Integration of High-Performance Task-Based In Situ for Molecular Dynamics on Exascale Computers

Estelle Dirand

▶ To cite this version:

Estelle Dirand. Integration of High-Performance Task-Based In Situ for Molecular Dynamics on Exascale Computers. Computer Science [cs]. Université Grenoble - Alpes, 2018. English. NNT : . tel-01949170v1

HAL Id: tel-01949170
https://hal.science/tel-01949170v1

Submitted on 9 Dec 2018 (v1), last revised 8 Mar 2019 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTE UNIVERSITE GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Estelle DIRAND

Thèse dirigée par **Bruno RAFFIN**, Directeur de Recherche, Inria Grenoble Rhône-Alpes
codirigée par **Laurent COLOMBET**, Chargé de Recherche, CEA DAM Île de France

préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et Technologies de l'information, Informatique**

Développement d'un système in situ à base de tâches pour un code de dynamique moléculaire classique adapté aux machines exaflopiques

Integration of High-Performance Task-Based In Situ for Molecular Dynamics on Exascale Computers

Thèse soutenue publiquement le **6 novembre 2018**,
devant le jury composé de :

Monsieur Bruno RAFFIN
Directeur de Recherche, Inria Grenoble Rhône-Alpes, Directeur de thèse
Monsieur Laurent COLOMBET
Chargé de Recherche, CEA DAM Île de France, Co-directeur de thèse
Monsieur Gabriel ANTONIU
Directeur de Recherche, Inria Rennes - Bretagne Atlantique, Rapporteur
Monsieur Hank CHILDS
Associate Professor, University of Oregon, Rapporteur
Monsieur Christophe CALVIN
Chargé de Recherche, CEA Saclay, Examinateur
Madame Violaine LOUVET
Ingénieur de Recherche, GRICAD - Grenoble Alpes Recherche - Infrastructure de Calcul Intensif et de Données, Examinateur
Monsieur Raymond NAMYST
Professeur, Université de Bordeaux, Président du jury

Communauté UNIVERSITÉ Grenoble Alpes

FROM RESEARCH TO INDUSTRY cea

Doctoral Thesis of

Communauté Université Grenoble Alpes

Integration of High-Performance Task-Based In Situ for Molecular Dynamics on Exascale Computers

Estelle Dirand

Under the supervision of
Laurent Colombet and Bruno Raffin

Commissariat à l'énergie atomique et aux énergies alternatives (Cea)
Centre Dam-Île de France – Bruyères-le-Châtel, 91297 Arpajon Cedex
Département de Physique Théorique et Appliquée
☎ +33 (0)1 69 26 40 00

Pour Papa, qui m'a donné la force de continuer
malgré les difficultés

Remerciements

Il y a trois ans je suis arrivée au CEA pour commencer cette grande aventure qu'est la thèse et maintenant que la soutenance est passée, il est grand temps de prendre quelques lignes pour remercier toutes les personnes que j'ai croisées en chemin et qui ont contribué de près ou de loin à l'aboutissement de cette aventure.

Je voudrais tout d'abord remercier les membres du jury pour avoir accepté de juger mon travail : Raymond Namyst pour avoir accepté d'être le président du jury, Gabriel Antoniu et Hank Childs pour avoir accepté la lourde tâche de rapporteurs et pour leurs commentaires plus que positifs sur mon manuscrit et Christophe Calvin et Violaine Louvet pour avoir accepté le rôle d'examinateurs et pour leurs questions intéressantes et leurs commentaires positifs. Une petite pensée pour Frédéric Desprez qui n'a finalement pas pu être présent à cause de la SNCF.

Une thèse n'est rien sans les encadrants qui l'accompagnent, c'est pourquoi je voudrais remercier Bruno Raffin et Laurent Colombet de m'avoir suivie durant ces trois années et de m'avoir poussée à aller toujours plus loin. J'ai beaucoup appris grâce à vous pendant cette thèse, à la fois sur les aspects techniques mais aussi sur les aspects transverses tels que la rédaction, la présentation orale et aussi et surtout sur comment prendre du recul sur mon travail. Je vous en suis évidemment très reconnaissante. Merci aussi à Bruno pour m'avoir fait confiance et m'avoir recommandée pour le poste chez Total. Et promis Laurent, je n'oublie pas qu'il faut que je sois plus optimiste et plus fière de mon travail. Mais par contre, j'attends toujours les madeleines avec une belle bosse ! Je tiens aussi à remercier Thierry Carrard pour tous ses conseils pendant les derniers mois de thèse. Tu m'as beaucoup apporté d'un point de vue technique et tu as su me motiver pour commencer la rédaction de ce manuscrit au moment où je n'arrivais pas à m'y mettre. Et je n'oublierai pas non plus les heures de débugage à rajouter des accolades partout dans le code.

Je voudrais ensuite remercier les physiciens du service pour avoir accueilli une informaticienne parmi vous. Un grand merci à Laurent Soulard pour les cours de physique des chocs, de l'écaillage et du micro-jetting et à Nicolas Pineau, Olivier Durand et Jean-Bernard Maillet pour les brainstormings de début de thèse sur les besoins en in situ. Une mention particulière pour Claire Lemarchand avec qui je n'ai pas eu beaucoup l'occasion de travailler directement mais qui a su m'écouter au moment où la thèse était un peu dure. Merci aussi à François Jollet de m'avoir accueillie dans le service et un énorme merci à Brigitte Flouret et Sandra Boullier pour avoir grandement facilité tous les aspects administratifs.

Pour tous leurs conseils et leurs idées durant cette thèse, je souhaite remercier mes collègues de Paratools, Jean-Baptiste Besnard, Julien Adam et Antoine Capra. J'ai aimé les réunions que nous avons faites pour faire émerger les idées, les expérimentations git avec Julien et les séances de débugage avec Jean-Baptiste.

L'heure est maintenant venue de remercier les stagiaires, thésards et post-docs qui ont partagé leur quotidien avec moi pendant une partie ou l'intégralité de ces trois ans. Je voudrais donc remercier Ahmed, Aloïs, David, Emmanuel, Gérôme, Giovanni, Ioannis, Jean, Jean-Baptiste, Jean-Baptiste, Jean-Charles, Jordan, Loïc, Luc, Lucas, Luis, Nils, Paul, Quentin, Raphaël, Richard, Sami, Théo, Thibaud, Tristan, Xavier et tous ceux que j'ai oublié de citer. Une mention spéciale pour Paul, Thibaud, Jean-Baptiste, Hugo et Hugo, mes compagnons de thèse, à qui je souhaite une bonne fin de thèse pour ceux qui n'ont pas encore soutenu et une bonne continuation pour ceux qui sont déjà de l'autre côté du tunnel. Durant cette thèse, j'ai eu l'occasion d'avoir deux super co-bureau, Jean-Charles pendant la première année et Raphaël (HippopoRaph ?) pour les deux années d'après. J'ai vraiment apprécié partager mon bureau avec vous et je vous remercie chaleureusement pour les discussions techniques et un peu moins techniques. Une petite dédicace à Gérôme et Lucas pour les soirées et après-midi jeux, à David pour les paris et les gâteaux qui allaient avec et à Jordan qui n'était pas un non-permanent mais qui a toujours fait partie de notre groupe et à qui je souhaite tout le meilleur pour sa nouvelle vie en Belgique.

Finalement, je voudrais remercier mes amis, ma famille et ma belle-famille. Merci à Céline et Léa d'être mes amies depuis le collège et la primaire respectivement et d'avoir toujours été là pour les moments durs et les moments moins durs. Céline, je ne te remercierai jamais assez pour toutes nos conversations remontage de moral et évacuation de stress. Merci aussi aux amis de l'ENSTA, en particuliers Pauline, Catherine et Marie. Et un grand merci à Jacky, Michèle, Riri, Jacky, Margot et Mamie qui ont fait le déplacement depuis la Haute Saône et l'Alsace pour venir m'écouter présenter ma thèse en anglais.

Je tiens à remercier de tout mon cœur Maman pour avoir été là depuis toujours et pour m'avoir toujours soutenue même quand il a fallu pour cela que je parte loin de la maison. Je ne te remercierai jamais assez pour tout ça. Et aussi un énorme merci pour être venue m'aider à préparer le pot de thèse et pour avoir passé plus de 10h avec moi dans la cuisine à couper des patates et à faire des kougelhopf. Sans toi, j'aurais été beaucoup plus stressée par la préparation !

Richard, je garde un paragraphe spécialement pour toi. Comme promis, je te remercie pour m'avoir aidé à préparer le pot et surtout pour le kilo de comté que tu as râpé (presque pour rien) pour les salades. Mais bien entendu tu as fait beaucoup plus que ça. Je te remercie du fond du cœur d'être rentré dans ma vie il y a maintenant deux ans et de me combler de bonheur depuis ce jour. Tu es à la fois mon meilleur ami, mon confident et l'homme de ma vie et je ne pourrais jamais assez te remercier pour tout le soutien que tu m'as apporté pendant ces deux années et en particuliers pour les derniers mois de la thèse. Merci d'avoir relu mon manuscrit et de m'avoir aidé à l'améliorer, merci pour ton aide pour la préparation de la soutenance et merci d'avoir été là et de l'être toujours pour me soutenir, me conseiller et me faire rire.

Abstract

The exascale era will widen the gap between data generation rate and the time to manage their output and analysis in a post-processing way, dramatically increasing the end-to-end time to scientific discovery and calling for a shift toward new data processing methods. The in situ paradigm proposes to analyze data while still resident in the supercomputer memory to reduce the need for data storage. Several techniques already exist, by executing simulation and analytics on the same nodes (in situ), by using dedicated nodes (in transit) or by combining the two approaches (hybrid). Most of the in situ techniques target simulations that are not able to fully benefit from the ever growing number of cores per processor but they are not designed for the emerging manycore processors. Task-based programming models on the other side are expected to become a standard for these architectures but few task-based in situ techniques have been developed so far.

This thesis proposes to study the design and integration of a novel task-based in situ framework inside a task-based molecular dynamics code designed for exascale supercomputers. We take benefit from the composability properties of the task-based programming model to implement the TINS hybrid framework. Analytics workflows are expressed as graphs of tasks that can in turn generate children tasks to be executed in transit or interleaved with simulation tasks in situ. The in situ execution is performed thanks to an innovative dynamic helper core strategy that uses the work stealing concept to finely interleave simulation and analytics tasks inside a compute node with a low overhead on the simulation execution time.

TINS uses the Intel® TBB work stealing scheduler and is integrated into ExaStamp, a task-based molecular dynamics code. Various experiments have shown that TINS is up to 40% faster than state-of-the-art in situ libraries. Molecular dynamics simulations of up to 2 billions particles on up to 14,336 cores have shown that TINS is able to execute complex analytics workflows at a high frequency with an overhead smaller than 10%.

Résumé

L'ère de l'exascale creusera encore plus l'écart entre la vitesse de génération des données de simulations et la vitesse d'écriture et de lecture pour analyser ces données en post-traitement. Le temps jusqu'à la découverte scientifique sera donc grandement impacté et de nouvelles techniques de traitement des données doivent être mises en place. Les méthodes *in situ* réduisent le besoin d'écrire des données en les analysant directement là où elles sont produites. Il existe plusieurs techniques, en exécutant les analyses sur les mêmes nœuds de calcul que la simulation (in situ), en utilisant des nœuds dédiés (in transit) ou en combinant les deux approches (hybride). La plupart des méthodes in situ traditionnelles ciblent les simulations qui ne sont pas capables de tirer profit du nombre croissant de cœurs par processeur mais elles n'ont pas été conçues pour les architectures many-cœurs qui émergent actuellement. La programmation à base de tâches est quant à elle en train de devenir un standard pour ces architectures mais peu de techniques in situ à base de tâches ont été développées.

Cette thèse propose d'étudier l'intégration d'un système in situ à base de tâches pour un code de dynamique moléculaire conçu pour les supercalculateurs exaflopiques. Nous tirons profit des propriétés de composabilité de la programmation à base de tâches pour implanter l'architecture hybride TINS. Les workflows d'analyses sont représentés par des graphes de tâches qui peuvent à leur tour générer des tâches pour une exécution in situ ou in transit. L'exécution in situ est rendue possible grâce à une méthode innovante de *helper core* dynamique qui s'appuie sur le concept de vol de tâches pour entrelacer efficacement tâches de simulation et d'analyse avec un faible impact sur le temps de la simulation.

TINS utilise l'ordonnanceur de vol de tâches d'Intel® TBB et est intégré dans ExaStamp, un code de dynamique moléculaire. De nombreuses expériences ont montrées que TINS est jusqu'à 40% plus rapide que des méthodes existantes de l'état de l'art. Des simulations de dynamique moléculaire sur des système de 2 milliards de particules sur 14,336 cœurs ont montré que TINS est capable d'exécuter des analyses complexes à haute fréquence avec un surcoût inférieur à 10%.

Contents

1 Introduction **17**

1.1 On the Path Toward Exascale Supercomputers 17
1.2 Need for In Situ Processing of Simulation Data 18
1.3 Thesis Objectives 19
1.4 Thesis Contributions and Organization 20

I Molecular Dynamics and I/O Challenges for Exascale **21**

2 Background on Data Analytics on Supercomputers **23**

2.1 Architecture of Supercomputers 23
2.1.1 Evolution of the Compute Node Architecture 23
2.1.2 Node Interconnect 29
2.1.3 Filesystem and I/O 31
2.2 In Situ Processing 33
2.2.1 Synchronous In Situ 33
2.2.2 Over-Subscription of the Cores 35
2.2.3 Core Separation 36
2.3 In Transit and Hybrid Processing 38
2.3.1 In Transit Processing 38
2.3.2 Hybrid Processing 40
2.4 In Situ Workflows Control 42
2.5 Chapter Summary 44

3 Task-Based Molecular Dynamics for Exascale Computers **45**

3.1 Intel® TBB, a Task-Based Runtime 45
3.1.1 Task Creation with TBB API 46
3.1.2 TBB Resource Management 49
3.1.3 Tools to Control the Task Execution 52
3.2 ExaStamp, a Molecular Dynamics Code for Material Sciences 53
3.2.1 Molecular Dynamics for Material Sciences 54
3.2.2 ExaStamp Architecture 54

3.3 Challenges for the Integration of an In Situ Framework Inside ExaStamp 58
3.3.1 Target Architectures 59
3.3.2 ExaStamp Performance 60
3.3.3 Ideas for the Implementation of a Task-Based Hybrid Framework 61

II Toward a Task-Based In Situ Technique 63

4 Turning a Synchronous In Situ into an Asynchronous Task-Based In Situ 65
4.1 Integration of Analytics for Synchronous Execution 65
4.1.1 Integration of Synchronous In Situ in ExaStamp 66
4.1.2 Implementation of Analytics Routines inside ExaStamp 66
4.1.3 Comparison of the Synchronous In Situ and the File Output Approaches 68
4.2 Highlighting Periods of Unused Resources 70
4.2.1 Implementation of a Task Monitoring System 70
4.2.2 Measure of the Thread Usage in the Synchronous Approach 70
4.3 Derivation of a Task-Based Asynchronous IN Situ Approach (TINS) 71
4.3.1 Spawning of an Analytics Task 73
4.3.2 Evaluation of TINS compared to a Synchronous Execution 75
4.4 Chapter Summary 76

5 Implementation of a Thread Isolation to Improve TINS Performance 77
5.1 Evaluation of TINS compared to the Goldrush Process-Based Approach 77
5.1.1 Usage of Goldrush on the Cobalt Supercomputer 77
5.1.2 Instrumentation of ExaStamp with Goldrush API 78
5.1.3 Comparison of TINS and Goldrush 80
5.2 Evaluation of TINS compared to the Damaris Static Helper Core Approach . . . 81
5.2.1 Instrumentation of ExaStamp with Damaris API 81
5.2.2 Comparison of TINS and Damaris 85
5.3 Implementation of a Thread Isolation Mechanism in TINS 86
5.3.1 Separation of the Tasks into Disjoint Arenas 87
5.3.2 Implementation of an Analytics Master Thread 87
5.3.3 Comparison of the Two Versions of TINS and Damaris 91
5.3.4 Highlighting the Limitations of the Static Helper Core Approach 93
5.4 Chapter Summary 95

6 Implementation of a Dynamic Helper Core Strategy with Automatic Sizes 97
6.1 Implementation of an Adaptive Static Helper Core Approach 97
6.1.1 Design of the Algorithm 97
6.1.2 Highlighting the Limitations of the Approach 99
6.2 Implementation of a Dynamic Helper Core Strategy with a Temporary Isolation 100
6.2.1 Designing a Temporary Thread Isolation with TBB 100
6.2.2 Implementation of the Temporary Thread Isolation in TINS 102
6.2.3 Evaluation of the Dynamic Helper Core Approach 105
6.3 Implementation of an Adaptive Dynamic Helper Core Approach 109
6.3.1 Design of the Algorithm 110

6.3.2 Validation of the Approach . . . 111
6.3.3 Highlighting the Limitations of the Approach . . . 112
6.4 Chapter Summary . . . 113
6.5 Part Summary . . . 114

III Toward an Evolutive Task-Based Hybrid Framework 115

7 Design of a Framework to Automatically Orchestrate Analytics Execution 117
7.1 Orchestration of Simulation and Analytics Codes . . . 117
7.1.1 Integration of TINS Architecture in the Simulation Code . . . 118
7.1.2 Development of Analytics Outside of the Simulation as TINS Plugins . . . 119
7.1.3 Compilation and Loading of TINS Plugins . . . 121
7.2 Automatic Creation of a Graph of Plugins . . . 122
7.2.1 Definition of the Analytics Workflow . . . 122
7.2.2 Construction of a Graph of Plugins . . . 123
7.2.3 Management of Simulation and Analytics Data . . . 126
7.3 Extension of TINS with an In Transit Mode . . . 129
7.3.1 Design of an In Transit Mode . . . 129
7.3.2 Implementation of a Prototype and Preliminary Results . . . 130
7.3.3 Execution of Analytics Plugins in a Standalone Mode . . . 132
7.4 Chapter Summary . . . 133

8 Validation of TINS on a Production Run 135
8.1 Description of the Physics and the Analytics Workflow . . . 135
8.1.1 Computation of the Steinhardt Parameters . . . 136
8.1.2 Definition of the Analytics Workflow . . . 139
8.2 Limitations of the Tera-1000-2 Supercomputer . . . 141
8.2.1 Disabling of the OS Scheduler on the KNL Nodes . . . 141
8.2.2 Temporary Absence of the `MPI_THREAD_MULTIPLE` Threading Level . . . 141
8.3 Validation of TINS . . . 142
8.3.1 Numerical Validation . . . 142
8.3.2 Preliminary Performance Measurements . . . 144
8.3.3 Gain of TINS for the Physicists . . . 148
8.4 Chapter Summary . . . 149

9 Conclusion and Perspectives 151
9.1 Contributions . . . 151
9.2 Perspectives . . . 153
9.3 Towards Advanced Uses of TINS . . . 154

IV Additional Content 157

10 Résumé de la Thèse en Français 159
10.1 Introduction . . . 159
10.2 Organisation du Manuscrit . . . 161

10.3 Conclusion 164

Bibliography **165**

1 Introduction

1.1 On the Path Toward Exascale Supercomputers

To understand physical phenomena, physicists design models and perform physical or numerical experiments to validate, invalidate or refine their models. Physical experiments are performed in laboratories where the environment of the physical phenomena at stake is reproduced. For example, extreme conditions can be reproduced by lasers or energetic environments by particle accelerators. Physical experiments help the physicist understand finely the physics at stake but they have limitations. The first limitation is financial, the use of advanced facilities and advanced monitoring tools making physical experiments very expensive. In the car industry for example, performing one crash test can cost from hundred of thousands to millions dollars, which limits the number of crash tests that a manufacturer can perform within a given envelope. The second limitation is the reproducibility of some physical phenomena. Indeed, some phenomena cannot be reproduced in a laboratory, as this is the case for example when studying the movement of planets or galaxies in astrophysics or the interaction of oceans and the atmosphere in climate science.

During the past few decades, the use of numerical simulations has been democratized to complement physical experiments. Physical phenomena are modeled by mathematical equations solved by scientific simulations running on high performance supercomputers. A simulation usually alternates two phases: a computing phase where the computing resources of the supercomputer are used to solve the numerical equations and a writing phase where the state of the system is periodically extracted and sent through high performance networks to persistent storage. Simulation data are written into files that are later read back by analytics codes in a *post-processing* step necessary for converting raw simulation data into metrics that can be used by physicists to understand the physical phenomena at stake.

The desire to have more and more accurate simulations in a reasonable time has driven the development of high performance supercomputers into complex architectures with different levels of memory and parallelism. Several nodes are interconnected into a distributed memory architecture and the nodes are in turn composed of several cores sharing memory. The number of cores per node has increased over the years, from the *multicore* nodes with a few number of cores to *manycore* nodes with several dozens of cores. In June 2018, the most powerful supercomputer,

Summit[1], was composed of more than 4,000 nodes for a total of 2 millions cores, reaching 10^{17} floating point operations per seconds (namely one hundred million billions floating point operations per second). This trend towards more computing resources is not likely to stop and the supercomputers manufacturers have already begun a race toward the exascale with 10^{18}, namely one billion of billions, floating point operations per second. The first exascale supercomputers are expected to be released around year 2020.

