
Start	149.166,953 ms	All connected events	109255 events of various types
Wall Duration	22,652 ms		
▾Args			
mpi_tag	41269		
target_region_index	1		
source_location	"core_sgemm.c:144:8"		

Figure 5.4: Timeline SGEMM with some tasks selected to find dependencies, following these steps: (1) select task 6514 and use edge arg to find corresponding execution event in the head process; (2) select execution event and use edge arg to find dependent execution event; (3) select dependent execution event and use edge arg to find corresponding task in work process; (4) Task 6513 found.

When zooming the timeline, it is possible to see that in the execution threads, there are disproportionately higher idle times compared to other timeline points that execute the same tasks. Usually, these longer pauses are due to dependencies between tasks or data communication. In cases of high idle time, identifying the cause can be important for solving the problem. If related to task dependencies, it could come from task scheduling. Also, if related to data communications, it could be a problem in how the data is being transferred, in the connection between the nodes, or even a communication thread overload, requiring a greater number of Data Event Handler threads in each work process. Figure 5.3 shows an example of a disproportionate idle time between tasks 38951 and 6514. Figure 5.4 shows the process to identify which are the dependencies of task 6514, but this process can be applied to any task. After the inspection of the dependencies, it is verified that task 6514 depends on task 6513 and that 6514 is executed immediately after 6513 is finished, explaining the idle time.

Analyzing the head process helps to understand how the code behaves over time as it has control events. For example, its execution events offload tasks to another process. Figure 5.5 shows the disposition of the task events at the beginning of the application, most of them are for allocation (Alloc) and sending data (Submit) to the work processes. Figure 5.6 displays the behavior during most of the execution, in which there are many execution events (Execute) and among them some data forwarding (Forward) and deletion

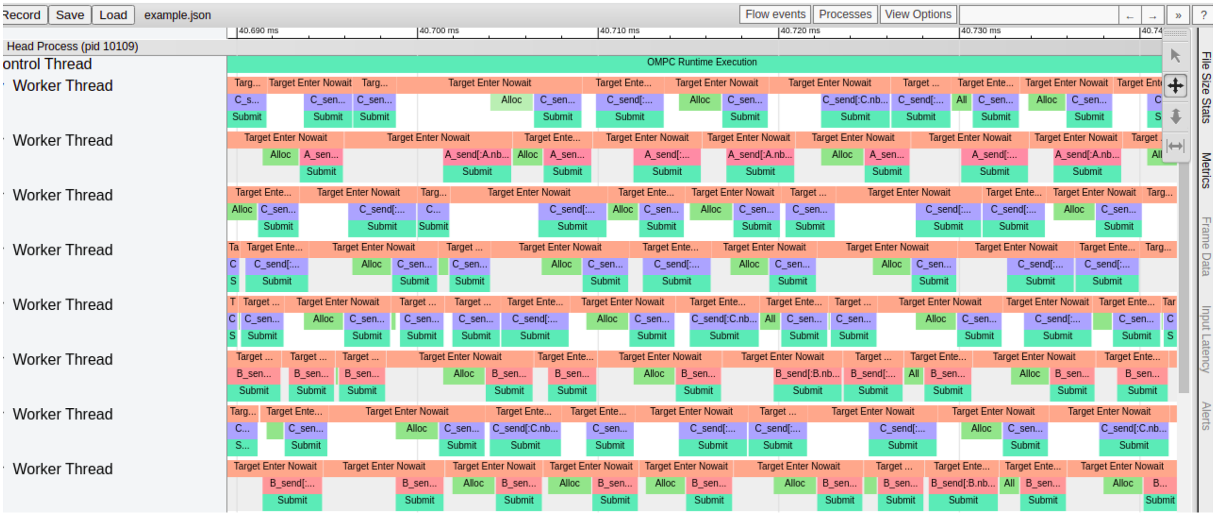


Figure 5.5: Timeline SGEMM with head process begin events.

events (Delete). Finally, at the end of the execution, shown in Figure 5.7, there are still many execution events, but with a greater amount of data deletion events.

From the analysis of the timeline, it is possible to understand the behavior of the code and how the OMPC runtime creates the events. In addition, it is possible to identify problems in the application and points that can be improved. With the extraction of metrics, the analysis becomes even more detailed, making it possible to measure parallel performance.

5.3 OMPC Plasma - General Metrics

The general metrics provide quantitative, timing, and efficiency information of the application, described in Section 4.5.1. They were tested using the SGEMM (general matrix-matrix multiplication) and SPOTRF (positive definite triangular factorization) kernels of OMPC Plasma. These tests intend to relate better performance with better metric results. The best values for each metric are also described in the Section 4.5.1. The experiments were performed with 8 worker processes, that is 9 nodes, and 213 threads.The matrix size was fixed and the block size varied for each experiment, therefore the number of tasks and the time spent on each also varied and affect the metrics.