By accessing more computing resources, physicists are able to simulate more complex phenomena. In the field of molecular dynamics applied to material sciences, the study of the phase transition of a material under shock has been until now limited to systems of a few million particles. With the advance of supercomputers, the systems can now have hundreds of millions of particles, leading to a better understanding of the phenomenon. However, this also leads to an increase in the size of the data produced by numerical simulations. While the exascale era will bring more computational capabilities, recent studies have shown that the rate at which files will be written and read will not keep up the rate at which they will be produced [8]. Simulations will produce more and more data because they will be able to increase the sizes of their systems, but the supercomputers networks will not be able to follow the data generation rate, which will greatly impact simulation performance and the end-to-end time to scientific discovery.

The end-users of current petaflop supercomputers are already facing this Input/Output (I/O) bottleneck and they often get around the problem by simply decreasing the output frequency. This way, the pressure of I/O on the simulation performance is reduced but this has a critical impact in the scientific discovery process. The frequency of data output should be chosen according to the physics at stake, some physical phenomena occurring on a few iterations while others need a large number of iterations. When the frequency of output is reduced for performance reasons, the physicists are likely to miss physical phenomena and to re-execute the simulation to catch it. This increases the time until scientific discovery and the number of allocated resources.

1.2 Need for In Situ Processing of Simulation Data

To perform data analytics at a higher frequency without saving more data into the filesystem, the *in situ* paradigm proposes to complement the post-processing approach by analyzing data while still resident in the compute node memory. Analytics are executed on-line with the simulation, reducing the need to write data into the storage system. Analytics are either executed on the same nodes than the simulation (in situ), on distinct nodes (in transit) or on a combination of the two (hybrid).

The more direct way to perform in situ analytics is called *synchronous* and consists in periodically stopping the simulation to execute analytics instead. Analytics codes are most often integrated into the simulation and can directly access simulation data. However, the end-to-end execution time is the addition of simulation and analytics times and heavy analytics have a significant impact on the end-to-end execution time. This end-to-end execution time of in situ analytics can be reduced thanks to *asynchronous* in situ techniques that optimize the resource usage on a compute node. One asynchronous approach relies on *time sharing*. Simulation and analytics codes are executed on the same cores and coarse-grained scheduling approaches are used to alternate between simulation and analytics executions more efficiently than syn-

[1] https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/

chronous approaches. Another asynchronous approach adopts a *space sharing* strategy where analytics codes are executed concurrently with the simulation on dedicated cores called *helper cores*. While these approaches have proved their performance for multicore architectures and for simulation codes that are not able to fully benefit from all the cores of the nodes, they are not designed for manycore architectures and they do not target simulation codes optimized for these architectures. Indeed, they rely on coarse-grained scheduling approaches that do not allow efficient execution of analytics codes during the short periods when the simulation does not use all the cores, leaving an opportunity for further performance improvements.

With the advanced of manycore processors, task-based programming models [22] are currently emerging as a standard for the future exascale supercomputers. They propose high levels tools for the programmers to describe their programs as a set of tasks. The tasks are then executed on the compute cores in a transparent way thanks to fine-grained schedulers. This approach is designed so that simulation codes can fully benefit from multi and manycore processors and it provides good composability properties. It is indeed possible to create tasks without taking care of other portions of codes or libraries that could create tasks simultaneously, the scheduler being in charge of transparently executing the different tasks. Task-based programming models can thus allow to have analytics tasks created concurrently with simulation tasks and interleaved on the same resources thanks to a shared scheduler. However, the potential of task-based programming for in situ processing has been seldom investigated so far.

1.3 Thesis Objectives

In this thesis, we study the integration of a task-based in situ framework inside a task-based molecular dynamics code designed for exascale supercomputers. The target code of our study is ExaStamp [29], a molecular dynamics code developed at CEA for the last 5 years. ExaStamp is designed for manycore processors and uses several levels of parallelism to achieve very good performance on modern architectures. It uses the Intel® Threading Building Blocks (TBB) library [5] that provides a task-based programming model for the creation of simulation tasks and a work stealing scheduler to efficiently balance the tasks execution on multi and manycore processors. A simulation like ExaStamp cannot take advantage of existing in situ techniques, either because it exhibits short sequential regions difficult to harvest or because removing cores to dedicate them to analytics has a significant impact on simulation performance.

The challenges of this thesis are multiple. As a first goal, we need to propose task-based in situ methods that create and execute analytics tasks concurrently with the simulation. We will therefore need to find the best instant in the simulation execution to integrate in situ analytics and to design in situ methods that use the work stealing concept to efficiently interleave simulation and analytics tasks with a low overhead on the simulation execution time. More generally, the goal of this thesis is to implement a task-based hybrid framework where complex analytics workflows can be executed transparently in situ, in transit or in an hybrid way. We will therefore need, as a second goal, to design a generic, intuitive and evolutive tool that uses a task-based programming model so that the users can easily develop analytics workflows as if they were post-processing codes and deploy them in situ or in transit transparently.

1.4 Thesis Contributions and Organization

This thesis is organized as follows. Chapter 2 recalls the basic concepts behind supercomputers and gives an overview of the existing tools developed for in situ, in transit and hybrid processing. Chapter 3 introduces in more details the TBB library and the ExaStamp molecular dynamics code and highlights the challenges of implementing a task-based in situ framework for such a simulation code.

Chapters 4, 5 and 6 detail the different steps toward the implementation of a dynamic task-based in situ strategy. Chapter 4 shows how a synchronous in situ approach can be turned into a more efficient task-based asynchronous approach. Chapter 5 shows how an isolation mechanism leads to the implementation of a task-based static helper core strategy equivalent or even better than the traditional helper core approach. Chapter 6 shows how the isolation mechanism can be adapted to implement a dynamic helper core strategy more efficient than the static helper core approach and than existing middleware. We will show in particular that the dynamic helper core strategy reduces the end-to-end time of in situ analytics compared to two state-of-the-art approaches and that the overhead of the approach is kept under 10% for high frequency analytics of large scale simulations on more than 14,000 cores. These results led to the publication of a paper in an international conference [37].

Chapters 7 and 8 detail the implementation and usage of a task-based hybrid framework. Chapter 7 shows how a graph-based plugin system separates simulation and analytics codes for the intuitive creation of task-based analytics workflows that can be executed in situ or in transit transparently. Chapter 8 validates the task-based framework on a production run. We will show in particular that this framework is robust, evolutive and generic and that the synchronizations between simulation and analytics are completely hidden by the framework to ease the development of analytics and the deployment of complex analytics workflows in a production environment and by non-expert developers.

Chapter 9 finally summarizes the contributions of this thesis and draws perspectives of this study.

Part I

Molecular Dynamics and I/O Challenges for Exascale

2 Background on Data Analytics on Supercomputers

Simulations usually alternate two phases: a *computing phase* where the computing resources are used to solve the mathematical equations and a *writing phase* where data are extracted from the computing resources to be saved in persistent storage system. These output data aim at being later analyzed and visualized to get insights into the physical process at stake. The desire to have more and more accurate simulations in a reasonable time has led to the development of high performance supercomputers with different levels of computing resources and efficient storage systems (Section 2.1). However, a growing gap between data generation rate and the time necessary to write data into the storage system has been observed for the last decades. Novel in situ techniques emerge to reduce the amount of data stored into persistent storage systems. They propose to analyze and visualize simulation data while still resident in the supercomputers, by creating middleware that use the resources of the simulation (Section 2.2) or dedicated resources (Section 2.3) to execute analytics. At a higher level, methods are also proposed to analyze or visualize simulation data with the smallest end-to-end execution time (Section 2.4).

2.1 Architecture of Supercomputers

A supercomputer is organized around a compute and memory hierarchical model. Several nodes are interconnected to form a distributed memory model and each node is in turn composed of several cores in a shared memory model. The compute nodes are then connected to a parallel filesystem through a network. The goal of this section is to present the different components of a supercomputer and the associated vocabulary: the compute node and the shared memory programming model (Section 2.1.1), the connection between the nodes and the distributed memory model (Section 2.1.2) and the way data are output to the filesystem (Section 2.1.3).

2.1.1 Evolution of the Compute Node Architecture

In the past decades, the compute node architecture has evolved from monocore processors with one core to multi and manycore architectures with several cores and a complex memory hierarchy. We explain in this section the compute node evolution and the tools that exist to make use of the node architecture.

Monocore Processors

A traditional monocore Central Processing Unit (CPU) can be represented schematically by the Figure 2.1. It is composed of a microprocessor (hereafter called *processor*) in charge of the computation, a volatile main memory such as Random Access Memory (RAM) to store the data and devices to handle Input/Output (I/O) operations to non volatile memories for example. These different components communicate with each other thanks to communication buses. The processor is itself composed of different elements. An Arithmetic and Logic Unit (ALU) performs the elementary computation operations. Registers and caches are different levels of memory inside a processor.

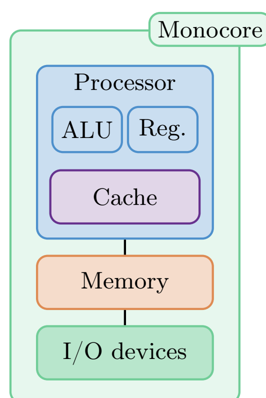

Figure 2.1 | Schematic view of a monocore processor.

The registers are used for the arithmetic operations. They have the fastest access but they are also the smallest in capacity. Some specific registers are also used for the Single Instruction Multiple Data (SIMD) paradigm or *vectorization*. In a loop iteration, instead of performing the same operation on different data one after the other, the SIMD paradigm proposes to pack data elements into arrays and to perform the operation on the array, hence performing the same operation on different data at the same time. Several extensions exist with different register sizes: SSE (128 bits), AVX (256 bits) and more recently AVX-512 (512 bits) that allows to perform vector operations on 8 double precision floating points at the same time.

Caches are fast memories used as an intermediate when it comes to access the main memory. When the processor needs to access data stored in the main memory, it first looks into the caches to see if these data are available at this level. If this is the case, a *cache hit* occurs and the processor directly uses data from the cache. Otherwise, a *cache miss* occurs and data need to be loaded into the caches. The *latency* is a measure of the speed of a cache and corresponds to the elapsed time between a data request and the time when data arrive. Because it is difficult to build caches that have both a large storage capacity and a low latency, the caches are decomposed in different levels (usually L1 to L3), the fastest level (L1) being the smallest in size and the slowest level (L3) being the largest in size (Table 2.1). When looking for data, the processor thus checks the presence of these data in every cache level before loading data from the main memory.

During the last three decades, the processors manufacturers have improved the processor performance thanks to three factors: the increase of the processor frequency, the increase of the size of the caches and several optimizations such as the instruction pipelining. Increasing the

Table 2.1 | Typical sizes and latencies for the different cache levels compared with an order of magnitude of size and latency of traditional RAM[1].

Level	Size	Latency
L_1	32 KB	1 ns
L_2	256 KB	4 ns
L_3	8 MB or more	40 ns
RAM	Several GB	80 ns

processor frequency was made possible by decreasing the size of the transistors that compose them. Gordon Moore observed in 1965 that the number of transistors in an integrated circuit doubles approximately every two years [88]. The *Moore's law* proved accurate for several decades and the simulation codes could directly benefit from the processor technology advances, gaining performance without any code modifications. However, the Moore's law is less and less applicable because of power consumption and heat dissipation issues. Herb Sutter declared in 2005 that *free lunch is over* [107] and that new technologies have to be found for achieving higher performance. He put forward *multi-threading* and *multicore processors* as approaches to get there. From that point, we observe an increase in the number of cores per processors on leadership supercomputers (Figure 2.2).

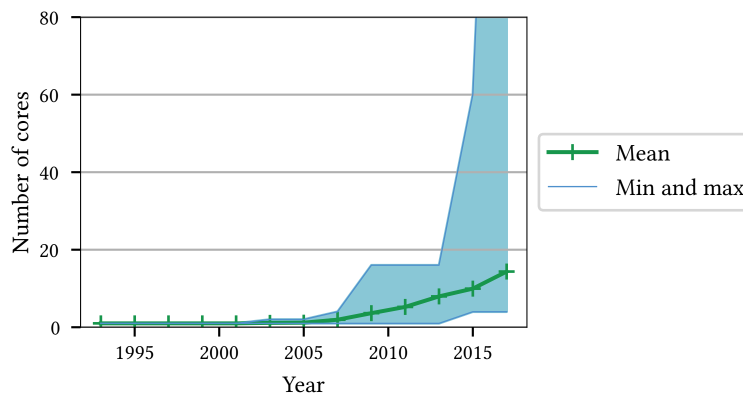

Figure 2.2 | Evolution of the number of cores per socket for the Top500 supercomputers [6]. The peak in 2014 denotes the use of manycore processors and the peak in 2015 shows the democratization of GPU usage with 260 cores per socket (truncated here for readibility issues).

Multicore Processors

The first step to multicore processors is the *Symmetric MultiProcessing* (SMP) where two or more processors are connected to a single and unique main memory (Figure 2.3). In this case, the processors are called *cores* and the term processor refers to the combination of the cores. The higher levels of caches are likely to be shared by several cores while there is often one L1 cache per core. The connection of the cores and the memory is made through a crossbar switch. This technology, also called *Uniform Memory Access* (UMA) technology, guarantees that all the cores

[1] https://software.intel.com/en-us/articles/memory-performance-in-a-nutshell

have a direct access to the main memory but the approach proved to be inefficient for large number of cores [31].

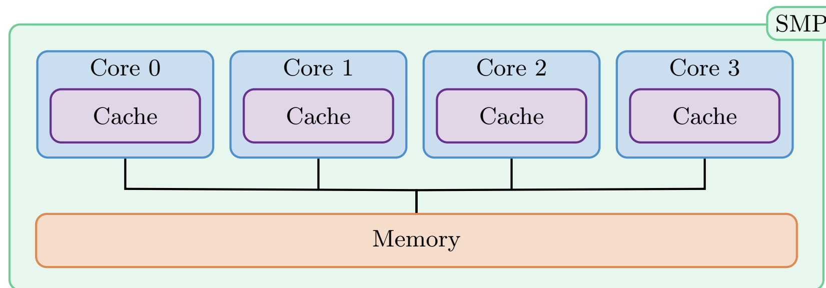

Figure 2.3 | Schematic view of a SMP multicore processor where 4 cores share the same memory.

An alternative is the *Non-Uniform Memory Access* (NUMA) approach where the processor is composed of two or more SMP-like nodes (Figure 2.4), called the NUMA nodes. The cores still share a common main memory but the memory is divided into blocks, one block per NUMA nodes. The cores of one NUMA node can access both the memory located in their NUMA node and the memory located in other NUMA nodes. However, the access is direct on their NUMA node while it goes through an interconnection network otherwise, leading to longer access times. Compared to a UMA approach, this technology reduces the number of cores that need to connect to the unique main memory. It increases the number of cores per processor but distant memory accesses are more expensive. The access to distant NUMA nodes is transparent to the users although several tools allow the users to control the data locality of an application to reduce NUMA effects. Specific protocols, such as *cache coherent NUMA* (ccNUMA), also exist to ensure that a modification of data in a particular cache is transmitted to the other cores so that they always work on the same data.

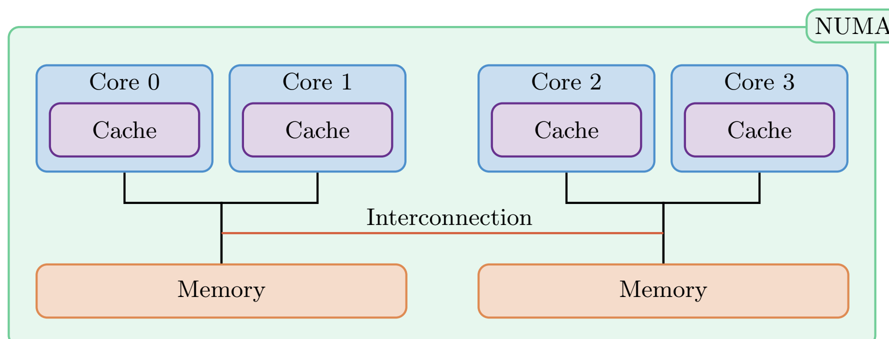

Figure 2.4 | Schematic view of a NUMA multicore processor where the 4 cores are split into 2 NUMA nodes with a block of the memory associated to each NUMA node. An interconnection network links the two blocks of memory.

Manycore Processors

The use of Graphics Processing Units (GPU) for general-purpose computing began popular in the 2000s, in particular to solve matrix-based problems. GPUs and CPUs highly differ in their architectures. The GPUs provide way more cores than the CPUs and they extensively use the

SIMD paradigm: the cores are grouped into *warps* and the cores of a warp execute the same instructions at the same time. GPU processing proved to be efficient for many algorithms [92] but requires specific algorithms and data management that limit its adoption in the scientific community.

Parallel effort to the development of GPUs has been the development of manycore processors with a large number of classical CPUs. In contrast to multicore processors that are designed to be efficient for both sequential and parallel applications, manycore processors have simpler cores running at a lower frequency and providing a high degree of parallelism for parallel applications but poorer performance for sequential applications. The idea of these architectures is that codes parallelized for multicore processors should benefit from manycore processors at (nearly) no cost. As an example, Intel® released in 2016 the Knights Landing (KNL) composed of 72 cores divided into 36 tiles (Figure 2.5). A tile corresponds to 2 cores with a shared cache, the cores being similar to classical cores but with a reduced frequency. The memory is composed of two parts, a large classical RAM memory and a MCDRAM memory that can be used as a cache or as a high speed extension of the RAM. The tiles are connected through a 2-dimension grid and the KNL can be configured to expose 1, 2 or 4 NUMA nodes.

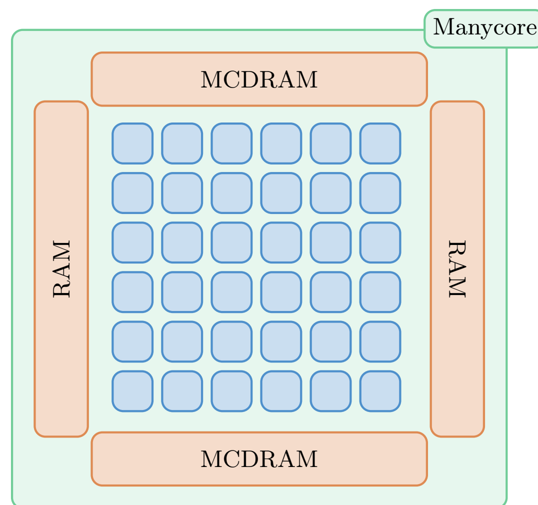

Figure 2.5 | Schematic view of a manycore processor.

Thread Programming

To take advantage of the cores available in multi and manycore processors, application developers can adopt a thread-based programming model. In this case, several *threads* execute on the cores of the processors. The threads are sequences of operations that are executed concurrently on the cores and that share the memory of the processor. Several tools exist to manage the expression of threads. The most low-level tool is Posix Thread, usually known as *pThreads*, that provides an API for thread creation, control execution and destruction [25]. While pThread provides numerous tools to manage the life of the threads, explicit creation and management of a pool of threads is complex and other higher level tools are often preferred for parallel programming.

OpenMP [32, 3] is an extension of Fortran, C and C++ codes that provides compiler directives mainly for loop parallelization. The strength of OpenMP is that the programmer adds `pragma` directives on top of the loops that need to be parallelized and OpenMP manages a pool of worker threads with their creation, destruction and synchronization in a transparent way. OpenMP codes often work according to a *fork-join* pattern [82]. During sequential regions, only one thread, called the *master thread* is active. When reaching a parallel region, work is distributed among the master thread and the pool of worker threads.

Task-Based Programming Models and Work Stealing Concept

Instead of managing threads, *task-based programming models* [22] propose to describe the parallelism of an application at a higher level. The programmer describes *tasks* that are functions associated with data. Tasks can have dependencies: a task may need the completion of other tasks before being executed. An application is therefore described as a *Directed Acyclic Graph* (DAG) where the nodes are the tasks (computation and input data) and the edges dependencies between them. A *scheduler* is in charge of distributing the tasks on a set of worker threads created at the beginning of the application. With the advance of manycore processors, task-based programming models are likely to become a standard for the future supercomputers [81].

One of the scheduling techniques used in a task-based programming model is the *work stealing scheduling* [23]. Intel® Cilk [22] and Intel® Threading Building Blocks (TBB) [99, 5] are task-based programming models that implement such scheduling. In a work stealing context, the threads are first assigned a set of tasks that they have to execute. When a thread has executed all of its tasks, it invokes a stealing mechanism. It selects a victim thread and steals a task from the the victim's set of tasks if available. Otherwise, it tries with another victim until it has found one victim to steal or until the scheduler puts it to sleep. The main advantage of the work stealing scheduling is that it offers *load balancing* between the threads. If a thread has been assigned less work than another thread, it can steal work of other threads instead of being idle, hence reducing the thread idleness periods and the execution time of the parallel region. Many efforts have been dedicated to reduce the work stealing costs. In particular, the threads are supposed to steal old tasks that are more likely to generate local work than newest ones, reducing the number of times a thread has to steal.

Scheduling a task is not always easy, in particular on NUMA machines where data locality is very important. Without careful tuning, a task may be executed on a core that does not belong to the NUMA node where necessary data are stored leading to costly data transfers. More evolved runtimes exist to handle these kinds of issues. For example, StarPU [15] is a C library that provides advanced scheduling strategies to assign tasks to the computing units based on task dependencies, priorities and data locality, and XKaapi [53] is a runtime system that implements a work-stealing strategy to execute applications parallelized with the task-based programming model introduced in OpenMP in the 3.0 and 4.0 updates. Virouleau et al. [112] propose several heuristics to control data placement on NUMA nodes at different levels (initialization or application execution) based on the architecture topology and tasks data dependencies.

In multithreaded applications, the *efficiency* is a metric that shows the capacity of the application to use the cores of the processor. An application is 100% efficient when it provides enough parallelism so that all the cores of the processor are active during the makespan of the

application. Parallel applications are almost never 100% efficient, especially because of sequential regions where only the master thread is active.

Simultaneous Multi-Threading

Usually, existing tools create at most one thread per core because traditional cores can only execute one thread at a time. It is possible to *over-subscribe* the cores by creating more threads than the number of cores. In this case, the processor Operating System (OS) is in charge of scheduling the threads on the cores. One current strategy is the time slicing where the OS alternates between several threads. To do so, the OS performs a *context switch* by frequently saving and restoring the state of the threads to suspend and resume them. However, core over-subscription has a cost and must often be avoided unless carefully tuned, as we will see in Section 2.2.2.

A way to improve the performance of a processor is to use Simultaneous Multi-Threading (SMT), or *hyperthreading* to use the term introduced by Intel®. It enables simultaneous execution of multiple threads on a core by assigning to each physical core two or several logical cores. Each logical core has its own registers but the ALU and the caches of the physical core are shared by the logical cores. The main purpose of hyperthreading is to hide the *memory latency* that is to say the time a thread has to wait before data are available in the caches. When a thread needs to use data that are not available, another thread can use the ALU of the core and the performance of the simulation code should increase. However, it has been shown that the performance gain of hyperthreading highly depends on the application [72].

2.1.2 Node Interconnect

A supercomputer is composed of several compute nodes for achieving a high degree of parallelism. For example, the largest supercomputer in June 2018, Summit[2], is composed of 4,608 nodes [6]. In order to achieve high performance, the nodes need to be interconnected by a fast network and distributed programming models should be used to parallelize applications on multiple nodes.

Network Interconnection

When using several compute nodes, the program is separated into several *processes*, each of them executing on a compute node. Each node has its own memory and simulation data are distributed among the different nodes. However, processes may need data that are not on their own node to progress and the different processes have to exchange data. According to Amdahl, the execution of a program is led by thee factors: the computations that are intrinsically sequential, the computations that are parallel and the computations related to the communications and synchronizations. While methods allow to overlap computations and communications, data exchange still corresponds to a significant part of the total execution time and high performance networks are necessary to reduce the synchronization times.

When looking at the Top500 supercomputers in June 2018, we observe that many interconnect networks exist (Figure 2.6) but 3 systems are mostly used: Ethernet with rates between 10 and 100 GB/s, Intel® Omni-path with rates up to 100 GB/s and Intel® Infiniband with rates up to 200 GB/s.

[2] https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/

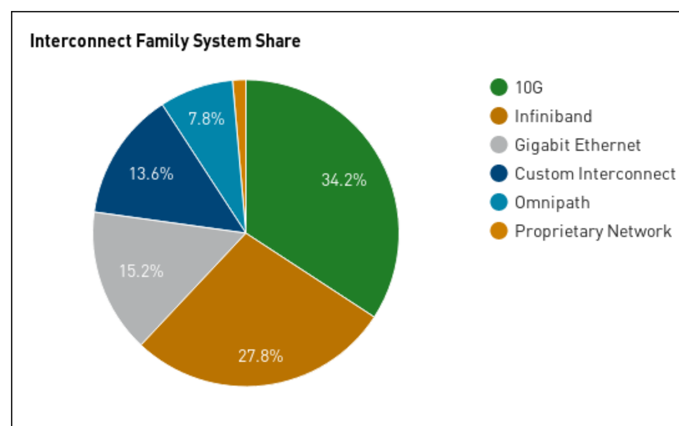

Figure 2.6 | Repartition of the network interconnect for the June 2018 Top500 supercomputers [6].

Distributed Memory Programming

When an application is composed of several processes distributed among several nodes, tools must be used to exchange data between the processes. Different tools exist but the predominant model in high performance computing is the Message Passing Interface (MPI) [100]. MPI is a standardized protocol for message-passing operations. The processes share a communicator, most often the default `MPI_COMM_WORLD` communicator and the programmer explicitly describes the communication scheme of the processes inside the communicator by using send and receive functions. The communications can be between two processes (point-to-point) or between all the processes (collective). In the first versions, the communications were always *blocking*, the processes waiting for the data exchange completion before resuming to the simulation computation. *Non-blocking* communications are now widely used to reduce the communication costs. In this case, a transfer request is created and handled by specific hardware. The process resumes whenever the request has been issued, enabling overlapping between computations and communications. The programmer can check the status of the request at any time in the program and they must manually check that data have been received before using them. In blocking and non-blocking communications, a synchronization is necessary between processes involved in the communication. *One-sided communications* propose to reduce this synchronization. Each process exposes a part of its memory to the other processes so that they can directly access it to write and read data.

On multicore processors, it is possible to run several MPI processes per node. In this case, the processor memory is split between the different processes and a process cannot access the memory of another process, even if they are located on the same node. Data sharing can be made by using *shared memory segments* and MPI proposes services so that intra-node communications are faster than inter-node communications. With the advance of manycore processors where the cores have a smaller frequency, using only MPI for the intra-node parallelization is more and more difficult and many codes are now switching to MPI+X programming models where MPI is used for the inter-node parallelization and communications and a multithreaded programming language (OpenMP, TBB, …) is used for intra-node parallelization. This is called *hybrid programming*. Usually, one MPI process is launched per NUMA node to limit the NUMA effects. Each MPI process is composed of a master thread and a set of worker threads managed by a scheduler, with one distinct scheduler per MPI process. To handle hybrid programming, MPI provides four

levels of thread safety:

- `MPI_THREAD_SINGLE`: only one thread is used in each MPI process. This is the default mode for MPI only applications;
- `MPI_THREAD_FUNNELED`: there may be multiple threads per MPI process but only the master thread is allowed to perform MPI calls;
- `MPI_THREAD_SERIALIZED`: there may be multiple threads per MPI process and several threads are allowed to perform MPI calls but only one at a time;
- `MPI_THREAD_MULTIPLE`: there is no restrictions, all the threads in a MPI process are allowed to perform MPI calls simultaneously. This is the most constraining threading level for the MPI implementations and all the vendors do not include it in their library due to overheads in the execution times [108].

Distributed Task-Based Programming

To use the term from Hoque et al [60], MPI+X programming encourages the practice of *hero-programmers* where the developer needs to express the parallelism, map it to the available resources and manage the data transfer between the nodes. New distributed programming models try to extend the task-based programming model presented in the previous section to a distributed memory context. StarPU has been extended to distributed memory environment by explicitly [14] or implicitly [7] specifying MPI communications. Legion [20] is an asynchronous many-task model that supports distributed memory architectures. A Legion program is decomposed into a task hierarchy where each task declares which parts of the data it needs to access or modify. Task execution is transparent for the developer. Task dependencies are deduced by the runtime based on these declarations and the runtime performs the necessary data movement operations. PaRSEC [60] is a task-based runtime for distributed architectures capable of tracking and moving data from different nodes. Dependencies between tasks are explicitly described by a domain specific language. HPX [65, 66] proposes a programming model based on the interfaces defined by the C++ standards. The user can use C++ like functions to write asynchronous codes that can be executed in shared or in distributed memory. The goal of these programming models is to provide an alternative to MPI+X programming. However, shifting to these programming models requires numerous code modifications and MPI is still mostly favored by application developers.

2.1.3 Filesystem and I/O

Large-scale simulations create a large amount of data that need to be stored to later be analyzed in a post-processing phase. Efficiently outputting data is a major challenge for the exascale era [13]. We present in this section the storage system of supercomputers and the I/O libraries used to efficiently output data.

Parallel Filesystems

High performance software programmers rely on *parallel filesystems* to store the large amount of data produced by their programs. Data are split into blocks distributed on a set of disks distinct

from the compute nodes and connected thanks to a high performance interconnection network. The programmer does not need to know the location of the different data blocks to open a file. Indeed, the parallel filesystem hides the underlying complexity and the programs query files as if data were stored on a unique filesystem with a large capacity. The most common parallel filesystem is Lustre [40].

I/O Libraries

Several I/O libraries exist to output simulation data into the filesystem. MPI-IO [109] is part of the MPI standard. It provides an API similar to the standard MPI API to open one file per iteration and let all the MPI processes write into the same file. A MPI-IO file is a binary file composed of a header with metadata information and a body where each MPI process writes its corresponding data at a specific offset location in the file. The offset guarantees that the MPI processes write their data in distinct locations. Parallel NetCDF [73] and Parallel HDF5 [50] are parallel libraries that use hierarchical file formats enriched with metadata to ease data manipulation. Hercule [111] is a tool developed by CEA to store data into a database-like format.

The Adaptable IO System (ADIOS) [76] is an I/O library that provides a generic interface to use transparently different I/O transport layers. The observation of the ADIOS team is that the performance of I/O libraries and the optimal file format significantly differs from a supercomputer to another. However, it is not the role of the application developer to modify the I/O layer of a simulation code every time the simulation is deployed on another architecture. ADIOS therefore provides a generic and simple API that simulation codes use to output data. The strength of ADIOS relies in its external XML data file where the different variables produced by the simulation are described with their dimensions, units and so forth. The simulation just needs to provide a pointer to the corresponding variables to the ADIOS API and ADIOS handles the data output. The XML file also allows to choose I/O methods and file formats, by using for example MPI-IO, NetCDF or HDF5. Once a simulation has been instrumented with the ADIOS API, any changes in the I/O method is made without any code recompilations, by just modifying the XML data file. To reduce the cost of writing data, ADIOS provides asynchronous I/O layers that overlap simulation computation and data output.

I/O Bottleneck

I/O has been recognized as a major challenge for exascale computing. Studies show in particular that the gain in I/O performance will not follow the gain in computation performance. For example, a factor change of 500 in the number of floating point operations per seconds is expected between petaflopic and exaflopic supercomputers while the factor change is expected to be 10 - 30 only for the I/O rate [8]. This is called the *I/O bottleneck* and it leads to a major limitation for the post-processing approach. Indeed, large-scale simulations generate data at a much higher rate than the filesystem can actually manage and simulation performance is significantly degraded in case of frequent file outputs. This poses a major issue for scientific discovery because the end-user is likely to reduce the output frequency to limit the impact on simulation performance [21].

There are different ways to enable scientific discoveries at a higher frequency with lower impacts on the simulation performance. Works have first been focused on creating I/O libraries that enable asynchronous data output to overlap simulation computation and file writing [76, 64].

However, files still need to be read for the scientific discovery process and this is still very costly in term of time and memory consumption. To reduce the need to write data into the persistent storage, *simulation-time* visualization and data analytics paradigms [113] propose to visualize and analyze data while still resident in the supercomputer memory. Research have first been focused on real-time rendering of simulation data by saving periodically screenshots of the state of the system [114] instead of writing data into the filesystem but it proved to be inefficient for scientists because it was not explorable enough. Many works have focused on improving the visualization of such systems [9] and on simulation-time data analytics in a broader range. In the following, the word *analytics* is used to design both data analytics and data visualization. We distinguish two kinds of simulation-time analytics and we follow the terminology employed by Bauer et al. [18]. In the *in situ* paradigm, simulation and analytics are executed on the same nodes while they are executed on different sets of nodes in the *in transit* paradigm. The next two sections aim at introducing the concepts of in situ (Section 2.2) and in transit (Section 2.3) data processing.

2.2 In Situ Processing

In the in situ paradigm, simulation and analytics are executed on the same nodes. In the *synchronous* approach (Section 2.2.1), the simulation is periodically stopped to execute analytics instead. It is possible to achieve better performance by executing simulation and analytics concurrently. This is called *asynchronous* in situ and we can distinguish two main techniques: simulation and analytics can be interleaved on the same cores (Section 2.2.2) or use distinct sets of cores (Section 2.2.3).

2.2.1 Synchronous In Situ

In a *synchronous* in situ processing, simulation is periodically stopped to execute analytics or visualization routines (Figure 2.7). The simulation is first executed on the cores of the nodes. When the simulation reaches an iteration of analytics, the simulation stops and the cores are used to execute analytics instead. The simulation resumes to the next iteration when the analytics ends.

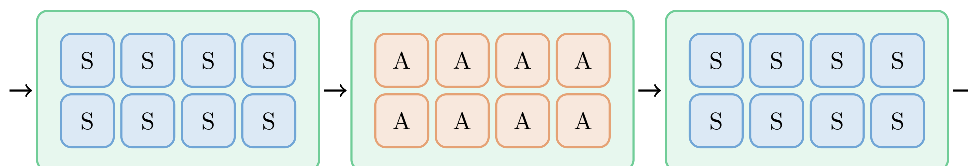

Figure 2.7 | Synchronous execution of in situ analytics on a processor with 8 cores. The cores are first used to execute the simulation. When reaching an analytics iteration, the simulation is stopped and the cores are used to execute the analytics instead.

The more direct way to perform synchronous in situ processing is to directly embed the analytics or visualization routines inside the simulation code. Many leadership scientific applications have embedded analytics [97, 56, 104] that can be used at different levels. Lightweight analytics are used to periodically monitor the evolution of a parameter of interest. Often these lightweight

analytics are executed at a high frequency to check the status of the simulation. Heavier analytics are used to determine physical properties [105] and are executed at a lower frequency to reduce the end-to-end execution time. The main advantage of this approach is that the analytics are aware of the simulation data structure and can directly work on simulation data without a copy to be performed. The main drawback is that these techniques are application dependent and cannot be easily reused for other codes.

ParaView [16] and VisIt [27] are both well-known visualization applications relying on the VTK visualization library [102] and that provide general purpose libraries to integrate visualization routines inside the simulations.

ParaView Catalyst [19] has originally been designed to run synchronously with the simulation, the analytics and visualization routines directly using simulation data. The library makes a link between the simulation and the tools provided by ParaView. The instrumentation of a simulation with Catalyst requires the definition of an adaptor to convert the simulation data structures into VTK data structures understandable by ParaView. This adaptor allows an isolation between the simulation code and the VTK library but often a copy is necessary to map between the two data structures. The interface between the simulation and Catalyst is then made thanks to three functions: an initialization call initializes the adaptor and Catalyst, a co-processing call converts the simulation data into VTK data structures and executes the analytics and/or visualization routines and a finalization call cleans Catalyst states. Catalyst also proposes an interface to connect with ParaView sessions to enable live exploration of the data.

VisIt Libsim [68] was originally developed to ease the interactive connection between a running simulation and the VisIt GUI. Simulation data can therefore be explored interactively, offering a powerful tool for debugging and for computational steering. With the advance of in situ processing, Libsim was enhanced with a batch mode. The library provides functions to save complex visualizations and plots and to export data in an in situ way. An adaptor is also needed to make the link between the simulation and the VTK data structures used by VisIt. The adaptor functions create Libsim objects that store pointers to simulation data, enabling Libsim to use data in a zero-copy way. Libsim consists in a control library and a runtime library. The simulations is only linked to the control library. This way, simulations instrumented with VisIt grow or use additional memory only when they perform in situ processing.

To integrate a general purpose library into a simulation code, the simulation code must be instrumented with the library API. Moreover, custom analytics have to be implemented with the library requirements. Each library proposes its own API and requirements, which leads to interoperability problems. To address this issue, SENSEI [17] proposes a generic in situ interface so that the user can easily switch between Catalyst, Libsim and the ADIOS library. The data model of SENSEI is built on a variant of the VTK data model. In particular, VTK was enhanced to support multi component arrays such that structures of arrays and arrays of structures. It enables to map most of the simulation data into VTK data structure in a zero-copy way. The user can then instrument the simulation code with the SENSEI API and choose the analytics and visualization routines as well as the I/O library that execute them in an XML file.

When executing analytics synchronously with the simulation, the end-to-end execution time is the addition of the simulation and analytics time, plus some possible overheads due to resource sharing. Executing analytics synchronously can therefore highly increase the end-to-end execution time. Moreover, the synchronous approach does not take into account the fact that simulations are almost never 100% efficient. Indeed, simulations present sequential regions where

the cores are idle (MPI communications for example) or portions of codes that are not efficient enough to use all the available cores. This is all the more true with the current increase of the number of cores in modern processors. One idea to increase the performance of in situ analytics is to harvest these CPU cycles to execute analytics. This is called *asynchronous* in situ and it aims at reducing the end-to-end execution time compared to synchronous in situ. Asynchronous in situ methods can fall into two categories that are the subjects of the next two sections: simulation and analytics can run on the same cores (Section 2.2.2) or on distinct sets of cores (Section 2.2.3).

2.2.2 Over-Subscription of the Cores

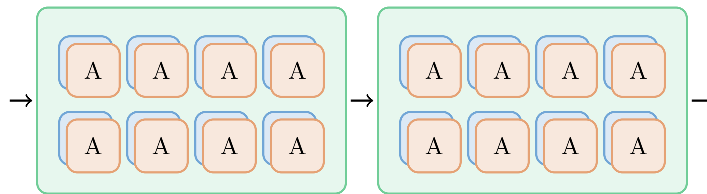

Figure 2.8 | Asynchronous execution of in situ analytics with core over-subscription on a processor with 8 cores. Simulation and analytics processes can use the 8 cores for their computations, leading to core over-subscription.

The first technique to asynchronously run simulation and analytics on the same nodes is to over-subscribe the cores of the nodes (Figure 2.8). Simulation and analytics are usually two distinct processes that can use all the cores of the nodes. The simulation process is usually set with a higher priority than the analytics process and the OS scheduler is in charge of co-scheduling the two processes. In an asynchronous execution of in situ analytics, it is necessary to guarantee that the simulation does not overwrite the data used by the analytics. A copy of the relevant data is often necessary and simulation data are made available to the analytics process either via sockets or thanks to shared memory segments. Efficiently co-scheduling simulation and analytics processes on the same cores is difficult because of contentions on shared resources (caches, memory buses, ...). It is therefore a potential source of interference for the simulation [86] and the OS scheduler has been proved to be insufficient to efficiently co-schedule simulation and analytics. Zheng et al. [119] have demonstrated the limitations of the OS scheduler for six representative simulation codes and five analytics benchmarks that stress different subsystems of the machines (computations, caches, memory, network, ...). They show that the co-scheduling of simulation and analytics by the OS scheduler can lead to a simulation slowdown of up to 57% compared to the execution time of the simulation alone. In particular, they highlight that the OS scheduler only focuses on core idleness while the analytics may also induce contentions on shared resources as the last level cache or the memory controller.

In the same study, Zheng et al. have measured the proportions of sequential regions in six representative MPI+OpenMP applications. These sequential regions can come from MPI communications and I/Os for example and correspond to portions of codes where only the master thread is active, the other threads being idle. They show that the cumulative sequential periods can represent up to 65% of the total execution time and that the percentage of sequential regions increases when running the simulation on more nodes. These sequential regions are often short (less than 1ms) and often too short to schedule analytics without impacting simulation performance. They therefore propose the Goldrush middleware to run analytics process dur-

ing long enough simulation sequential regions. Simulation and analytics processes are launched simultaneously but the analytics process is initially suspended. When the simulation reaches a sequential region, Goldrush determines if the sequential region is long enough based on an history of previous iterations saved by the middleware. If the sequential region is predicted to be long enough (more than 1ms), the analytics process is resumed with a SIGCONT signal. It is suspended again with a SIGSTOP signal when the simulation reaches the next OpenMP parallel region. Goldrush also monitors hardware counters to check the impact of the analytics on shared resources and throttles the execution rate of the analytics if the contention on shared resources is too important. Goldrush can be used in a simulation either by instrumenting the simulation code with the Goldrush API or without any code modifications by directly instrumenting the OpenMP runtime library. Goldrush is shown to be up to 42% faster than the OS scheduler and the asynchronous execution of analytics can be performed with an overhead of 1.7% on average compared to the execution time of the simulation running alone. This solution is well adapted for simulations that use a fork-join model that alternates between sequential and parallel regions and requires to instrument the simulation code or the multithreaded runtime library. However, we will see in Chapter 5 that it is more difficult to take advantage of Goldrush capacities when the simulation has short sequential regions. Moreover, Goldrush only focuses on the simulation sequential regions and does not harvest the analytics sequential regions nor the inefficient parallel regions.