The SGEMM experiments use a matrix size of 120000 and vary the block size between 2000 (EXP 1), 5000 (EXP 2), and 10000 (EXP 3). Table 5.2 shows the quantitative and time metrics, with EXP 2 performing best and having the best time metric results. Table 5.3 shows efficiency metrics results, and despite some metrics being better in EXP 1 and EXP 2, the most relevant ones are better in EXP 2. For example, Computation Load Balance Efficiency is about 0.1, 0.99 and 0.95 for EXP 1, 2, and 3, respectively. The Computation Efficiency is about 0.86, 0.92 and 0.97 for EXP 1, 2, and 3, respectively. Finally, Computation Parallel Efficiency is about 0.85, 0.86, and 0.8 for EXP 1, 2, and 3, respectively. That is, Computation Load Balance Efficiency is superior in EXP 1 and Computation Efficiency in EXP 3, however, EXP 2 provides the best Computation Parallel Efficiency that represents the balance between the other two metrics (ie. the combined

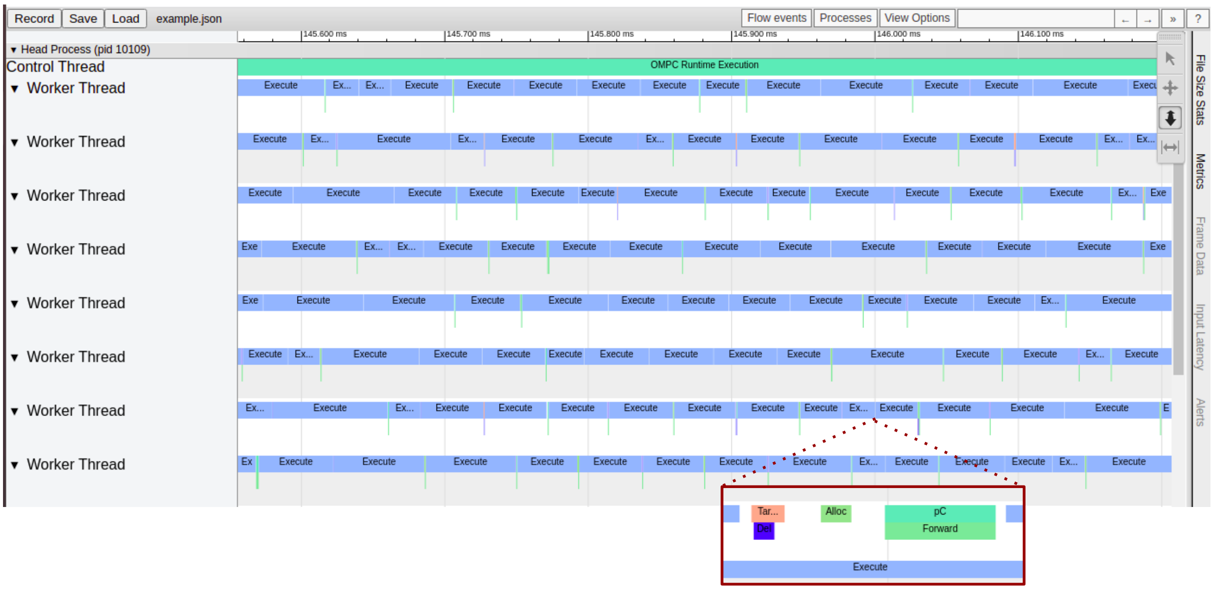


Figure 5.6: SGEMM timeline with events for most of the head process execution. The box in red shows the timeline events between two execution events.