Mondragon et al. [87] propose a more general kernel-based approach to the co-location of simulation and analytics processes on the same nodes by over-subscribing the cores. For several simulation and analytics codes, they measure the overhead on the simulation code when co-locating analytics codes on the same cores using different scheduling strategies. They show that advanced OS scheduling policies can efficiently co-schedule simulation and analytics processes. These scheduling policies are intended to be more general than Goldrush but they are not always available in HPC operating systems and the appropriate scheduling policy is often analytics dependent.

2.2.3 Core Separation

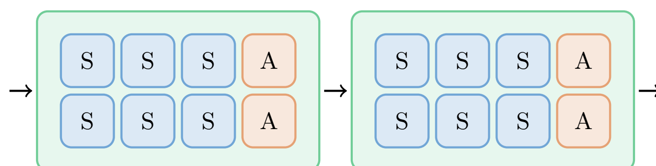

Figure 2.9 | Asynchronous execution of in situ analytics with a helper core strategy on a processor with 8 cores. Simulation process runs on 6 cores and analytics process uses the remaining 2 cores.

Because of the negative impact of core over-subscription on the simulation performance and because of the advance of multi and manycore processors, a second technique to asynchronously run simulation and analytics on the same node is to execute them on distinct sets of cores (Figure 2.9). On each node, a set of cores is dedicated to analytics while the remaining are in charge of simulation processing. These dedicated cores are commonly named *helper cores* and we call this approach the *static helper core* approach in contrast with the *dynamic helper core* approach that

we will present in Chapter 6. Usually, a small amount of cores are confiscated to the simulation so that the simulation always run on more threads than the analytics. The simulation runs on less cores but the performance loss is generally less than the ratio of confiscated cores because the simulation is usually not 100% efficient [41, 117].

Several works use the static helper core approach to dedicate cores for non-computational tasks. Li et al. [74] start from the observation that simulation codes do not scale well enough to use all the cores of a manycore architecture. Moreover, they also assume that future HPC systems are likely to be equipped with solid-state disks (SSDs) as an intermediate storage system to mitigate the pressure on storage system. They therefore design a functional partitioning runtime environment to dedicate cores to non-computational tasks (checkpointing, data transformation, ...) and to use SSDs to improve the I/O throughput. Ma et al. [78] propose the active buffering approach where the compute cores transfer data to dedicated I/O cores that are in charge of writing them into the filesystem. The method hides the writing and data migration costs by executing asynchronously simulation and I/O tasks. GepPSea [103] also implements a helper core strategy to offload application-specific tasks such as communication services, dynamic load distribution and network protocol processing.

Dorier et al. [41] use the dedicated core approach to hide the I/O jitters coming from two standard file output techniques: the file-per-process approach where each process writes a file and the collective I/O approach where the processes synchronize to open one share file per iteration. They introduce the Damaris middleware where one core on each node is dedicated to I/O management. Damaris uses a process-based approach. The middleware separates at runtime the simulation MPI communicator in two: one for the simulation and one for Damaris, which corresponds to the helper cores. The data exchange between simulation and analytics is made possible thanks to a shared memory segment managed by the Boost library [1]. The middleware also proposes a plugin system to allow the end-user to define transformations to be applied to the data prior to the output, as compression or indexing. Damaris is configured at runtime thanks to an XML file that describes the plugins that need to be loaded by the middleware and that describes simulation data (names, descriptions, dimensions, ...) in order to reduce the data needed to be stored in the shared memory segment. Damaris is then used to create one file per node and it was shown that file output with the middleware is 35% faster than file-per-process approach and 3.5 times faster than collective I/O.

Damaris/viz [42] extends the Damaris middleware so that helper cores are used to execute in situ analytics. Zero-copy sharing of data in the shared memory segment is made possible for simulations that use a double buffering technique. In this case, the simulation updates the data at iteration $i + 1$ based on a copy of the data made at iteration i. If the simulation allocates its data structures directly in the shared memory segment, simulation and analytics can work simultaneously on the data of iteration i. At the end of the iteration, the simulation does not need the buffer of iteration i anymore and lets the analytics destroy it. User-defined analytics can be executed thanks to the plugin system. Damaris XML file is also enhanced to describe mesh information so that VisIt and ParaView can be used asynchronously thanks to the middleware [43]. More details about the Damaris middleware will be given in Chapter 5. In particular, we will see that the main drawback of the static helper core strategy relies on the choice of the number of helper cores. The optimal number of helper cores allows to take benefit from the analytics parallelization without removing too much cores from the simulation. We will see that a wrong choice in this parameter may lead to significant performance penalties.

Performance gain of the static helper core approach are usually significant compared to synchronous or core over-subscription approaches but the simulation and analytics are isolated on distinct subsets of cores and the static helper core strategy does not allow the analytics to harvest the sequential regions of the simulation. Even if they have not been used in the in situ context, current works focus on methods to dynamically assign disjoint sets of cores for processes running on the same nodes. Cho et al. [28] propose to adapt the core resources according to the performance characteristics of parallel and sequential code sections of running applications. Moore et al. [89] and Grewe et al. [55] compute the best number of threads for OpenMP applications when executed concurrently with other multithreaded applications. Raman et al. [98] propose a runtime to monitor parallel task execution and adjust the number of threads iteratively to find an optimal number of threads. Hugo et al. [61] extend the StarPU runtime with the *context* feature that allows an application to run on a subset of the available processing units. An *hypervisor* is then used to dynamically resize the different contexts executed in a node based on resource usage and computation progress to minimize the application execution times. Harris et al. [57] propose the Callisto prototype to dynamically vary the number of cores of co-located applications. When two applications run on the same nodes, the cores are split into two distinct groups with equal size. When an application enters a sequential region or when it does not need all the dedicated cores, the second application can use these otherwise idle cores for its own computations. To our knowledge, no publications mention the use of these techniques for in situ processing but they may be used in the in situ world to prevent the resource loss induced by the static helper core approach and we will see in Chapter 6 how to implement a dynamic helper core approach using the TBB work stealing scheduler.

2.3 In Transit and Hybrid Processing

The in situ approach has proved to have many advantages. In particular, simulation data are available on the nodes where the analytics and visualization take place, reducing the need for data movement between the nodes. Frameworks are implemented so that the analytics can directly access simulation data, without a copy to be performed. However, a copy is most often necessary to transform the data into a format understandable by the in situ framework. Moreover, the presence of an analytics process on the nodes of the simulation tends to disturb the simulation execution because of contentions on shared resources. Finally, the memory available to run in situ analytics is often constrained by the memory already used by the simulation. To reduce the impact of analytics processes on the simulation execution, several works focus on *in transit* processing, where simulation and analytics run on distinct nodes (Section 2.3.1) and on *hybrid* processing where lightweight computations are performed in situ and heavier analytics in transit (Section 2.3.2). Despite the data movements between nodes and data redistribution costs induced by these methods, in transit and hybrid processing are proved in some cases to have advantages compared to in situ processing [67].

2.3.1 In Transit Processing

In an in transit paradigm, the simulation and analytics processes are executed on distinct sets of nodes (Figure 2.10). The nodes where the analytics run are called the *staging nodes* and there are usually less staging nodes than compute nodes. In this section, we use the term analytics to en-

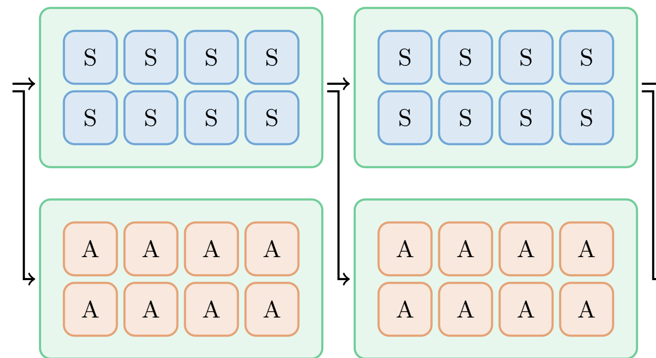

Figure 2.10 | In transit execution of analytics on two processors with 8 cores each. Simulation process runs on one node and analytics process on the second node.

capsulate user-defined data processing, visualization routines or file writing into the filesystem. When the simulation reaches an iteration of analytics, data are sent to the staging nodes where they are processed by the analytics processes. Different tools are developed to enable in transit processing.

Zheng et al. [116] propose the PreDatA middleware to prepare the data to be stored, inspected or analyzed. They start with the observation that the file writing phase has to be optimized to reduce the writing time and hence reduce the impact on the simulation execution time but that it is also important to optimize the reading phase for later post-processing. The PreDatA middleware algorithm is decomposed in three steps. First, data are extracted from the simulation by the ADIOS library. Lightweight user-defined processings can eventually be executed on the simulation nodes. The simulation data are then packed into buffers and transmitted to the staging nodes thanks to ADIOS. The data chunks are finally processed by the staging nodes in a streaming manner using an approach close to the MapReduce paradigm [34]. By defining their own functions in the MapReduce paradigm, the end-user can plug their own data operations. Each staging node corresponds to one MPI process and the processing algorithms are multithreaded to use all the cores of the staging nodes.

DataSpaces [39] implements a distributed in-memory storage system on staging nodes. It works in a client server mode where DataSpaces is the server hosted on the staging nodes and the simulation is one of the clients. Simulation data are extracted from the simulation, indexed and stored in the shared space. The data extraction is made thanks to DART (Decoupled and Asynchronous Remote Transfers) [38], an asynchronous communication and data transport layer using Remote Direct Memory Access (RDMA) and one-sided communications. Once the data are indexed and stored, a query engine is provided to extract information based on a key-value system. Several applications can dynamically register as clients of DataSpaces. They can use the query engine to access simulation data stored in the staging nodes. DataSpaces implements a data redistribution mechanism and the storage is transparent to the clients: they query data to DataSpaces that is in charge of forwarding the request to the staging nodes holding the data and to provide to the clients the required data.

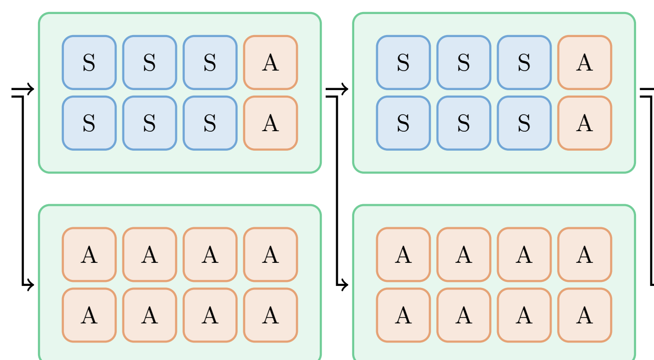

Figure 2.11 | Hybrid execution of analytics on two processors with 8 cores each. On the first node, a simulation process runs on 6 cores and an analytics process uses the remaining 2 cores. On the second node, an analytics process uses the 8 cores.

2.3.2 Hybrid Processing

The in transit paradigm enables to extract simulation data and to send them to staging nodes where they can be processed, filtered and written asynchronously to the filesystem. It enables to reduce the perturbations of the analytics and I/O processes on the simulation execution time that occur when simulation and analytics processes share the same nodes. To go one step further, some works propose the hybrid paradigm where lightweight analytics are performed on the compute nodes and heavier analytics and I/Os are executed on the staging nodes (Figure 2.11). It allows for example to reduce the data movement from the compute nodes to the staging nodes by filtering the data in situ before sending them to the staging nodes.

GLEAN [113] is an hybrid framework where analytics can run either in situ synchronously with the simulation or in transit on dedicated staging nodes. GLEAN can be used through its API inserted in the simulation code or more transparently thanks to calls to I/O libraries such as Parallel NetCDF or HDF5. When running in situ, GLEAN is embedded into the simulation code and share the same address space than the simulation, providing user-defined analytics with zero-copy. In situ analytics include data transformation, data reduction or I/O optimizations. The simulation is blocked to copy data from the compute node to the staging nodes but analytics and simulation processing are performed asynchronously. On the staging nodes, GLEAN runs as an MPI job that communicates with the compute nodes. It then uses I/O libraries such as MPI-IO to write data to the filesystem.

The data staging frameworks presented so far use the staging nodes mostly for storing the data and pre-process or transform them to be efficiently written into the filesystem. However, these techniques do not fully use the cores in the staging nodes. Bennett et al. [21] propose a hybrid framework to use the available computing capabilities. Well-parallelized analytics codes are executed in situ to reduce the size of the data sent to the staging nodes and less-parallelized or even sequential analytics are executed on the staging nodes. They show that a large class of algorithms used for scientific visualization or data analytics can be modified to have in situ and in transit parts and they rewrote three commonly used algorithms. The framework is built on DART for the data movement and on DataSpaces to share data between the simulation and the analytics.

Zheng et al. [117] tackle the issue of analytics placement. For each simulation-analytics

dataflow, the question is to know where to run analytics: synchronously with the simulation, asynchronously on the same node than the simulation, asynchronously on staging nodes or in a post-processing way. Data placement has indeed been identified as critical for the performance of simulation and analytics coupling [118]. They identified that most of the existing in situ and in transit frameworks have fixed data placements and are not flexible enough to allow different placement strategies. They propose the FlexIO middleware that offers flexible data movement to enable analytics to be launched either in situ or in transit. FlexIO relies on ADIOS for the extraction of data from the simulation, shared memory segment for intra-node data movement and RDMA for inter-node data movement. The end-user can use different placement strategies without any changes in simulation or analytics codes thanks to the ADIOS layer used by FlexIO. Codelets can be executed along the I/O path to perform on-the-fly lightweight processings. FlexIO provides flexible data transport but the user still has to choose the data placement. To help them find an optimal data placement, the middleware proposes two placement policies: a holistic placement policy reduces data movement costs and a node topology aware policy takes into account the cache topology and deep memory hierarchy.

Dreher et al. [44] propose to execute simulation and analytics in a *dataflow* model. The analytics form a pipeline where the output of one analytics code is the input of another one. They propose a flexible framework to describe and execute simulation and analytics dataflows as a graph where the nodes, called *modules*, are the parallel or sequential applications and the edges are the communication channels between the nodes. They redesign the FlowVR middleware, originally designed for large-scale virtual reality [10], to allow in situ and in transit executions of analytics. The simulation and analytics applications have to be turned into modules that are processes with input and output ports. Transforming an application into a module is made possible thanks to the FlowVR API that consists in three main functions: *wait* to suspend the module until data are available in its input ports, *get* to get the data from the input port and *put* to put the data in the output port. The modules are compiled as different executables and a Python script is used to create the graph by defining where and on how much resources the executables should run as well as the input and output ports of the different modules. FlowVR then launches the different applications on the required resources. On each node, a daemon is in charge of transmitting the messages between the applications. When the applications are on the same node, data exchanges are made through shared memory segments. When the applications are on distinct nodes, data transfers are made thanks to MPI. FlowVR provides redistribution modules when the simulation and analytics do not have the same MPI splitting. FlowVR also supports processes and shared memory segments binding to reduce the interferences between the applications.

Decaf [46] is also a middleware that supports the hybrid approach. Its design is close to the design of FlowVR where multiple executables are linked into dataflows to form a graph. The graph definition is simplified and the execution does not rely on daemons. Decaf relies on the MPMD (Multiple Program Multiple Data) capability of MPI. All the executables are launched in the same MPI context, sharing the `MPI_COMM_WORLD` communicator. A Decaf dataflow is composed of a producer, a consumer and a *link* that corresponds to an intermediate parallel program that transforms the data between the producer and the consumer. The links are inspired of the PreDatA codelets except that they have dedicated resources. Decaf creates five communicators for each dataflow: one for each producer, consumer and link, one for the communications between the producer and the link and one for the communications between the link and the consumer. The link can be located on the producer node, on the consumer node or on dedicated

nodes. Data are transferred from producer to link, processed in the link and transferred from link to consumer thanks to MPI and the Bredala [45] library that provides a data model and a redistribution pattern when the producer and consumer are not executed on the same number of MPI processes. To be executed within Decaf, the applications must be instrumented with the Decaf API that corresponds to a few functions with a put/get model. The workflow is then described as the combination of several dataflows in a Python script.

The in situ approaches presented so far apply to MPI or MPI+X codes, where X is mostly OpenMP. As seen in Section 2.1, task-based programming models are alternatives to MPI+X approaches and in situ processing techniques also emerge in this context. Pebay et al. [94] have identified asynchronous many-task model to be well adapted to in situ processing because it only requires to describe which data are shared by simulation and analytics and not when, where and how this sharing must occur. Moreover, the analytics tasks are likely to be interleaved between simulation gaps. Indeed, the application is decomposed into simulation and analytics tasks with input data. Tasks being executed when inputs become available, analytics tasks can be scheduled during simulation sequential regions. Heirich et al. [58] have reported early experiments using Legion for in situ visualization. They show in particular that the Legion runtime manages to interleave simulation and analytics tasks without reducing the simulation throughput. Using these results for legacy MPI+X codes is still an issue because subsequent code modifications are required to switch an MPI application into a Legion application.

Larsen et al. [69] propose ALPINE, a flyweight hybrid infrastructure that supports synchronous in situ analytics and visualization and that can send data to staging nodes thanks to ADIOS [76] for example. ALPINE is the production version of the Strawman mini-app [70] with more data transformation and a distributed memory model. They implement VTK-h, a library that adds a distributed memory layer to VTK-m [90], this distributed memory layer being based either on MPI or on DIY [95]. Data are described thanks to the Conduit library [2]. ALPINE enables three kinds of actions: making one or several pictures, extracting data to write simulation data into the filesystem or to send data to other nodes thanks to ADIOS and transforming the data thanks to VTK-h filters.

2.4 In Situ Workflows Control

More generally, in situ processing can be seen as a *workflow* system [35]. A workflow system is composed of several distinct applications that need to share data. For example, multi-physics workflows can link several codes that do not compute the same physics. The workflows are generally described as graphs where the nodes are the different applications and the edges data dependencies between the applications [75]. Many scientific workflow management systems have been developed [12, 36, 84, 115] but they often rely on files to exchange data between the codes. In situ processing relies on a tighter coupling for improved performance. A simulation produces data and a set of analytics and visualization codes are connected to the simulation to process or visualize simulation data. In situ processing does not rely on files to pass data to analytics codes but it is a sub-part of workflow systems because it leads to complex workflows for analyzing and visualizing simulation data [51, 44, 35].

Efficiently managing in situ workflows is a difficult problem because it deals with the interaction of different components that do not show the same behavior. For example some applications may be very quick and the others very slow. All the components of a workflow have to be taken

into account to optimize the end-to-end execution time, that is to say the elapsed time between the moment the simulation begins and the moment the analytics of the last iteration ends.

One idea to reduce the end-to-end time is to choose the analytics frequency so that the analytics execution time can be overlapped by the simulation. Choosing the workflow components frequency is finding a balance between the scientific discovery process and the end-to-end execution time. To have more insights in the simulation, the user is likely to set a high frequency for data analytics but this may lead to high end-to-end execution times. On the contrary, reducing the analytics frequency may reduce the end-to-end execution time but can be detrimental to the scientific discovery process. Malakar et al. [80, 79] propose an analytical model for optimal execution of analytics given a memory and time budget. They take into account the simulation and analytics execution times, the available memory on compute and staging nodes, the network bandwidth, the importance of analytics and a minimum analytics frequency to determine the optimal analytics frequencies that optimize the end-to-end execution time. Because they take into account a minimum frequency for the analytics, their method ensures that analytics are at least executed according to the end-user choice but they can be executed more often.

The nodes of a workflow are likely not to have the same execution times. When the consumer takes longer than the producer, data are often buffered by the producer so that it can resume to the next iteration without waiting for the consumer to be ready. The data are then read or sent through the network in a FIFO (First In First Out) way. Dreher et al. [47] identify this as a bottleneck of in situ middleware. In particular in the case of visualization algorithms, it may be preferable to visualize the more recent data instead of the older ones to have better insights of what is going on in the simulation. They therefore propose the Manala library that allows the end-user to modify the data exchange policy between the producer and the consumer.

Fu et al. [52] make a comparison study of several state-of-the-art libraries that create in situ workflows, including Decaf and DataSpaces. They identify synchronizations and interlocks between the applications as a limitation of the approaches. In particular, the copy of large data sets from the simulation to the staging nodes induces large overheads on the simulation execution time. They propose the Zipper runtime where several helper threads in both simulation and analytics sides are used to send and receive blocks of data through a low latency network and eventually to the parallel filesystem if the analytics application is too slow. Zipper is based on asynchronous task parallelism to process blocks of data as they are made available by the simulation instead of sending large blocks of data, hence reducing the synchronization between simulation and analytics.

Traditionally, large blocks of data are sent to analytics processes because a conservative approach is often chosen for the data copy. In situ middleware provide a generic interface for the design and deployment of in situ workflows whose policy is to instrument the simulation code only once and to create in situ workflows without any code recompilations. Therefore, the simulation output is often not specialized for a particular analytics but rather left to be general: the simulation outputs all useful data and the analytics is in charge of filtering the data and using only what is necessary for them. This conservative approach leads to unnecessary data sent through the network and hence time and memory wasting. Mommessin et al. [85] propose a contract system between a producer and a consumer. The producer describes the data it can provide, the consumer describes what data it needs for its processing and the comparison between the two contracts tells what data fields have to be sent from the producer to the consumer.

2.5 Chapter Summary

The evolution of supercomputer has led to a growing gap between the data generation rate and the capacity to store and analyze these data in the traditional post-processing approach. The in situ paradigm proposes to analyze data while still resident in the compute node memory to reduce the need to store data into the storage system. Many techniques have been developed to analyze and visualize data in situ on the same nodes than the simulation or in transit on dedicated nodes and to minimize the end-to-end execution time of simulation-time analytics.

The in situ techniques were mostly developed for multicore processors with a few number of cores per processors. In particular, simulation and analytics run on different processes in most of the cases. With the advance of manycore processors, task-based programming models are currently emerging as a standard for the future exascale supercomputers and few techniques have been developed with this emergent programming model in mind. In particular, the work stealing concept offers a good opportunity for in situ processing because it can be used to harvest the sequential and the inefficient regions of a simulation to run analytics tasks instead. In the next chapter, we will introduce in more details the Intel® TBB library that implements the work stealing concept and the ExaStamp molecular dynamics code that uses the TBB library for its intra-node parallelization and we will identify the challenges of implementing a task-based in situ system within the simulation code.

3 Task-Based Molecular Dynamics for Exascale Computers

With the advance of multi and manycore processors, the simulation codes have to be optimized to take the best of the different levels of parallelism, the hierarchical memory and the vectorization capacities of the architectures. To that end, most of the modern codes are currently parallelized with a MPI+X programming model where MPI is used for the inter-node parallelization and a multithreaded programming language is used for the intra-node parallelization. Task-based programming model is envisioned to become a standard for the future supercomputers and is therefore a good candidate for being combined with MPI in modern codes. In particular, the work stealing concept provides load balancing capacities to enhance the thread efficiency of parallel applications. Intel® TBB is a library that provides a task-based programming model and a work stealing scheduler for the high level creation of tasks. Its code composability features make it a good candidate for the implementation of a task-based in situ framework (Section 3.1). In the context of molecular dynamics, ExaStamp is a simulation code developed with three levels of parallelism: inter-node with MPI, intra-node with Intel® TBB and explicit vectorization with Intel® intrinsics (Section 3.2). The code is optimized for modern supercomputers and offers a good opportunity for the study of the integration of a task-based in situ framework (Section 3.3).

3.1 Intel® TBB, a Task-Based Runtime

Task-based programming models propose a high level interface for programmers to describe their program as a set of dependent tasks and a scheduler to distribute efficiently the tasks to a set of worker threads it created. As summarized in Figure 3.1, a task-based program can be decomposed in three levels, the scheduler being the link between the different levels. The programmer only needs to express the potential parallelism of their program and the scheduler handles transparently the difficult part of distributing tasks to the worker threads and mapping the threads to the available cores given criteria such as task dependency and data locality for example. Intel® Threading Building Blocks (TBB) is an example of libraries to express task-based parallelism. It is implemented in modern C++ and it proposes parallel loop constructs based on lambda functions. Sequential lambda functions are written by the programmer and are transparently transformed into parallel algorithms by the TBB library. In this section, we describe the three levels of the task-based programming model of TBB. We first describe the TBB API to create tasks in the higher level (Section 3.1.1). We then explain how worker threads are created and how the tasks are distributed among the worker threads (Section 3.1.2). We finally describe

tools that help the task execution and in particular the mapping between the threads and the available cores (Section 3.1.3).

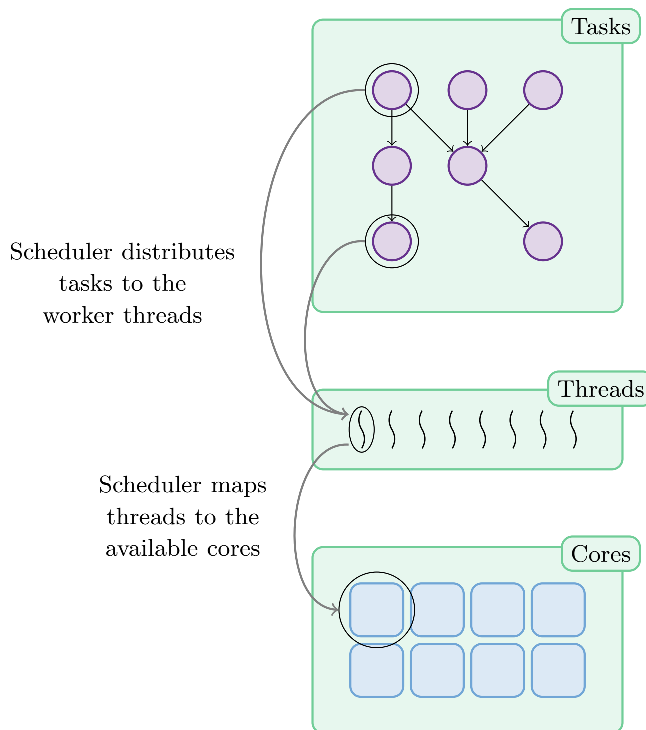

Figure 3.1 | Three different levels of the task-based programming model: the program is composed of a set of tasks distributed to a set of worker threads and mapped to the available cores by the scheduler.

3.1.1 Task Creation with TBB API

TBB provides different ways to create tasks, either implicitly thanks to predefined templated functions, or explicitly by spawning tasks or by creating task graphs with dependencies. The different methods have advantages and drawbacks and will be used throughout the manuscript for the implementation of our task-based in situ framework.

Implicit Task Creation

Scientific simulations are mostly organized around a main timeloop where variables are updated at each iteration based on the previous configuration. These variables are often stored as arrays (positions of particles in molecular dynamics, mesh elements in computational fluid dynamics, ...). Thus, one of the easiest way to parallelize an application is to parallelize its iteration loops. While OpenMP proposes to parallelize the loops thanks to `pragma` directives read by the compiler, TBB proposes templated C++ functions where the user describes the loops using TBB objects and lambda functions. The two most common patterns are `parallel_for` and `parallel_reduce`.

The `parallel_for` function is used to parallelize a loop that iterates on the elements of an array. For example, Figure 3.2 shows a sequential code to apply a function `foo` to all the elements of the array `tab` and the equivalent version with a TBB `parallel_for`. TBB provides

an object, `blocked_range`, that represents the interval on which the task applies. In the example, the task 0 corresponds to the computation of the 4 first elements of `tab`, the associated `blocked_range` being therefore composed of the interval $[0, N/4[$. The `blocked_range` of a task is found recursively during the execution of the parallel for loop (see Section 3.1.3) in a transparent way. The programmer just needs to define the function applied on the elements contained in the `blocked_range` thanks to a lambda function. The `parallel_reduce` function is used to parallelize a loop that performs a reduction on the array elements. For example, it can be used to compute the summation of the elements of an array. A `blocked_range` object is also used to retrieve the interval of indices on which a task applies but the programmer defines two lambda functions, one for defining the task as in the `parallel_for` pattern, and one to define what operations to perform for the reduction of two tasks.

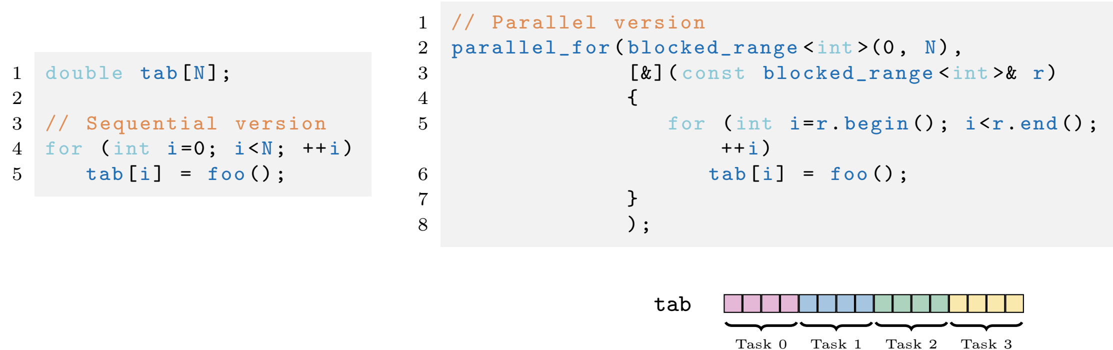

Figure 3.2 | Sequential for loop (left) and equivalent version with TBB `parallel_for` (right). The array is implicitly decomposed into 4 tasks by the TBB scheduler.

One of the advantages of the TBB library is that the programmer can easily define parallel algorithms thanks to the lambda functions. It is for example possible to apply complex reduction patterns or to manipulate several arrays together in the same task. The parallel patterns create tasks implicitly and the call to a parallel region is blocking: all the tasks created during a parallel region have to be executed before proceeding to the next computations.

Explicit Task Creation

Tasks can also be created explicitly, the programmer being in charge of defining the task, submitting it and waiting for its completion if necessary. In Figure 3.3, a task is spawned by the master thread (line 16). The master thread can spawn a task and perform computations asynchronously with the task execution. A call to the `wait_for_all` function (line 22) indicates when the master thread needs the task to be completed and this call guarantees that the task is executed. The declaration of the computation inside a task is made by defining a class that derives from the `tbb::task` class and by overloading the `execute` function. A task may create other tasks implicitly or explicitly and the programmer defines if the task should wait for the completion of children tasks thanks to the `wait_for_all` function.

When explicitly spawning tasks that will in turn spawn other tasks, TBB is in charge of maintaining a Directed Acyclic Graph (DAG) with the dependencies between the tasks. The programmer is not responsible for the DAG generation. The advantage of this explicit task creation is that the programmer has the control over what is inside a task and when it is created. More-

```
class HelloTask: public tbb::task
{
  HelloTask() {};
  tbb::task* execute()
  {
    std::cout << "Hello World!" << std::endl;
    wait_for_all();
    return NULL;
  }
};

int main()
{
  // Spawn the task
  HelloTask* t = new( tbb::task::allocate_root() ) HelloTask();
  task::spawn(*t);

  // Perform computations that do not need the task completion
  ...

  // Wait for task completion
  t->wait_for_all();
}
```

Figure 3.3 | Explicit task creation thanks to the TBB task API.

over, contrarily to the implicit task creation, the task spawning is not blocking and it is possible to execute several portions of code asynchronously.

Flow Graph

It is also possible to explicitly describe an application as a set of dependent tasks linked thanks to a DAG. This is the *flow graph* feature of TBB. Figure 3.4 shows the definition of a simple flow graph G. First, the nodes are created thanks to lambda functions (lines 3 and 9) and the dependency between the two nodes is expressed with the `make_edge` function (line 15). The nodes communicate with each other through messages that contain data or just synchronization information. A node is executed upon reception of the messages of its predecessors. In particular, the execution of the graph begins with the `try_put` function (line 16) that gives a message to the first node in the graph. TBB provides the `continue_msg` empty class when data are not explicitly passed. A node that receives a `continue_msg` knows that its predecessor has completed. The whole graph execution is finished after the `wait_for_all` call (line 17).

TBB provides different kinds of nodes depending on what they need as inputs, what they need to output and what kind of computations they perform. For example, `source_node` are used to generate data for their consumers, `continue_node` are used when no explicit data is required from the predecessors and `function_nodes` are used when data are consumed in input ports and produced in output ports. The computations performed by the nodes may also include implicit and explicit task creation and TBB is in charge of scheduling the graph given the explicit definition of the dependencies.

```
1  int main() {
2    tbb::flow::graph G;
3    tbb::flowcontinue_node< tbb::flow::continue_msg >
       hello( G,
4    []( const tbb::flow::continue_msg &)
5    {
6      cout << "Hello";
7    }
8    );
9    tbb::flow::continue_node< tbb::flow::continue_msg >
       world( G,
10   []( const tbb::flow::continue_msg &)
11   {
12     cout << " World\n";
13   }
14   );
15   make_edge(hello, world);
16   hello.try_put(continue_msg());
17   G.wait_for_all();
18   return 0;
19 }
```

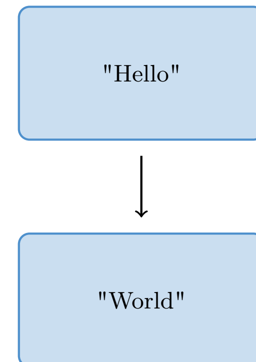

Figure 3.4 | Definition of a TBB flow graph composed of two nodes. The nodes do not exchange data but synchronize through a `continue_msg` message.

3.1.2 TBB Resource Management

TBB provides a task-based programming language and also comes up with a scheduler for the task execution. At the beginning of the program, an object `task_scheduler_init` is initialized with a number of threads, n. TBB creates $n-1$ worker threads so that the total number of threads running concurrently (worker threads and the *master thread* that initialized the TBB scheduler) will never exceed n. By default, the number of threads is set to the number of logical cores in the processor, N, but it is possible to set a smaller number of threads. It is also possible to set $n > N$ but TBB will not create more worker threads than the number of logical cores to avoid core over-subscription. Since TBB 3.0, the scheduler implements a *lazy thread creation*. The threads are not created during TBB initialization but when the first task is spawned. This way, the worker threads are actually created when they are necessary. The worker threads created by the scheduler are destroyed only at the end of the program. They are kept during the lifetime of the parallel program, even if they do not have tasks to execute.

Work Stealing

TBB uses a work stealing scheduler to execute the tasks. The work stealing concept has already been presented in Chapter 2 and we detail here the way this concept is implemented in TBB. Each thread has its own `deque` of spawned tasks ready to be executed. When the tasks come from templated functions (`parallel_for` or `parallel_reduce` for example), the threads get *chunks* of work corresponding to the interval of the loop indices on which it should work. When a thread gets a chunk of work (see Figure 3.5), it divides it into two sub-chunks twice as small as the first chunk. It places one of the sub-chunks at the beginning of its own `deque` (Task 0 in Figure 3.5) and divides again the other sub-chunk into two smaller chunks. One of the chunk is inserted at the end of the thread `deque` (Task 1) while the other is again sub-divided and so forth

until the task granularity specified by the programmer has been reached. More details about the task granularity will be given in Section 3.1.3. At that point, the thread deque is composed of large tasks at the beginning and small tasks at the end.

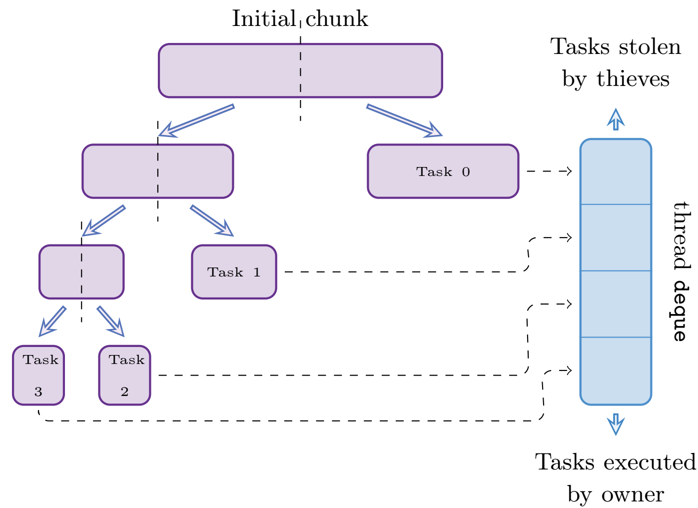

Figure 3.5 | Sub-division of a chunk of data by a thread to fill its deque of ready tasks.

The thread begins the task execution with a task popped at the end of the thread deque. After completing the execution of the task, the thread chooses the next task to be executed according to several rules. If the task returns another task or if the task has a successor, the thread executes the successor task. If there is no successor, the thread executes the task at the end of its deque. When all the tasks of its deque have been executed, the thread invokes the stealing mechanism on a random thread [63] and steals a task from the beginning of the deque of the victim. The execution strategy adopted by TBB minimizes the cost of the stealing operations in two ways. First, the stealing occurs only when necessary, that is to say when the deque of a thread is empty. Secondly, the threads steal at the beginning of their victim's deque, where large chunks of work are stored. These large chunks are likely to generate local work, preventing the threads to steal often.

Code Composability

Over the years, TBB has been enhanced with new features to improve the composability of TBB applications. In particular, TBB team wanted to add features to enable the efficient execution of programs where several threads explicitly created by the programmer co-exist in the same application. In this case, we use the term *master thread* to refer to the application main thread and to the threads explicitly created by the programmer.

Before TBB 3.0, the different master threads shared the same pool of worker threads and submitted tasks in the same task queues. In this implementation, master threads could execute tasks submitted by another master thread. However this approach showed limitations because a master thread could get stuck in another master thread task execution instead of performing its own computations. This is why TBB 3.0 introduced new concepts and in particular the creation of

arenas associated to each master thread so that the work published by a master thread is invisible to the other master threads.

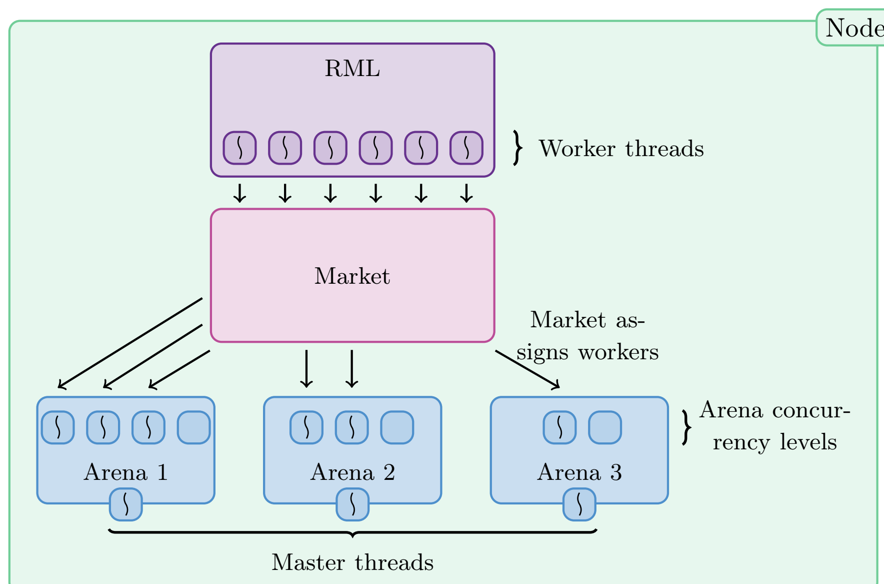

Figure 3.6 | TBB inner structure. The RML and the market are shared by the master threads and master threads submit tasks in their own arena. The RML contains here 6 worker threads and the market assigns to each arena a number of threads proportional to their requirement because the total number of threads requested by the masters (9 here) is greater than the number of worker threads in the RML.

TBB inner structures can be schematized by Figure 3.6. When a master thread initializes TBB for the first time (by a call to `task_scheduler_init`), a Resource Management Layer (RML) and a Market are created. The RML hosts the pool of TBB *worker threads* and the market assigns the worker threads to the different master threads. The limit of the total number of worker threads managed by the market is set to one less than the maximum between the argument passed to the `task_scheduler_init` object and the number of logical cores seen by TBB. An arena is then created for the calling master thread. A master thread submits tasks inside its own arena and the arena *concurrency level* determines the maximum number of tasks that can be executed simultaneously. Each arena is therefore assigned a number of slots corresponding to the number of worker threads that can take part in the tasks execution. The number of slots of an arena is set to one less than the minimum between the argument passed to the `task_scheduler_init` object and the number of workers allowed in the RML. When another master thread initializes a `task_scheduler_init` object, another arena is created for this master thread and the RML and market objects are shared by the master threads. This way, the master threads share the same worker thread pool but their tasks are submitted to distinct arenas. When the total number of worker threads requested by the different master threads is greater than the number of worker threads hosted by the RML, the market allots to each arena a number of worker threads proportional to each arena request. The library provides mechanisms to migrate threads from one arena to the other during the code execution to fulfill the arena concurrency levels.

The code composability feature of TBB makes it a good candidate for the implementation of a task-based in situ framework. Simulation and analytics tasks can be created concurrently in arenas of different concurrency levels, which allows to control the number of threads that

executes the different codes. Moreover, the mechanisms to migrate the threads from one arena to the other makes it possible to load balance simulation and analytics tasks during the code execution, hence reducing the thread idleness periods. These features will be the building blocks of the implementation of a dynamic helper core strategy more flexible than the static helper core strategy and will be discussed in Chapter 6.

3.1.3 Tools to Control the Task Execution

TBB provides different tools to control the task execution. We explain here two features of TBB, the *partitioners* and the *observer* to define the task execution policy and to give hints for the thread placement.