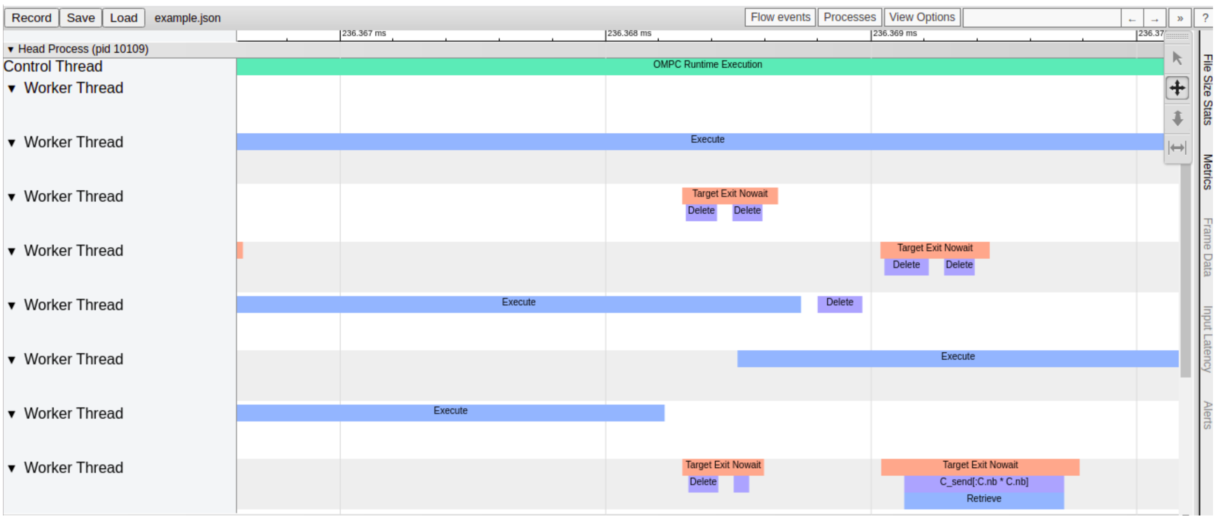


Figure 5.7: Timeline SGEMM with head process end events.

Table 5.2: Quantitative and time metrics results for SGEMM experiments from OMPC Plasma. Highlighted fields represent the best results for each time metric across the three experiments. There is no best result for quantitative metrics as they are fixed values related to the configuration set.

METRIC	**EXP 1**	**EXP 2**	**EXP 3**
Block Size	2000	5000	10000
Number of Tasks	216000	13824	1728
Total time [h:min:sec:ms]	0:9:21:831	**0:8:57:910**	0:10:26:250
Parallel Time [h:min:sec:ms]	12:14:16:591	**12:10:37:452**	13:46:21:961
Computation Time [%]	85.2	**86**	75

Table 5.3: Efficiency metrics results for SGEMM experiments from OMPC Plasma. Highlighted fields represent the best results for each metric across the three experiments.

METRIC	**EXP 1**	**EXP 2**	**EXP 3**
Computation Load Balance Efficiency	**0.9867**	0.9303	0.8244
Computation Efficiency	0.8657	0.9217	**0.9666**
Computation Parallel Efficiency	0.8542	**0.8575**	0.7969
Communication Load Balance Efficiency	**0.9960**	0.9899	0.9467
Communication Efficiency	0.2838	**0.3227**	0.1308
Communication Parallel Efficiency	0.2827	**0.3194**	0.1238
Process Load Balance Efficiency	**0.9865**	0.9843	0.9464
Process Communication Efficiency	0.9319	0.9704	**0.9719**
Process Efficiency	0.9193	**0.9552**	0.9198
Serial Region Efficiency	0.1016	0.1058	**0.0935**

metrics). As mentioned in Section 4.5.1, the combined metrics are more relevant that the metrics that compose them. Communication Parallel Efficiency and Process Efficiency have similar results. Serial Region Efficiency is better in Exp 3, though it is not too relevant to this application, since the purpose of this metric is to measure the portion of the execution that is serial, and most of the time is spent in the parallel region.

The SPOTRF experiments use a matrix size of 180000 and vary the block size between 2000 (EXP 1) and 5000 (EXP 2). Table 5.4 shows the quantitative and time metrics, with EXP 1 performing best and having the best time metric results. Table 3.3 shows efficiency metrics results, with EXP 1 performing better in most of them. Just like in SGEMM experiments, the worst results are not too relevant since it performs better in combined metrics and the Serial Region Efficiency is quite low.

In conclusion, the most relevant metrics performed better in the most efficient experiment, both for the SGEMM kernel and for the SPOTRF kernel. However, for SGEMM, the best block size choice was the one with intermediate granularity. As for SPOTRF, it was the case with finer granularity, and therefore with more tasks. This demonstrates that the choice of granularity directly affects performance and that its ideal value will vary according to the application's own characteristics. In addition, it is possible to verify that the application is already highly optimized. Except for the communication and serial metrics, where smaller values are expected, all other metrics have values greater than 0.8.

Table 5.4: Quantitative and time metrics results for SPOTRF experiments from OMPC Plasma. Highlighted fields represent the best results for each time metric across the two experiments. There is no best result for quantitative metrics as they are fixed values related to the configuration set.

METRIC	EXP 1	EXP 2
Block size	2000	5000
Number of Tasks	125580	8436
Total time [h:min:sec:ms]	**0:6:47:282**	0:7:22:430
Parallel Time [h:min:sec:ms]	**6:52:12:538**	7:40:58:537
Computation Time [%]	**86.4**	76.6

Table 5.5: Efficiency metrics results for SPOTRF experiments from OMPC Plasma. Highlighted fields represent the best results for each metric across the two experiments.