Partitioners

As explained above, when a thread gets a chunk of work, it splits it until it reaches the task granularity set by the programmer. This is done automatically by TBB thanks to the `blocked_range` object. It represents a half-open range that can be recursively split until reaching the sub-ranges corresponding to the task granularity. Special care must be taken when setting the task granularity to minimize the overhead of the task creation and scheduling. Setting a too small granularity will lead to a lot of small tasks and the overhead of the task creation and execution will be more visible. On the contrary, a too large granularity leads to large tasks for which the stealing mechanism cannot be used efficiently because the large tasks cannot be further split and the work cannot be shared by the threads.

TBB provides heuristics to automatically find the best task granularity but the programmer can also control the granularity by themselves. To do so, two parameters can be used: a *grainsize* g can be set in the `blocked_range` object and a *partitioner* can be used in the `parallel_for` or `parallel_reduce` functions. Three partitioners exist and have different behavior, in particular for the task granularity:

- `auto_partitioner` is the default partitioner of TBB whose goal is to minimize the number of sub-divisions while still allowing load balancing. The threads initially get the same amount of work, the range being split in a number of sub-ranges proportional to the number of threads. The sub-ranges are subdivided again only when load balancing is necessary. When the programmer has set a grainsize g, the `auto_partitioner` guarantees that the task granularity is not smaller than $g/2$. The advantage of the `auto_partitioner` is that the overheads of the task creation are small because the task granularity is kept high most of the time but load balancing is still possible because the task granularity can be reduced to enable load balancing;

- `simple_partitioner` specifies that the range should be sub-divided until it cannot be sub-divided further. In this case, the granularity is such that $g/2 \leq$ granularity $\leq g$ and the grainsize chosen by the user has a great importance. This partitioner should be used when the developer wants to choose the task granularity and wants to have a control over the TBB scheduler. However, it must be used with care because of the impact of the grainsize choice on the code performance;

- `affinity_partitioner` is used to take better benefit from the caches. The partitioner tries to assign the same tasks to the same threads from one iteration to the other to opti-

mize for cache affinity. The granularity is set automatically by the partitioner. When the programmer sets a grainsize g, the `affinity_partitioner` just guarantees that the task granularity is always greater than $g/2$. This partitioner is used when the same functions are executed from one iteration to the other, as this is the case for most of the simulation codes.

Observer

TBB provides a high-level interface for the task creation and hides the low-level thread management. However, it is sometimes helpful to have information about thread execution or to give information to the scheduler for thread placement or priority for example. The `task_scheduler_observer` (*observer* in the following) can be used for that purpose.

An observer is an object that detects when a thread starts or ends taking part in the task scheduling. The `on_scheduler_entry` method is called when the thread enters for the first time a parallel region. By defining a class that derives from `task_scheduler_observer`, and by implementing the `on_scheduler_entry` method, it is possible to execute codes when the threads are first created. This can be used to set the thread affinity as we will see in Chapter 8. TBB also provides a preview mode that allows to define observers bound to different arenas. When using these observers, the threads call the `on_scheduler_entry` method when they enter an arena and the `on_scheduler_exit` method when they leave this arena. It can be used to monitor when a thread enters and leaves an arena or to apply affinity masks to the arenas, as it will be discussed in Chapters 5 and 6.

Intel® TBB is a C++ library that provides a task-based programming model and a work stealing scheduler. It provides different ways to create tasks, implicitly or explicitly, and good code composability properties that make it a good candidate for the implementation of a task-based in situ framework. In particular, its arena system can be used to create concurrently simulation and analytics tasks and to dynamically balance the number of threads that execute them. In the following section, we will present ExaStamp, a molecular dynamics code that uses the TBB library for its intra-node parallelization and that will be our target code for the integration of a task-based in situ framework.

3.2 ExaStamp, a Molecular Dynamics Code for Material Sciences

Classical molecular dynamics is a computational method to describe the evolution of a set of particles over time. The main principle behind molecular dynamics is the iterative integration of Newton's equations of motion for a set of particles, the force on one particle depending on its interaction with all other particles [11]. Molecular dynamics applications are widely used in three areas: material sciences, chemistry and biology. We introduce in this section ExaStamp, a molecular dynamics code dedicated to material sciences and in particular shock physics. ExaStamp targets manycore architectures and has been designed with three levels of parallelism to achieve high performance on these architectures. We first briefly explain the use cases of ExaStamp (Section 3.2.1) before going more in depth into ExaStamp architecture (Section 3.2.2).

3.2.1 Molecular Dynamics for Material Sciences

ExaStamp [29] is a modern molecular dynamics code developed at CEA since 2012. It is dedicated to material sciences and is especially used for shock physics [49]. Shock physics aims at understanding the behavior of matter under extreme conditions such as high pressure and high temperature. This kind of physics is present in different areas, from cosmology to understand meteorite impacts [96] to industrial fields for semiconductor research and for the design of new materials. One of CEA field of studies is the material deformation under shocks propagation. It can be used to study micro-jetting [48], micro-spallation [104] or phase transition for example. To understand finely the underlying physics, this kind of simulation requires a high number of particles, in the order of several billions, during times up to the nanosecond, corresponding to more than one million iterations.

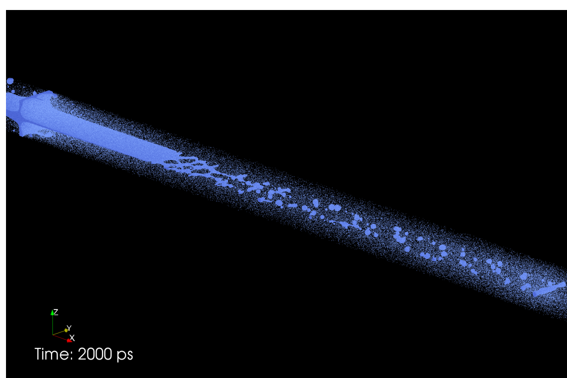

Figure 3.7 | Shock propagation inside a metallic crystal that leads to the generation of micro-jetting.

Molecular dynamics for material sciences differs from chemical and biological simulations in two ways. Biological and chemical simulations require fewer particles, usually around a few million particles, and the interactions between the particles are long-ranged. It means that the movement of a particle depends on all the particles of the simulation, which leads to complex interactions and parallelization. On the other hand, material sciences require hundreds of millions to billions particles but the interactions are most of the time short-ranged interactions, the interactions between the particles being neglected for particles that are distant from more than a *cutoff* distance. The parallelization of such a short-ranged system is highly simplified because it allows to divide the domain into blocks, as will be seen in the following section.

3.2.2 ExaStamp Architecture

ExaStamp is written in modern C++11 and uses three levels of parallelism: MPI for inter-node parallelism, Intel® TBB for intra-node parallelism and explicit vectorization. It uses a domain-decomposition approach (Figure 3.8) where the global domain is split on as many sub-domains as the number of MPI processes and each MPI process is assigned to a sub-domain. Each MPI process thus holds a portion of the particles and the particles are split into a hierarchical data structure composed of four main objects (implemented as C++ classes). Each MPI process holds a `Node` which is in turn composed of one or several `Domains`. The `Domain` is itself composed of a `Grid` of `Cells` where the particles are stored.

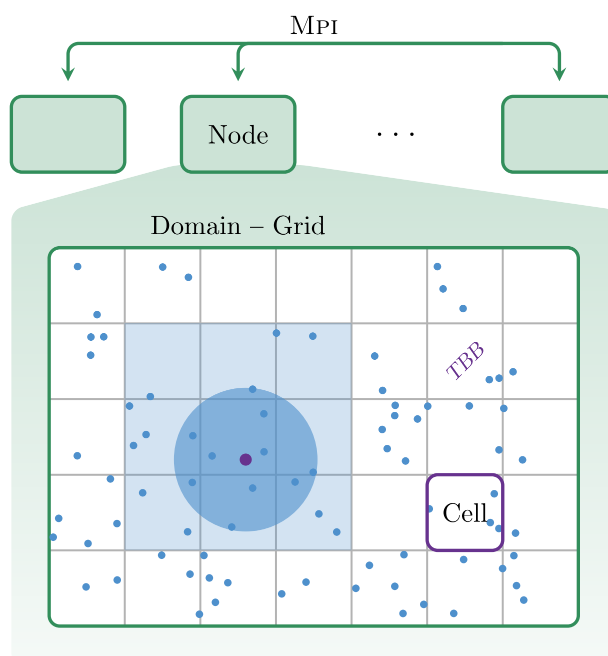

Figure 3.8 | ExaStamp architecture (adapted from [29]). Each MPI process holds a `Node` composed of one or several `Domains` (only one `Domain` depicted here). A `Domain` is in turn composed of a `Grid` of `Cells`.

Hierarchical Data Structure

The `Node` is the highest object of ExaStamp and corresponds to the highest interface with the developer. In particular, it defines the `doComputeWork` function that describes the sequence of operations to be performed to compute an iteration (Figure 3.9). After an initialization phase, the function enters a `while` loop that ends when reaching the desired number of iterations. The computation of an iteration is decomposed in three steps. First, an integration scheme is used to update the positions of the particles based on the particles configuration of the previous iteration (`oneStep`, line 12). A load balancing step is then performed if necessary (`balance`, line 15). Finally, data are output into the filesystem at a user-defined frequency (`writeIO`, line 18). The `Node` also holds a communication manager in charge of MPI communications. The different MPI communication patterns (point-to-point and collective) are encapsulated in methods managed by the communication manager. In particular, the programmer can define their own custom types and does not need to care about the size of their messages, everything being hidden by the communication manager.

A `Node` is composed of a `Domain` that sits on top of a multi or manycore processor. The parallelization inside a `Domain` is made thanks to Intel® TBB. ExaStamp has been designed so that a `Node` may be comprised of several `Domains`. Thus, there would be one `Node` per compute node and one `Domain` on each NUMA node, inter `Node` communications being replaced by copies to reduce the communications. However, this feature has not been implemented yet and we use ExaStamp with one `Domain` per `Node`.

A traditional iteration in molecular dynamics consists in a loop over the particles and for each particle, a second loop on the particles to compute the force on this particle, the force depending on the other particles. As we have already seen above, ExaStamp is designed for material sciences

```
void Node::doComputework()
{
  initialization();
  // Time loop
  while( !time->isFinished() )
  {

    // increment time
    ++(*time);

    // One iteration
    oneStep();

    // Load balancing
    balance();

    // end of iteration
    writeIO();
  }
}
```

Figure 3.9 | Timeloop definition in the `Node` class.

where the interactions between the particles are most of the time short-ranged interactions. To illustrate this, let us consider the purple particle in Figure 3.8. The particles that interact with this purple particle are the ones in the blue circle of radius the cutoff distance. Instead of looping over all the particles in the `Domain`, ExaStamp organizes the particles into a `Grid` of `Cells`. A `Cell` is composed of a few particles and the size of a `Cell` is a bit greater than the cutoff radius so that the neighbors of the purple particle are searched in the cell that holds this particle and in the 8 neighboring cells (26 in 3D).

The particles are therefore stored according to an AOSOA (array of structures of arrays) scheme. Each `Domain` is composed of an array of `Cell` objects. A `Cell` object is in turn a structure of arrays composed of several arrays. Their size is the number of particles in the `Cell` and they correspond to the attributes of the particles. There are mostly 11 important attributes: the global indexes, the types, and the positions, velocities and forces along the three axes. This AOSOA structure makes the retrieval of particles attributes complicated. Indeed, an attribute is not seen as a contiguous array of size the number of particles but as several contiguous arrays of sizes the number of particles in each cell.

Because the particles depend on the positions of their neighbors for the force computation, each `Grid` also holds a layer of ghost cells that correspond to the cells hold by other `Domains`. The ghost update is made after each iteration thanks to MPI communications. For each `Cell`, the `Grid` also stores a list of neighboring cells, some being `Cells` actually hold by the `Grid` and some other belonging to the ghost layer.

Time Integration

ExaStamp proposes several integration schemes. The most widely used is called *verlet integration scheme* and is described in Figure 3.10. The positions of the particles at iteration $i + 1$ are first updated based on the positions, velocities and forces of the particles at iteration i. The velocities are then updated of half an iteration, the velocities at iteration $i+1/2$ requiring the velocities and

forces at iteration i. The forces of iteration $i+1$ are then computed using the positions at iteration $i+1$. The velocities of iteration $i+1$ are finally updated thanks to the velocities at iteration $i+1/2$ and the forces at iteration $i+1$. The main advantage of this approach is that the code does not need to use a *double buffering* technique where two copies of the particles states are maintained, one reflecting the state of iteration i and the other using data at iteration i to compute the state at iteration $i+1$. Here, we do not need to maintain two copies of the data because the modifications of the $i+1^{\text{th}}$ iterations can be made in place without overwriting necessary data.

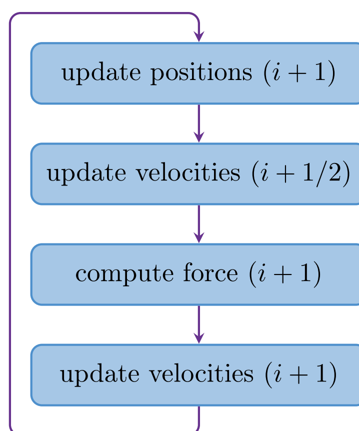

Figure 3.10 | ExaStamp iteration computation using a verlet integration scheme.

ExaStamp uses Intel® TBB for the intra-node parallelization by using `parallel_for` functions to implement a *fork-join* pattern. The code alternates between sequential regions where the master thread is the only one to execute and parallel regions where master and worker threads share the tasks execution. The parallelization is made over the `Cells` of the `Grid`, each task corresponding to the update of the particles in a set of `Cells`. The loops are parallelized given an affinity partitioner whose goal is to assign tasks in a way that optimizes cache affinity. This way, ExaStamp can run with one MPI process per node, the TBB scheduler being in charge of optimizing data locality on the NUMA nodes.

I/Os

ExaStamp is parametrized thanks to a text input file. The end-user indicates parameters for the potential, the number of iterations, the MPI splitting and so forth thanks to a key-value file read at initialization by the simulation code.

ExaStamp provides different data formats for output data: a binary MPI-IO and Hercule data format to use analytics tools developed by CEA [104], ASCII XYZ files to be used with molecular dynamics software (VMD [62], Ovito [106], ...) or VTK data format for data visualization with ParaView. The file format as well as the output frequency are chosen by the user prior to the simulation in the input file. When the simulation reaches an iteration of output, the attributes of the particles (index, type, positions and velocities) are copied into a temporary buffer and written in the desired file format. The copy is necessary for two reasons. First, data output is managed by the `Node` object but data are actually stored in the `Cell` object. Secondly, the particles attributes are stored as arrays of pointers managed by the `Grid` object. It is therefore not possible to simply pass a pointer to the necessary data to the functions that write the files.

File output is natively performed synchronously with the simulation, without overlapping between computation and output. The frequency of output therefore has a great impact on the total execution time. Figure 3.11 shows the total execution time of 1,000 iterations of ExaStamp, writing a MPI-IO file at different frequencies: a file is output every 1,000 iterations for the left bar while a file is output every 10 iterations for the right bar. We see that the time to write the files every 10 iterations corresponds to 41% of the total execution time. The frequency of output should be chosen based on the physics at stake: the more the system evolves rapidly, the more frequent the data analytics should be performed. However, data output has a cost and ExaStamp is also an example of a code where the user needs to find a balance between physical and computational properties.

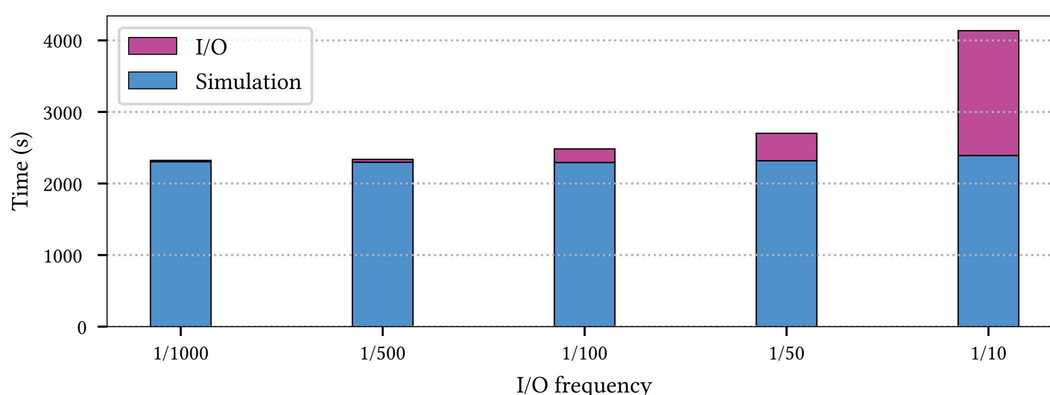

Figure 3.11 | Total execution time of ExaStamp when increasing the I/O frequency on a simulation of 1,000 iterations with 256,000,000 particles on 64 MPI Broadwell processes (1,792 cores). The output is a unique MPI-IO file per iteration.

ExaStamp is a new code and does not provide in situ capacities yet. In particular, no analytics have been developed inside the simulation code and no framework has been developed to execute analytics synchronously or asynchronously with the simulation. This makes it a good candidate for studying the integration of a task-based in situ framework inside a molecular dynamics code. In particular, we can study different task-based approaches, from the explicit spawning of analytics tasks by the simulation to more advanced dynamic helper core strategy where simulation and analytics codes are decoupled. However, the integration of in situ capacities inside the simulation code remains a challenge in particular because of the performance of ExaStamp on modern supercomputers, as it will be discussed in the following section.

3.3 Challenges for the Integration of an In Situ Framework Inside ExaStamp

ExaStamp has been designed for exascale supercomputers and in particular for architectures that provide a large number of cores per processor. We briefly present in this section the two supercomputers that ExaStamp targets (Section 3.3.1) and the performance of ExaStamp on these architectures (Section 3.3.2). We conclude by proposing ideas for the integration of a task-based in situ framework inside ExaStamp (Section 3.3.3).

3.3.1 Target Architectures

During this thesis, we use two supercomputers hosted by CEA and the French CCRT research center: Cobalt, a Broadwell supercomputer and Tera-1000-2, a KNL supercomputer.

Cobalt Supercomputer

Cobalt is a supercomputer hosted by the CCRT (French Research and Technology Center) since 2016. In June 2018, it was ranked 153th in the Top500 list and reaches 1,500 TFlop/s. The supercomputer is composed of 1,419 nodes for a total of 39,732 cores. The nodes are interconnected thanks to a EDR InfiniBand network. Each node of Cobalt has a total of 128GB of memory and is composed of two Intel® Xeon Broadwell processors running at 2.40GHz. Each Broadwell processor is composed of 2 NUMA nodes with 7 physical cores each. Figure 3.12 shows the composition of a NUMA node. Each physical core can host two logical cores that share the L1 and L2 caches. The L3 cache of 18MB is shared by the 7 physical cores and the NUMA node has 32GB of memory. A Broadwell node is therefore composed of a total of 28 physical cores. ExaStamp is compiled on Cobalt using `icpc` compiler (version 17.0.4.196) and is launched with Intel® MPI (version 2017.0.4.196) that supports `MPI_THREAD_MULTPLE`.

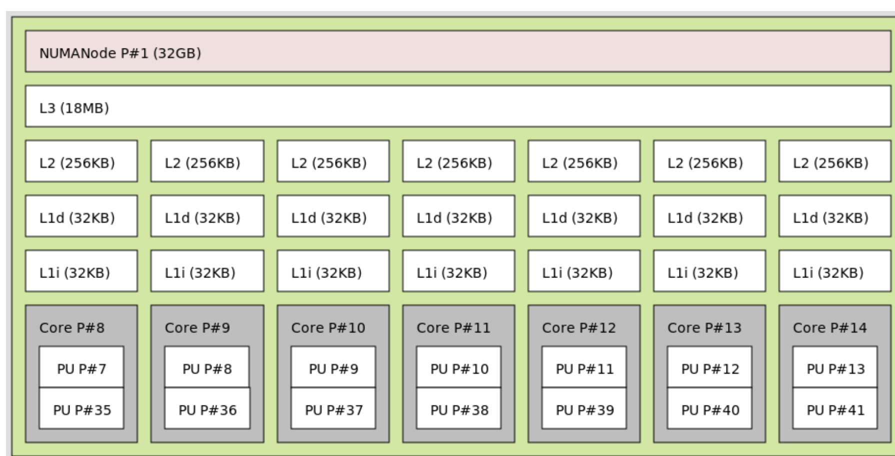

Figure 3.12 | Composition of the NUMA node 1 of a node of the Cobalt supercomputer.