METRIC	EXP 1	EXP 2
Computation Load Balance Efficiency	**0.9769**	0.9151
Computation Efficiency	**0.8843**	0.8336
Computation Parallel Efficiency	**0.8639**	0.7628
Communication Load Balance Efficiency	**0.9905**	0.8959
Communication Efficiency	0.2486	**0.2614**
Communication Parallel Efficiency	**0.2462**	0.2342
Process Load Balance Efficiency	**0.9933**	0.9490
Process Communication Efficiency	0.9555	**0.9757**
Process Efficiency	**0.9491**	0.9259
Serial Region Efficiency	0.3162	**0.2735**

5.4 OMPC Plasma - Task Metrics

The task metrics are described in Section 4.5.2 and provide statistical information about execute and data tasks. The task metrics were tested using the SPOTRF kernel of OMPC Plasma. With 8 worker processes and 213 threads, the matrix size was fixed at 180000, and the block size varied between 2000 (EXP 1) and 5000 (EXP 2), therefore, the number of tasks and the time spent on each also varied. The intention of these tests is to evaluate how changing the input data, generating different amounts of task, changes the time of each task individually.

The results present the tasks separated by their source location, with average, maximum, minimum, and accumulated time expressed in percentages. Table 5.6 and Table 5.7 show the task metric results of EXP 1 and EXP 2, respectively. The most costly execution task is core_sgemm.c:144:8 for both cases and takes a significant amount of application time, about 83% for EXP 1 and 72% for EXP 2. Despite having a significantly larger amount of this task, even individually it is a costly task, also having the highest average, maximum, and minimum time values. Therefore, optimizing this task could generate significant performance gains. Although in this case, it is a call to the highly optimized GEMM kernel of the BLAS library. Also, when the block size changes from 2000 to 5000, there is an increase in the time spent on that task, which is to be expected. The variation of the block size values affects the application granularity, that is, the smaller the block size, the smaller the granularity, and consequently the greater the number of tasks. A finer granularity can provide a more parallelizable application, however, each task will generate an additional overhead that can be significant in the total application time. As in EXP 2, there are fewer tasks to perform the computation with the same matrix size the tasks take more time individually. Despite this, EXP 2 performs better, as its cumulative time is lower (ie, the sum of the time of all tasks), about 5 hours and 44 minutes for EXP 1 and 5 hours and 23 minutes for EXP 2. That is, the overhead generated by the largest number of tasks (EXP 1 has 110340 more core_sgemm.c:144:8 tasks than EXP 2) is relevant for this case.

The analysis of communication tasks is different from execution tasks. As shown in Figure 4.12, when a thread is executing an execution task, no other tasks are performed at the same time. However, as shown in Figure 4.13, several communication tasks are executed simultaneously, so the accumulated time does not reflect the time spent in the communication thread. Since, in the case of task metrics, the accumulation of times is considered individually, not the interval as in the OMPC metrics. So, it is better to analyze the time spent individually to estimate how much is being spent on data communication or to analyze the general communication metrics. In both experiments, the individual metrics (i.e. average, maximum, and minimum time) are substantially less than the time spent on executing tasks, at least 10 times smaller. Still, the results are well balanced, having little variation in the values for EXP 1 and a slightly greater variation in EXP 2, but still not relevant since the communication tasks are at least 10 times smaller for both EXP 1 and EXP 2. As in the execution tasks, EXP 2 also features longer communication tasks, which is to be expected as the data sent is larger due to its greater granularity.

Table 5.6: Task metrics for EXP 1 with a block size of 2000. The highlighted value represents the most costly execution task. The field names were abbreviated. SUM and COUNT represent, respectively, the accumulated task time and the number of tasks.

Task location	**AVG [m:ms]**	**AVG [10^4] %**	**MAX [10^3] %**	**MIN [10^4] %**	**SUM [h:m:s:ms]**	**SUM %**	**COUNT**	**TYPE**
core_ssyrk.c:111:9	0:300	0.0963	0.1094	0.0518	0:00:20:887	00.07	7666	Forward
core_sgemm.c:144:8	0:200	0.0862	0.1365	0.0509	0:08:09:663	01.73	200896	Forward
core_strsm.c:127:9	0:300	0.0895	0.1130	0.0518	0:00:16:671	00.06	6582	Forward
core_spotrf.c:85:9	0:200	0.0746	0.0113	0.0600	0:00:00:232	00.00	110	Forward
spotrf.c:276:13	0:200	0.0749	0.0949	0.0480	0:00:17:362	00.06	8190	Retrieve
spotrf.c:261:13	0:210	0.7452	0.2447	0.0659	0:02:52:624	00.61	8190	Submit
core_sgemm.c:144:8	**0:176**	**7.1038**	**1.5522**	**6.4061**	**5:43:40:938**	**83.46**	**117480**	**Execute**
core_strsm.c:127:9	0:900	3.6383	1.1634	3.2083	0:06:00:460	01.46	4005	Execute
core_ssyrk.c:111:9	0:920	3.7210	1.1756	3.3202	0:06:08:229	01.49	4005	Execute
core_spotrf.c:85:9	0:370	1.4960	0.3931	1.2213	0:00:03:327	00.01	90	Execute

Table 5.7: Task metrics for EXP 2 with a block size of 5000. The highlighted value represents the most costly execution task. The field names were abbreviated. SUM and COUNT represent, respectively, the accumulated task time and the number of tasks.

Task location	**AVG [m:ms]**	**AVG [10^2] %**	**MAX [10^2] %**	**MIN [10^2] %**	**SUM [h:m:s:ms]**	**SUM %**	**COUNT**	**TYPE**
core_sgemm.c:144:8	0:440	0.0140	0.0747	0.0088	0:08:21:955	1.6045	**11486**	Forward
core_ssyrk.c:111:9	0:450	0.0143	0.0516	0.0092	0:00:47:779	0.1527	1070	Forward
core_strsm.c:127:9	0:390	0.0125	0.0377	0.0092	0:00:34:500	0.1088	868	Forward
core_spotrf.c:85:9	0:370	0.0117	0.0170	0.0093	0:00:00:585	0.0019	16	Forward
spotrf.c:276:13	0:130	0.0042	0.0188	0.0027	0:00:17:696	0.0566	1332	Retrieve
spotrf.c:261:13	0:203	0.0650	0.1080	0.0103	0:04:30:667	0.8652	1332	Submit
core_sgemm.c:144:8	**2:718**	**1.0117**	**1.1696**	**0.9321**	**5:23:26:959**	**72.2380**	**7140**	**Execute**
core_ssyrk.c:111:9	1:392	0.5181	0.6119	0.4762	0:14:36:812	3.2637	630	Execute
core_strsm.c:127:9	1:372	0.5105	0.6480	0.4699	0:14:24:940	3.2164	630	Execute
core_spotrf.c:85:9	0:479	0.1784	0.2068	0.1533	0:00:17:257	0.0642	36	Execute

Table 5.8: Graph metrics of the random nearest kernel from Task Bench. The base case corresponds to EXP 1, highlighted in the table. The number in front of the critical path time, max critical path metric, and min critical metrics path metric represents how many times the same metric value appeared.

EXP	dependencies	radix	period	fraction	critical path time [ms]	max critical path metric	min critical path metric	average	median
1	***576***	**1**	**3**	**0.25**	**1 x 25553.43**	**1 x 0.92**	**1 x 0.23**	**0.56**	**0.66**
2	*696*	2	3	0.25	1 x 22606.70	3 x 0.82	1 x 0.31	0.57	0.51
3	*880*	2	3	0.50	1 x 21604.24	4 x 0.98	1 x 0.80	0.92	0.93
4	*1048*	2	3	0.80	1 x 23859.01	2 x 0.96	4 x 0.79	0.89	0.91
5	*698*	2	16	0.25	3 x 19108.32	4 x 0.98	1 x 0.65	0.87	0.85
6	*840*	2	16	0.50	1 x 21211.92	3 x 0.97	2 x 0.82	0.89	0.86
7	*1032*	2	16	0.80	1 x 45840.89	1 x 0.97	4 x 0.48	0.76	0.82
8	*701*	2	24	0.25	1 x 18608.63	4 x 0.99	1 x 0.74	0.93	0.94
9	*846*	2	24	0.50	1 x 28903.79	1 x 0.98	1 x 0.56	0.78	0.76
10	*1027*	2	24	0.80	1 x 22574.53	4 x 0.99	1 x 0.91	0.97	0.98

5.5 OMPC Task Bench - Graph Metrics

Graph metrics tests used the Task Bench application since there are several parameters to be configured that change how the dependencies are generated. The type of dependency used was random nearest and spread since they generate random dependency patterns and graphs that are more different from each other by varying the parameters. The varied parameters were the radix, which changes the number of dependencies per task, the period of dependency pattern, and the fraction of connected dependencies. The experiments were executed using 4 worker processes and 23 threads.

Table 5.8 shows the results obtained for the random nearest dependency pattern, divided by experiments numbered from 1 to 10. For these tests, radix 1 was used as the base case, and radix 2 was used to change the fraction and period parameters since they showed changes in the graph for radix greater than 1. Changing the parameters generated graphs with the same number of tasks (576), but with different numbers of dependencies. In total, each experiment had about 24 root nodes. so with also 24 longest times

In the base case, one dependency is generated per task. This is also the case where there is a greater imbalance between tasks, with a min and max critical path metric of 0.23 and 0.92, respectively, with the median value greater than the average indicating that there is a greater proportion of close values. to max. The greatest imbalance, in this case, is expected, since as the radix increases the number of dependencies increases, and as the number of tasks remains the same, the paths end up getting longer.