Tera-1000-2 Supercomputer

Tera-1000-2 is a supercomputer hosted by CEA since 2017. In June 2018, it was ranked 14th in the Top500 list and reaches 25 PFlop/s. The supercomputer is composed of 8,256 nodes for a total of 561,408 cores. The nodes are interconnected thanks to a Bull BXI 1.2 network. Each node of Tera-1000-2 is composed of one Intel® Xeon Phi Knight Landing (KNL) processors running at 1.4GHz. Each KNL processor is composed of 68 physical cores and 4 physical cores are dedicated to the system. Figure 3.13 shows the arrangement of the 4 first physical cores of a KNL. The cores are organized into tiles with one L1 cache per physical core and one L2 cache shared by two physical cores. Each physical core can host 4 logical cores. The node has a RAM memory of 96GB and a MCDRAM of 16GB used as a fast cache and seen by the application as a second NUMA node in the processor. The cores are organized in a *quadrant* mode where the KNL can

be seen as one SMP node. ExaStamp is compiled on Tera-1000-2 using `icpc` compiler (version 17.0.4.196) and is launched with Bull MPI that does not support `MPI_THREAD_MULTPLE`.

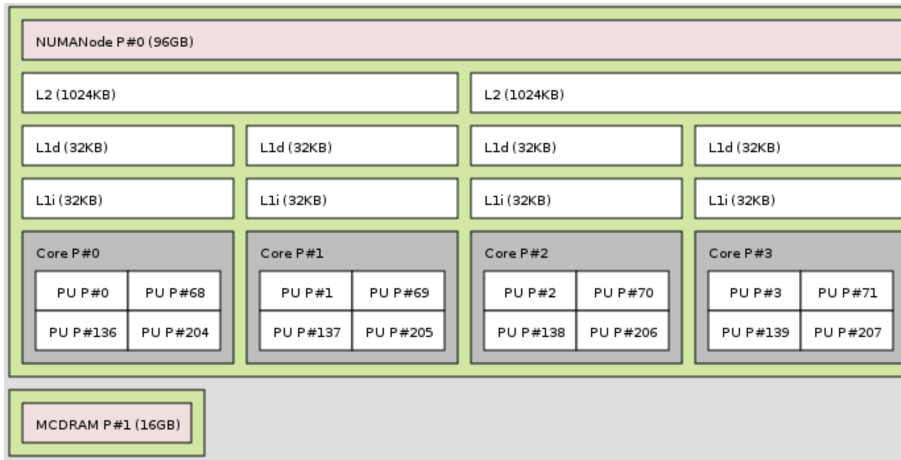

Figure 3.13 | 4 first physical cores of a KNL processor of the Tera-1000-2 supercomputer.

3.3.2 ExaStamp Performance

Figure 3.14 shows ExaStamp performance for two sets of measurements on the Cobalt supercomputer. For each experiment, we launch one MPI process per node and we use TBB for the intra-node parallelization on the 28 physical cores. We first measure in Figure 3.14 (left) the *strong scaling* of the code when using different numbers of threads with one MPI process on one node. The number of particles is set to 4,000,000 for all the measurements and we show the *speedup* $S_n = T_1/T_n$ where T_i denotes the time to execute the code on i threads. We then measure in Figure 3.14 (right) the *weak scaling* of the code when using different numbers of nodes. On each node we run one MPI process and the total number of particles is $4,000,000n$ where n is the number of MPI processes. We show the *efficiency* $E_n = 100 \cdot T_1/T_n$. Figure 3.15 shows the strong scaling of ExaStamp on one KNL node when varying the number of threads from 1 to 256.

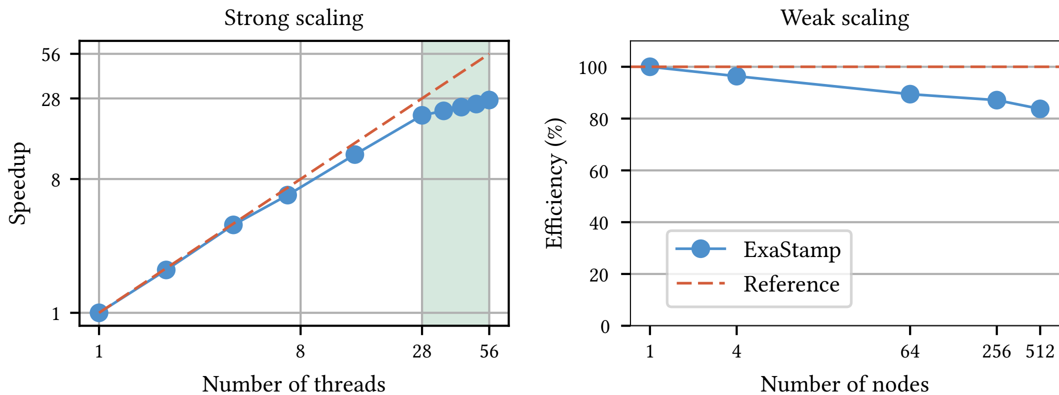

Figure 3.14 | Scaling of ExaStamp on the Cobalt supercomputer: strong scaling on one node with different numbers of threads (left) and weak scaling when varying the number of nodes (right). The green area in the left figure highlights the number of threads from which we pass into the hyperthreading area.

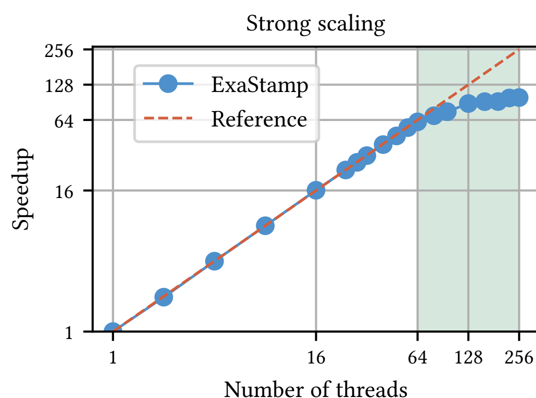

Figure 3.15 | Strong scaling of ExaStamp on one KNL node with different number of threads. The green area highlights the number of threads from which we pass into the hyperthreading area.

The performances of ExaStamp are very good on both architectures. The strong scaling is quasi-linear on the KNL processor, the speedup reaching 61 on 64 threads. The strong scaling is a bit less impressive on the Broadwell processor, the speedup reaching 22 on 28 threads. This is certainly due to the presence of NUMA nodes on the Broadwell processor. For both processors, the performances of ExaStamp reach a plateau in the hyperthreading area, demonstrating that hyperthreading cannot enhance the simulation performance. Regarding the efficiency, it is greater than 84% on a simulation of 2 billions atoms on 512 Broadwell nodes. The efficiency is very good but shows the impact of MPI communications at large scale.

The in situ techniques presented in Chapter 2 mainly focus on harvesting the wasted cycles of the simulation to execute analytics tasks instead. They either take benefit from the sequential regions of the simulation code (Goldrush approach) or use the fact that the simulation code is not efficient on a high number of cores (Damaris approach). ExaStamp is therefore not a target code for these approaches because it exhibits a quasi-linear speedup on a high number of cores and presents only small sequential regions. As we will see in more details in Chapter 5, approaches such as Goldrush does not manage to execute analytics during the short sequential regions of the code and approaches such as Damaris are very sensitive to the number of helper cores because of the efficiency of ExaStamp on a high number of cores. This calls for the integration of high performance in situ strategies inside the molecular dynamics code.

3.3.3 Ideas for the Implementation of a Task-Based Hybrid Framework

ExaStamp is a MPI+TBB code designed for exascale supercomputers and in particular for processors with a large number of cores. It uses Intel® TBB to create simulation tasks inside each MPI process using a fork-join model where tasks are implicitly created by the TBB library. ExaStamp shows very good performance on multi and manycore processors, its speedup reaching 22 on 28 Broadwell cores and 61 on 64 KNL cores. However, its synchronous file output makes difficult to output files at a high frequency without degrading simulation performance. While the good performance of ExaStamp make it difficult to take the best of traditional in situ techniques, the fork-join model used by the simulation still induces sequential regions that could be harvested to execute in situ analytics. Moreover, the simulation code is parallelized with TBB that provides

good code composability properties and a work stealing scheduler that may enable the efficient balance between simulation and analytics tasks execution.

As already explained in Section 1.3, the goal of this thesis is to study the implementation of a task-based in situ framework inside a task-based molecular dynamics code designed for exascale supercomputers. With its task-based programming model and its good performance on modern architectures, ExaStamp is a good candidate for this study. To solve the issues of traditional in situ techniques on a code such as ExaStamp, our idea is to leverage the work stealing scheduler and the good composability properties of TBB to implement a task-based dynamic helper core strategy more flexible than the state-of-the-art static helper core strategy. Instead of permanently dedicating resources to analytics execution, the idea is to dedicate resources to analytics only when simulation and analytics tasks exist concurrently and to remove this restriction when the simulation or analytics enter a sequential region or when all the analytics tasks have been executed. This way, we expect to benefit from the sequential regions of both simulation and analytics to execute in situ analytics with a low overhead on the simulation execution time. The idea is then to go one step further and to use the task-based programming model to express analytics workflows in the form of graphs of tasks. Each task corresponds to an analytics that can in turn create children tasks to be interleaved with simulation tasks in situ or to be executed alone in transit. This way, we expect to provide an intuitive and flexible environment for the development and deployment of complex analytics workflows by non-expert users.

The remaining of this document aims at presenting the different steps toward the integration of this hybrid framework inside the simulation code. Chapters 4, 5 and 6 are dedicated to the task-based in situ methods and Chapters 7 and 8 to the task-based hybrid framework.

Part II

Toward a Task-Based In Situ Technique

4 Turning a Synchronous In Situ into an Asynchronous Task-Based In Situ

Simulation codes usually output data periodically into the filesystem and these data are later read back for a post-processing step. However, file outputs lead to large amounts of data stored into the filesystem and have a negative impact on the simulation performance, limiting the high frequency of analysis of data at large scale. ExaStamp is an example of simulation code that exhibits very good performance on multi and manycore processors. However, its performances are severely impacted by the synchronous file output that it implements. As a new code, ExaStamp does not provide any features to analyze data in situ with the simulation. In this chapter, we propose to first integrate a synchronous approach in ExaStamp, to show the relevance of in situ processing compared to the traditional file output and to be used as a comparison point for the different task-based in situ modes we will develop in the following (Section 4.1). Although the simulation exhibits good performance on multicore processors, we show that it still presents periods where resources are unused, showing that the code could benefit from an asynchronous approach (Section 4.2). We finally implement a first task-based asynchronous method where an analytics task is spawned by the simulation (Section 4.3). This chapter aims at implementing a first task-based approach but also at introducing the tools that will be used throughout the manuscript to evaluate the different in situ techniques.

4.1 Integration of Analytics for Synchronous Execution

Synchronous in situ is an intuitive first step toward the integration of an in situ framework inside a simulation and most of the simulation codes already have some analytics coded inside [56, 97]. For example, in molecular dynamics, the pressure and the temperature are periodically computed and output to log files. These quantities are not necessary for the progress of the simulation itself but are useful for the end-user to check the status of the simulation and to know if something went wrong with the execution. We integrate in this section a synchronous in situ framework inside ExaStamp (Section 4.1.1) and we present a set of analytics developed inside the simulation (Section 4.1.2). The goal of this synchronous in situ implementation is twofold. First, we want to show that writing files at a high frequency has a larger impact on the simulation performance than synchronous in situ processing for various analytics (Section 4.1.3). Secondly, the synchronous approach will serve as a comparison point to evaluate the different in situ approaches presented in this manuscript.

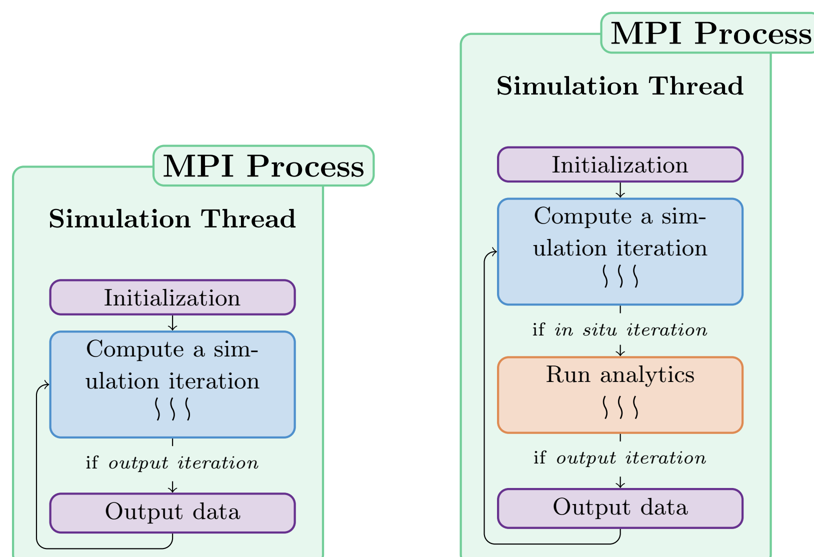

Figure 4.1 | ExaStamp main loop with file writing only (left) and with file writing and synchronous analytics (right). Synchronous analytics are executed with a higher frequency than file writing.

4.1.1 Integration of Synchronous In Situ in ExaStamp

As already seen in Chapter 3, ExaStamp can be schematized by Figure 4.1 (left). Inside each MPI process, a simulation master thread first performs an initialization phase. After this step, the simulation master thread enters an iteration loop where the particles positions, velocities and forces are updated at each iteration based on the configuration of the particles at the previous iteration. When the simulation reaches an output iteration, data are written into the filesystem in the file format asked by the user.

A synchronous in situ execution can be obtained by interleaving a call to an analytics routine between the end of the iteration and the file output (Figure 4.1 right). We propose to use in situ analytics as a complement of the post-processing approach: data are still written into the filesystem but with a lower frequency such that the impact on simulation performance is low while analytics are executed with a higher frequency. The main advantage of this synchronous approach is that the analytics can directly use ExaStamp data structures without a copy to be performed by the simulation. If the analytics are parallelized with TBB, they can also be executed by the worker threads instantiated by the scheduler. The main drawbacks are that the analytics need to be coded inside ExaStamp and that they need to know ExaStamp data structure.

4.1.2 Implementation of Analytics Routines inside ExaStamp

In this manuscript, we will describe the different steps toward the implementation of a task-based in situ processing framework. To validate the different approaches, we need to have a set of analytics benchmarks that are representative of computational physics and that show different patterns in term of parallelization, MPI communications and memory effects. We have chosen the four analytics summarized in Table 4.1 and described below. The analytics are used for in situ processing, using the same resources than the simulation. We therefore choose analytics that use the same MPI splitting than the simulation and whose execution times are smaller or

equivalent to an ExaStamp iteration. We make this choice because more time consuming analytics are more adapted to in transit processing because they would have access to more computing resources without removing computing resources from the simulation. In transit processing will be discussed in Chapter 7.

Table 4.1 | Analytics benchmarks to evaluate in situ methods. The analytics are representative of computational physics and show different patterns in term of parallelization, communication and memory effects.

Analytics	Description	Specificity
`statistics_seq`	Compute the mean of the positions for the particles inside each MPI process	- Sequential analytics - Memory intensive
`statistics_par`	Compute the mean of the positions for the particles inside each MPI process	- Local parallel computations: 1 TBB parallel reduction - Memory intensive
`radial`	Compute a local radial distribution function for the particles inside each MPI process	Local parallel computations: 2 nested TBB parallel for
`histogram`	Compute a global histogram of the x-positions	- Local parallel computations: 2 TBB parallel reductions - Global parallel computations: 2 MPI reductions

The two statistics routines perform local computations without any MPI communications. They both compute the mean of the positions of the particles that belong to the MPI process where the analytics is executed. We implemented a sequential version (`statistics_seq`) and a version parallelized with TBB (`statistics_par`) thanks to a parallel reduction. The goal of `statistics_seq` is to evaluate our system on a sequential analytics because the analytics are often not as parallelized as the simulation code used to produce the data [83, 26]. This analytics scans the three positions of each particle and makes only a few summations on them, making it a memory intensive analytics. The statistics computations are thus very sensitive to cache effects and in particular to NUMA effects. To stress even more the memory accesses, it is possible to run this analytics several times at each in situ iteration. If not stated otherwise, the mean is computed 1,000 times at each in situ iteration.

The `histogram` analytics provides two levels of parallelization: internally with TBB and between the processes with MPI. This analytics counts how many particles have a position in intervals of the form $[x_i, x_i + \Delta x]$. Histograms are often used in molecular dynamics [62, 83]. Here, we count the number of particles inside intervals of the form $[x_i, x_i + \Delta x]$ but computing the mean velocity of these particles instead allows to follow the propagation of wave fronts. These generalized histograms will be used in Chapter 8 for the study of a tin phase transition under shock. The algorithm is in three steps:

1. Determine the global bounds of the domain: each MPI process computes its own minimum and maximum positions with a TBB parallel reduction. The global bounds are then

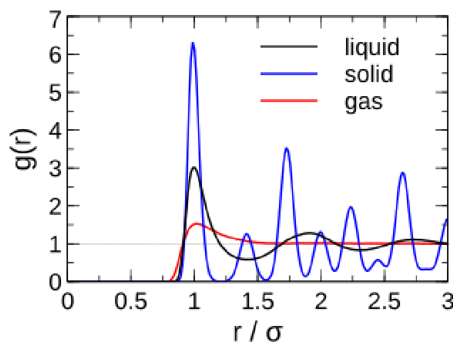

Figure 4.2 | Radial distribution function for different types of matter[1].

computed thanks to a collective `MPI_Allreduce`;

2. Compute a local histogram: the global domain is split in intervals of the form $[x_i, x_i + \Delta x]$. With a TBB parallel reduction, each MPI process can count how many of its particles belong to the different intervals;

3. Compute the global histogram: the global histogram is obtained by summing all the local histograms thanks to a collective `MPI_Reduce`.

The histogram is computed on 1,000 intervals. To inspect the influence of MPI communications, we can artificially increase the size of the arrays sent through the second `MPI_Reduce`. If not stated otherwise, the size of the array is set to 100,000,000 integers.

Finally, the `radial` analytics computes a local radial distribution function (RDF). The radial distribution function $g(r)$ is a commonly used function in molecular dynamics [120] and describes the probability of finding a particle at a distance r from a reference particle. The RDF is a measure of the microscopic structure of the matter and can be used for example to determine if the matter is a solid, a liquid or a gas, because they present distinct patterns (Figure 4.2). The computation of the RDF is straightforward if all the data are available in a single MPI process. Indeed, it consists in computing an histogram of the distances between all the pairs of atoms. However, a distributed version of the algorithm requires a lot of data exchanges and makes it an analytics way longer than an ExaStamp iteration. Because we already have the `histogram` to measure the influence of MPI communications, we extract from this analytics a kernel that only computes a local radial distribution function and does not perform MPI communications. This kernel as such does not have a physical meaning and corresponds to the computation of an histogram of all the pairs of atoms present in each MPI process. The kernel is multithreaded with two nested for loops and uses the `blocked_range2d` feature of TBB. This algorithm is used to show the effect of a compute intensive analytics.

4.1.3 Comparison of the Synchronous In Situ and the File Output Approaches

For an ExaStamp simulation of 1,000 iterations, Figure 4.3 compares the total execution times of ExaStamp with file output only (`ExaStamp-file`) and ExaStamp with file output and synchronous in situ (`ExaStamp-insitu`) for different frequencies and for two analytics (`histogram` and `statistics_seq`). In the first case, we look at the execution time when increasing the file

[1] `https://en.wikibooks.org/wiki/Molecular_Simulation/Radial_Distribution_Functions`

output frequency, as it was done in Figure 3.11. In the second case, a file is output every 500 iterations and we look at the execution time when increasing the in situ analytics frequency.

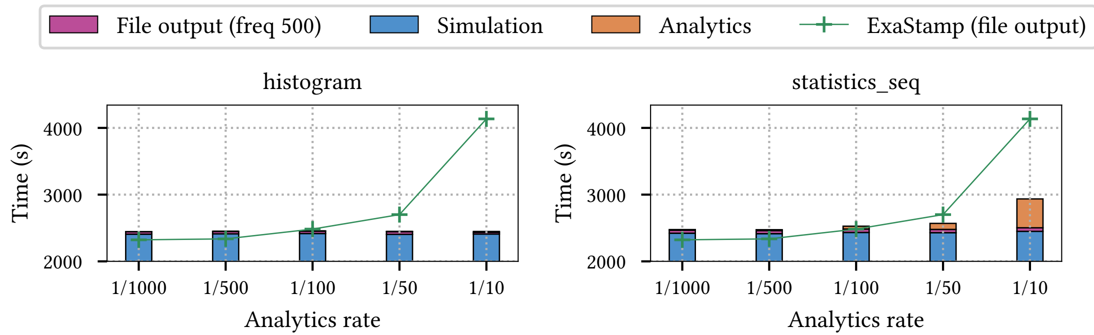

Figure 4.3 | Comparison of the total execution time of ExaStamp with file output only (green curve) and ExaStamp with file output and synchronous in situ analytics (bars) for two analytics: `histogram` (left) and `statistics_seq` (right). For ExaStamp with file output only, the file output frequency increases in abscissa. For ExaStamp with file output and synchronous in situ processing, a file is written every 500 iterations and the in situ analytics frequency increases in abscissa. ExaStamp simulation of 256,000,000 particles on 64 Broadwell nodes (1,792 cores) for 1,000 iterations.