This decrease in unbalance is also observed for the other experiments when the fraction value increases since it also increases the number of dependencies. This can be seen in EXP 2, 3, and 4, for example, where EXP 2 has a greater imbalance with a fraction of 0.25, and EXP 3 and EXP 4, with a fraction of 0.5 and 0.8, respectively, are much more balanced, with similar values of min, max, mean and median.

Increasing the period does not significantly change the number of dependencies. From EXP 2 to EXP 5 for example, the period increased from 3 to 16, but the dependencies only increased by 2 units. But the longer the period, the more balanced the graph. This can be verified by analyzing EXP 2, 5, and 8. In which the min, max, average, and median measures all increased with larger period values.

Table 5.9: Graph metrics of the random spread kernel from Task Bench. The number in front of the critical path time, max critical path metric, and min critical metrics path metric represents how many times the same metric value appeared.

EXP	**dependencies**	**radix**	**critical path time [ms]**	**max critical path metric**	**min critical path metric**	**average**	**median**
1	*576*	1	1 x 11193.90	1 x 0.94	1 x 0.66	0.81	0.82
2	*1152*	2	1 x 26619.37	8 x 0.99	2 x 0.90	0.96	0.95
3	*2304*	4	1 x 43269.38	1 x 0.99	6 x 0.96	0.97	0.97

Critical path optimization would not be very useful in experiments where the max critical path metric is greater than or equal to 0.96, as its optimization could reduce up to 4% of the time. One of the worst cases to optimize for is EXP 5, which has 3 longest times in proportion to the critical path (note that the values are rounded, so it doesn't mean that there are three paths with the same measure of time, but that the values are very similar). Also, there are 4 max critical paths in the value of 0.98, even if it were possible to optimize the 3 critical paths, the application time would reduce at most by 2%. EXP 1 would benefit the most from a critical path reduction, as it could reduce application time by up to 8%.

Table 5.9 shows the results obtained for the random spread dependency pattern, divided by experiments numbered from 1 to 3. For these tests, the objective was to analyze how changing the radix affected the critical path, so the fraction and period were fixed at values of 0.25 and 3, respectively, default values used by Task Bench. The number of tasks are also a fixed value of 576.

Compared to the nearest kernel, in kernel spread raising the radix significantly increases the number of dependencies. The same configuration is used in EXP 2 of nearest and spread, but in the nearest kernel, the dependencies grew to 696, while for the spread it grew to 1152. Furthermore, in the kernel spread the dependencies were increased proportionally to the radix number. In EXP 2 they doubled and in EXP 3 they quadrupled, with a radix of 2 and 4 respectively.

The graphs generated by the kernel spread experiment are also much more balanced, with very similar average and median values for each experiment, which vary at most by 0.01. In addition, these metrics also increase with a growing radix, further improving balance. Because the graph is well balanced, optimizing the critical path leads to little performance gain. Only EXP 1 could have a more significant gain, with a max critical path metric of 0.94. EXP 2 is the worst case where there are 8 paths of value 0.98.

The graphs are very large and difficult to visualize. Figure 5.8 and Figure 5.9 show the graph using kernel spread and nearest with 72 and 48 dependencies, respectively. For both cases, the number of tasks is 36, with a radix of 2, a period of 3, and a fraction of 0.25. As expected, as it has a greater number of dependencies, the spread graph appears more connected. The graphs are difficult to analyze and are quite smaller than the tested examples. In such cases, the graph metrics can help the user to extract relevant information from the application's graphs.

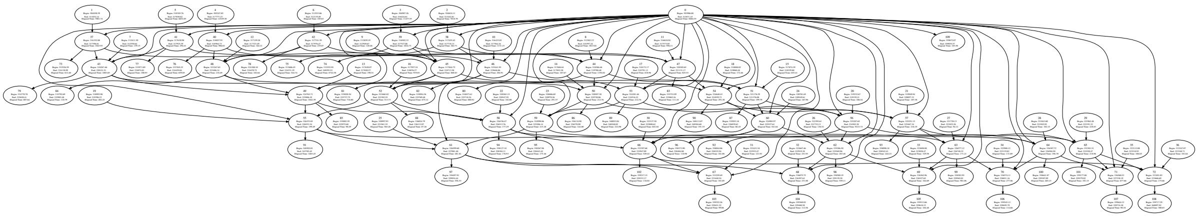


Figure 5.8: Random Spread Graph.

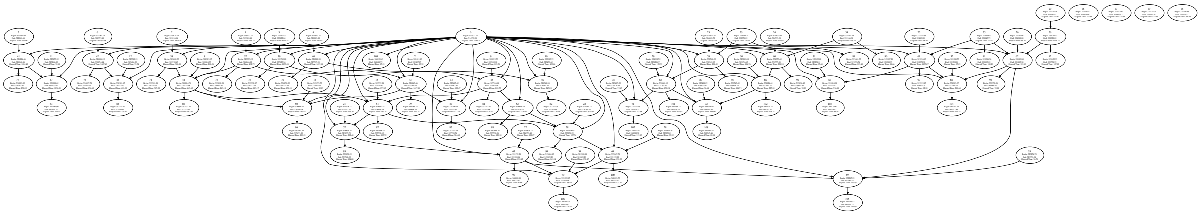


Figure 5.9: Random Nearest Graph.

Chapter 6

Conclusion

This work proposed the study and development of new specific profile features for applications based on task parallelism and running on HPC clusters. It aimed to meet the demands of such applications due to the lack of these features in current profiling tools. OMPC is a runtime that allows the implementation of applications with tasks distributed across computer nodes of clusters and was chosen to test the new profile features developed. This runtime permits simple code writing through OpenMP directives and was a suitable choice for development work as it provides task-related profiling data. This data was used as a basis for the creation of profile features with specific information for task programming.

The new features consisted of improving the timeline view and the implementation of profile metrics. The improved timeline, called user version timeline, has task dependencies, variable and source code information, and event links to help user analysis. The profile metrics were divided into three categories: general, task, and graph; providing statistics about program execution, specific task information, and critical path analysis, respectively.

The experiments carried out aimed to check the validity of the metrics and promote insights on how to use them to guide possible code optimizations. The general metrics experiments were tested using SGEMM and SPOTRF kernels of the OMPC Plasma application. It was verified that the best metrics results coincided with better performance results and that the metrics can help with overhead analysis generated by choice of granularity. The task metrics experiments were tested using the SPOTRF kernel and showed how they can be useful to identify possible optimization points in the program. Also, we described how to analyze execution and communication tasks. The graph experiments are performed with OMPC Task Bench and showed that the analysis of the critical path and the other longest time paths can help the user understand whether improving the critical path would bring significant performance gains. Timeline analysis allows to understand the behavior of the code and how events are executed, in addition to being able to help detect possible problems in the application. The analysis performed in conjunction with the metrics permits to evaluate the performance of the code.

The intent of metrics is for the programmer to use them to optimize the code. That is, after writing the parallel code, executing it, and obtaining the profile data, the user would then extract the metrics, identifying possible improvements to later verify if these actually reduced the application time. However, to carry out tests of this type it would be necessary to develop a new application, but due to lack of time, it was not possible to perform such experiments. Despite this, the analyzes realized in existing applications can serve as a guide for other applications. Metrics and improvements in the timeline provided more detailed information about the execution of tasks, fulfilling the research objective.

6.1 Lessons learned

As explained in Section 4.2, OMPC traces are generated in JSON format, due to its simplicity and compatibility with the Chrome Trace visualization tool. However, this decision also comes with limitations and complications. For applications with large data sizes that generate a significant number of tasks and events, the resulting trace size can be in the order of GB, which presents scalability challenges, especially in HPC applications.

Large traces also lead to long post-processing times for metrics extraction, trace merging, and trace improvement. Even a simple merge operation into a developer version trace that involves reading the trace, synchronizing it, and writing a new file can take several hours. Although synchronization is a simple process that involves iterating over events twice to find the lowest time and subtracting it from all event times, more complex operations such as trace improvement or metric extraction that require additional event iterations can significantly increase post-processing times. Several attempts were made to optimize this process by using different libraries for handling JSON or by implementing alternative versions of code using Python's Pandas library, but none of these solutions proved to be highly effective.

Although the Chrome Trace visualization tool was useful for avoiding the need to develop a new graphical interface, it also has limitations in terms of adding new information and visualization formats to the timelines.

6.2 Future works

As future works, the two most important points are: to carry out experiments with a new application and then use the metrics to optimize it; improve metrics extraction time, and trace improvement and merge. For the second point, two approaches were thought that can be performed concomitantly. The first is to use parallelism to perform operations that require iterating over events, since many functions of metric extraction and timeline improvement are independent. However, this approach would not solve the problem of long trace reading time. During the project, there were tests with several libraries for reading JSON, but even the most efficient library tested still took a very long time to read. So, the second approach would be to change the trace format to one that reads faster. We are studying the possibility to use the Perfetto Tracing SDK [2] in the future, a library in which it is possible to generate traces in a binary format. It contains its own graphics tool, called Perfetto UI, accessible via the web site, which has a visualization very similar to our current timeline model. The trace format generated by this library can be converted into the format read by the Chrome Trace visualization tool, as well as other tools. We expect that with traces in binary format, reading is much faster and improves the scalability of our profiling tools.

Bibliography

[1] Chrome Trace browser extension documentation. `https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/preview`, 2017. (Date last accessed 1-April-2023).

[2] Perfetto Tracing SDK. `https://perfetto.dev/docs/instrumentation/tracing-sdk`, 2020. (Date last accessed 1-April-2023).

[3] Task Bench extension with OMPC. `https://gitlab.com/ompcluster/task-bench`, 2020. (Date last accessed 1-April-2023).

[4] CARLA - Latin America High Performance Computing Conference. `https://carla22.org/index.html`, 2022. (Date last accessed 1-April-2023).

[5] CUDA Basic Linear Algebra Subroutine library from NVIDIA. `https://docs.nvidia.com/cuda/cublas/`, 2022. (Date last accessed 1-April-2023).

[6] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[7] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[8] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2012.

[9] Shajulin Benedict, Ventsislav Petkov, and Michael Gerndt. Periscope: An online-based distributed performance analysis tool. In *Tools for High Performance Computing 2009*, pages 1–16. Springer, 2010.

[10] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[11] Sally Bridgwater. Performance optimisation and productivity centre of excellence. In *2016 International Conference on High Performance Computing & Simulation (HPCS)*, pages 1033–1034. IEEE, 2016.

[12] David Böhme, Felix Wolf, Bronis R. de Supinski, Martin Schulz, and Markus Geimer. Scalable critical-path based performance analysis. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 1330–1340, 2012.

[13] Carla Cardoso, Hervé Yviquel, Guilherme Valarini, Gustavo Leite, Rodrigo Ceccato, Marcio Pereira, Alan Souza, and Guido Araujo. An openmp-only linear algebra library for distributed architectures. In *2022 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pages 17–24. IEEE, 2022.

[14] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[15] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Panruo Wu, Ichitaro Yamazaki, Asim Yarkhan, Maksims Abalenkovs, Negin Bagherpour, Sven Hammarling, Jakub Šístek, David Stevens, Mawussi Zounon, and Samuel D. Relton. Plasma: Parallel linear algebra software for multicore using openmp. *ACM Trans. Math. Softw.*, 45(2), May 2019.

[16] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurrency and computation: Practice and experience*, 22(6):702–719, 2010.

[17] William Gropp, William D Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[18] Laxmikant V Kale and Sanjeev Krishnan. Charm++ a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, 1993.

[19] Laxmikant V Kale and Amitabh Sinha. Projections: A preliminary performance tool for charm. In *Parallel Systems Fair, International Parallel Processing Symposium*, pages 108–114. Citeseer, 1993.

[20] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. The vampir performance analysis tool-set. In *Tools for high performance computing*, pages 139–155. Springer, 2008.

[21] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, et al. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011*, pages 79–91. Springer, 2012.

[22] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.

[23] Gustavo Leite. OMPC Bench a command-line benchmarking tool for ompcluster with taskbench., 2021.

[24] MCTI LNCC. SDumont supercomputer., 2021.

[25] Jonathan R. Madsen, Muaaz G. Awan, Hugo Brunie, Jack Deslippe, Rahul Gayatri, Leonid Oliker, Yunsong Wang, Charlene Yang, and Samuel Williams. Timemory: Modular performance analysis for hpc. In Ponnuswamy Sadayappan, Bradford L. Chamberlain, Guido Juckeland, and Hatem Ltaief, editors, *High Performance Computing*, pages 434–452, Cham, 2020. Springer International Publishing.

[26] Adrian Munera, Sara Royuela, Germán Llort, Estanislao Mercadal, Franck Wartel, and Eduardo Quiñones. Experiences on the characterization of parallel applications in embedded systems with extrae/paraver. In *49th International Conference on Parallel Processing - ICPP*, ICPP '20, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. Paraver: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-18: transputer and occam developments*, volume 44, pages 17–31. Citeseer, 1995.

[28] Vitoria Pinho, Hervé Yviquel, Marcio Machado Pereira, and Guido Araujo. Omptracing: Easy profiling of openmp programs. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 249–256. IEEE, 2020.

[29] Vinícius Garcia Pinto, Lucas Leandro Nesi, Marcelo Cogo Miletto, and Lucas Mello Schnorr. Providing in-depth performance analysis for heterogeneous task-based applications with starvz. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 16–25. IEEE, 2021.

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

[31] Elliott Slaughter, Wei Wu, Yuankun Fu, Legend Brandenburg, Nicolai Garcia, Wilhem Kautz, Emily Marx, Kaleb S Morris, Qinglei Cao, George Bosilca, et al. Task bench: A parameterized benchmark for evaluating parallel runtime performance. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[32] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer of Prometeus. TOP500 the list., 2021.

[33] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[34] C.-Q. Yang and B.P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *[1988] Proceedings. The 8th International Conference on Distributed*, pages 366–373, 1988.

[35] Herve Yviquel, Márcio Machado Pereira, and Guido Araujo. OMPCluster, Cluster Programming Made Easy!

[36] Keren Zhou, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu. Gvprof: A value profiler for gpu-based clusters. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.