For low frequencies (analytics every 500 and 1,000 iterations), `ExaStamp-insitu` has a longer execution time than `ExaStamp-file`. This is expected because a file is output every 500 iterations and synchronous computations are added in the first case while a file is output every 500 or 1,000 iterations without any extra computation in the second case. When the output frequency increases, the total execution time increases very quickly when writing files. If a file is output every 10 iterations, the time to write the files corresponds to 41% of the total execution time. On the other hand, the execution time of the synchronous analytics increases more slowly and the time to perform synchronous execution of analytics never exceeds 15% of total execution time.

Another aspect is not shown here: the disk memory needed to store the files. The simulation is composed of 256,000,000 particles and one MPI-IO file corresponds to 41GB. Storing a file every 10 iterations on a simulation of 1,000 iterations therefore requires approximately 4TB of data to be stored. On the other hand, the synchronous in situ approach allows analytics execution with only 82GB of data stored because data are output every 500 iterations only.

Notice that we did not measure here the time necessary to read the data file and to run the analytics in a post-processing way. Even without this measurement, we can see the benefit of the synchronous in situ approach compared to the post-processing approach, both in term of computation time and of memory usage.

In this section, we have implemented a set of analytics inside ExaStamp and we have executed them synchronously with the simulation to see the benefit of in situ processing compared to traditional file output. We will see in the following section that both simulation and analytics exhibit regions of unused resources, leaving the opportunity for more efficient in situ techniques.

4.2 Highlighting Periods of Unused Resources

The total execution time is a good metric to evaluate the performance of an in situ method but looking into more details into the execution of the code enables to find ways to improve the methods. The goal of this section is to propose a system to monitor and visualize the thread usage (Section 4.2.1). This system is useful to study the thread usage and to understand the performance of multithreaded applications. Graphs from this tool will be used in this manuscript to illustrate limitations of some in situ techniques. We use this tool in this section to highlight the sequential regions induced by the synchronous execution (Section 4.2.2).

4.2.1 Implementation of a Task Monitoring System

As already seen in Chapter 3, ExaStamp uses the Intel® TBB library that has a task-based programming model. To monitor the thread usage, we need to know when the different threads execute tasks and when they are idle. We thus propose to measure for each task, the beginning date, the end date and the thread that executes it. This way, we can create Gantt diagrams (as in Figure 4.5 for example) to visualize the tasks execution on each thread along the time. We rely here on the Pajé trace file format [33] that describes the code execution as a set of events. An event is defined with three variables:

- the starting time of the event measured with respect to the starting date of the program;
- the identifier of the thread that starts this event;
- the type of the event. We distinguish three types: start of a simulation task execution, start of an analytics task execution and start of an idle period (i.e. end of a task execution).

Figure 4.4 gives a code snippet of the instrumentation of a TBB parallel region. We define an array of events as a thread local storage so that each thread can update it without data race. The parallel region is defined with a lambda function (line 7) that represents a task. At the beginning of the task, a reference to the thread local storage is retrieved and we store the event with the starting time of the task, the ID of the thread executing it and an integer to tell the type of the task (1 for a simulation task, 2 for an analytics task). At the end of the lambda function, we store the event with the ending time of the task, the ID of the thread that executed it and the 0 integer to tell that the thread now starts an idle period.

The instrumentation of ExaStamp requires only a few code modifications because the TBB functions are encapsulated in ExaStamp functions and the code itself does not make direct calls to TBB. We just need to distinguish when a function was called for a simulation task or an analytics task. For sequential analytics, we define two events at the beginning and the end of the analytics execution respectively. At the end of the program, the simulation master thread goes through the thread local storage of all the threads and writes a Pajé file with all the events information. The visualization is then made possible thanks to the ViTE software [30].

4.2.2 Measure of the Thread Usage in the Synchronous Approach

Figure 4.5 shows the traces of the synchronous in situ execution of `statistics_par` and `statistics_seq` on a Broadwell node for two iterations of ExaStamp. The blue (resp. orange) areas

```
1  // Thread local vector to store the different events
2  typedef std::vector<std::tuple<double, int, int> > eventVector;
3  tbb::enumerable_thread_specific<eventVector> threadTiming;
4
5  template <class I, class J, typename Lambda>
6  tbb::parallel_for( tbb::blocked_range<J>(begin, end),
7                     [&](const tbb::blocked_range<J>& r)
8                     {
9                       // Store the beginning of the task in the thread local
                           storage
10                      tbb::enumerable_thread_specific<eventVector>::reference
11                         myTimer = Global::threadTiming.local();
12                      myTimer.push_back(std::make_tuple(startTime, gettid(), 1));
13
14                      // Lambda execution
15                      lambda(r.begin(), r.end());
16
17                      // Store the end of the task in the thread local storage
18                      myTimer.push_back(std::make_tuple(endTime, gettid(), 0));
19                    }
20                    );
```

Figure 4.4 | Instrumentation of a TBB parallel region to monitor the task execution. The events are stored in a thread-local storage. An event is added to the event list at the beginning and at the end of a task execution.

show the execution of simulation (resp. analytics) tasks and purple areas highlight thread idleness periods. The tool highlights the idle periods of the simulation. Even if ExaStamp shows a good scaling on the 28 Broadwell cores, the fork-join model that it implements induces sequential regions. The analytics parallelized with TBB also presents sequential regions, where the threads are idle. Obviously, we see that the sequential analytics leads to a lot of resource wasting because only one thread is running during the analytics. This is a major drawback of the synchronous approach: when the analytics is not parallel or not enough parallelized, several threads may not be used and resources are lost.

The visualization tools lets us highlight the unused resources of the synchronous in situ. These periods are inherent to the simulation and analytics code parallelization. The use of these idle resources could increase the performance of the in situ system. We therefore show in the next section how to transform the synchronous approach into a task-based asynchronous framework in a first attempt to use the otherwise wasted resources.

4.3 Derivation of a Task-Based Asynchronous IN Situ Approach (TINS)

In the previous section, we have shown that the synchronous in situ approach induces idle periods due to the parallelization of the simulation and the analytics. These idle periods establish the potential to implement a more efficient in situ system. We show in this section how to transform the synchronous execution into a task-based asynchronous execution (Section 4.3.1). We then show that the asynchronous approach leads to a smaller execution time than the synchronous

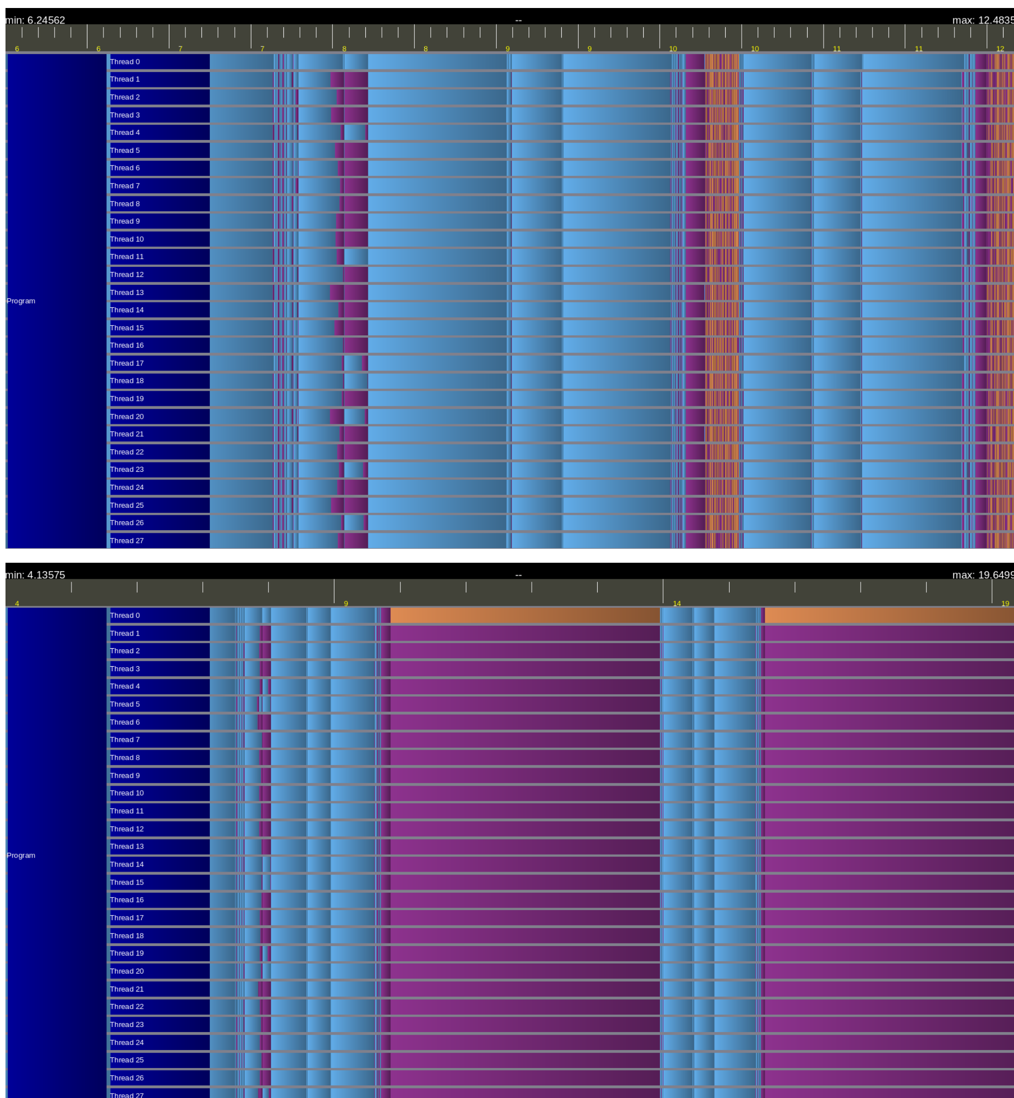

Figure 4.5 | Visualization traces for synchronous in situ execution of two analytics during two iterations of a simulation with 4,000,000 particles: `statistics_par` (top) and `statistics_seq` (bottom). The traces were measured on the 28 cores of one Broadwell node. Blue and orange areas correspond respectively to simulation and analytics tasks execution. Purple areas highlight thread idleness periods. Thread 0 (top) corresponds to the simulation master thread.

approach on a set of analytics (Section 4.3.2). This is the first step toward TINS, our Task-based asynchronous IN Situ approach integrated in the simulation code.

4.3.1 Spawning of an Analytics Task

Analytics Task Creation

In the synchronous approach presented in Figure 4.1 (right), the simulation master thread runs the analytics and waits for its completion before resuming to the next iteration. A first step toward an asynchronous approach is to make the simulation master thread spawn an analytics task and resume to the next iteration without waiting for the task completion (Figure 4.6). Transforming the analytics into a task is made possible with the TBB API as described in Chapter 3 and detailed in Figure 4.7. First, we need to define a class `AnalyticsTask` that derives from the `tbb::task` class. This class has methods and data members and it musts override the `execute` method to define the task execution. Here we call the `runAnalytics` function that can itself create tasks via TBB parallel regions for example. The `execute` method returns when all the tasks in the `runAnalytics` function have completed. The simulation creates the analytics task (line 17), spawns it (line 18) and resumes to the next iteration. This way, analytics and simulation tasks are created concurrently and we let TBB schedule the different tasks on the worker threads. This is the first version of TINS, our Task-based IN Situ approach that we will enhance throughout the manuscript.

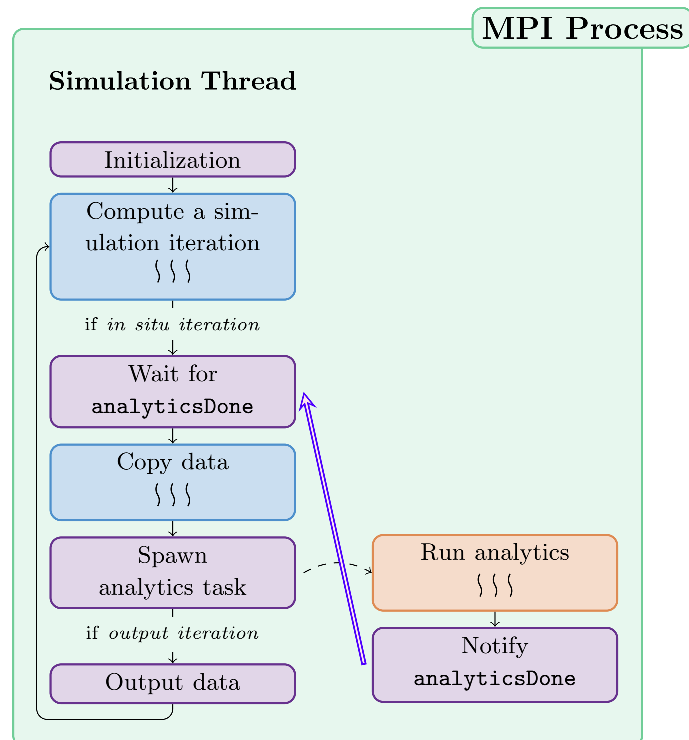

Figure 4.6 | ExaStamp main loop where the simulation spawns an analytics tasks at the end of the iteration and resumes without waiting for its execution. The double arrow highlights the synchronization between the simulation master thread and the analytics task: the simulation cannot copy data into the temporary buffer if the analytics is still executing.

```
// Definition of the analytics task
class AnalyticsTask : public tbb::task
{
  AnalyticsTask(MyData* data) : m_data(data) {};
  tbb::task* execute()
  {
    runAnalytics(m_data); // launch analytics
    this->wait_for_all(); // wait for completion
    notify(analyticsDone);
    return 0;
  }

  MyData* m_data;
};

// Creation and spawn of the analytics task
AnalyticsTask* t = new( tbb::task::allocate_root() ) AnalyticsTask(data);
tbb::task::spawn(*t);
```

Figure 4.7 | Creation of the analytics task by the simulation.

Data Copy

A question arises here: how can an analytics task work on ExaStamp data while ExaStamp is computing the next iteration on the same data? As we have seen in Chapter 3, ExaStamp does not use a double buffering technique to update the particle attributes at each iteration. In particular, the positions are directly modified when the simulation begins an iteration, which gives a very short time for the analytics to use the simulation data. Fortunately, molecular dynamics codes do not need all the available memory of the processors to run the simulation and we can rely on data copies to enable asynchronous executions of analytics. For example, when running a simulation of 4,000,000 particles per MPI process, ExaStamp data structures require roughly 4GB of memory per node, which is significantly less than the 128GB of RAM available on a Broadwell node. Copying one iteration of the data and working on two sets of particles attributes is thus not critical in term of memory usage.

When the simulation reaches an in situ iteration, it copies the data into a temporary buffer, spawns the analytics task and resumes to the next iteration. The analytics task works on the data copied by the simulation to prevent data race issues. To reduce the size of the copied data, we introduce the `ParticleInSitu` structure of arrays defined in Figure 4.8. We only copy the indices, types, positions and velocities of the particles, because they are the most used parameters for molecular dynamics data analytics [110, 97]. We will explain in Chapter 7 how the TINS framework is designed to allow more attributes to be shared from the simulation to the analytics. Making a copy of the data can also have another advantage: the analytics do not need to be aware of ExaStamp complex data structure anymore. This will be the first step for externalizing the analytics from ExaStamp code, as we will see in Chapter 7.

To have only one copy of the data at every time, TINS adds a synchronization between the simulation and the analytics task. The simulation can indeed not overwrite the data in the temporary buffer until the task has completed its work. When dealing with 4,000,000 particles per MPI process, copying the particles attributes in the `ParticleInSitu` structure requires 224MB of data, which is significantly less than the 128GB of memory of a Broadwell processor and it

```
1   struct ParticleInSitu
2   {
3     int nbPart;              // number of particles
4     int* id;                 // array with the particles indices
5     int* type;               // array with the particles types
6     double *rx, *ry, *rz;    // arrays with the particles positions along x, y, z
7     double *vx, *vy, *vz;    // arrays with the particles velocities along x, y, z
8   };
```

Figure 4.8 | `ParticleInSitu` data structure. The simulation copies data in this structure when reaching an in situ iteration and the analytics task uses this copy of the data for its computations. The structure attributes correspond to the most used parameters for molecular dynamics data analytics.

could be possible to store several copies of the data in order for the simulation not to wait for the analytics task completion. However, we are interested in this work in the end-to-end execution time from the beginning of the simulation to the end of the analytics. While having multiple copies of the data prevents the simulation to wait for the analytics completion, it does not have a great impact on the total execution time because all the tasks will be executed after the end of the simulation. Moreover, it would require a mechanism to free data when they are not used anymore, given more work to the simulation. We therefore decided to keep only one copy at a time, even if this means for the simulation to wait for the analytics task completion.

MPI Communications

The analytics tasks can be executed by any of the worker threads. In the case where the analytics need to perform MPI communications, these later may occur concurrently with MPI communications performed by the simulation master thread. Two aspects have to be taken into account here. First, we need to use a MPI thread level support that enables several threads to perform MPI communications simultaneously, that is to say `MPI_THREAD_MULTIPLE`. We make here the hypothesis that the approach is used with a version of MPI that supports `MPI_THREAD_MULTIPLE`. In the case where this threading support is unavailable, TINS framework can be adapted but requires extra developments, as it will be explained in Chapter 8. Secondly, simulation and analytics must use distinct MPI communicators in order for global communications not to be mixed.

4.3.2 Evaluation of TINS compared to a Synchronous Execution

Table 4.2 shows the total execution times of the synchronous and asynchronous approaches on an ExaStamp simulation on 64 Broadwell nodes for the four analytics. The execution times are expressed as percentages of the execution time of ExaStamp alone without in situ analytics and without file output. The analytics are executed after each iteration. Analytics are usually not performed as frequently because the physics of the system does not evolve that fast but we decided to stress the system to make the overheads more visible.

We observe that TINS is faster than the synchronous approach, the performance boost depending on the proportion of sequential regions in the analytics. The gain of TINS is low on parallelized analytics (5% faster for `statistics_par`, 3% faster on `radial`) because they only have a few sequential regions to harvest. The difference is more important in the `histogram` analytics (14% faster) because the MPI collective communications of this analytics are blocking.

Table 4.2 | Comparison of the synchronous approach and TINS on a simulation of 32 iterations with 256,000,000 particles on 64 Broadwell nodes. For each test, analytics are executed after each iteration. The total execution times are expressed in percentages with respect to the execution time of ExaStamp alone without in situ analytics and without file output.

	`statistics_seq`	`statistics_par`	`histogram`	`radial`
synchronous	317.43 %	110.30 %	120.93 %	154.59 %
asynchronous	193.78 %	104.84 %	103.47 %	150.17 %

They therefore induce large sequential regions where the worker threads can switch to simulation execution in the asynchronous case instead of being idle in the synchronous one. The effect is all the more visible on the `statistics_seq` analytics where TINS is 39% faster than the synchronous execution. In this sequential analytics, all the worker threads were idle in the synchronous case while they can execute simulation tasks in TINS.

The overhead with respect to the simulation alone depends on the analytics size. `statistics_seq` and `radial` analytics can be considered as heavy analytics because their asynchronous execution induces an overhead on ExaStamp of 94% and 50% respectively while `statistics_par` and `histogram` are more lightweight analytics with overheads of 5% and 3% respectively. The analytics are performed after each iteration to stress the system and even with this, TINS has overheads of less than 5% for lightweight analytics, showing good potentials to use TINS for high frequency data analytics.

We have seen in this section how to transform a synchronous execution into a task-based asynchronous execution thanks to the TBB task API. We have shown that TINS can be up to 39% faster than the synchronous approach on various analytics. We have also shown that TINS has an overhead of less than 5% when executing lightweight parallel analytics after each iteration, proving that TINS can be used for high frequency data analytics.

4.4 Chapter Summary

In this chapter, we have developed analytics inside ExaStamp and we have compared their in situ execution times either synchronously or with TINS, our task-based asynchronous approach where the simulation spawns an analytics task. We have shown that TINS can be up to 39% faster than the synchronous approach. It allows the high frequency execution of lightweight analytics within an overhead of 5% over ExaStamp but TINS performance is sensitive to the size and parallelization of the analytics. In the following chapter, we will compare TINS with state-of-the-art in situ middleware, Goldrush [119] and Damaris [42], and show how we can improve the performance of TINS with thread isolation.

5 Implementation of a Thread Isolation to Improve TINS Performance

In the first version of TINS, the simulation master thread spawns an analytics task at a given in situ frequency and resumes to the next iteration without waiting for the task completion. This approach enables to take benefit from some of the wasted cycles of the simulation to execute analytics with a reduce end-to-end execution time compared to the synchronous approach and with a low overhead on the simulation execution time for some analytics. In this chapter, we compare TINS with state-of-the-art middleware to validate the task-based approach compared to process-based approaches. We show that TINS outperforms Goldrush [119] on analytics parallelized with TBB (Section 5.1) and Damaris [42] on lightweight analytics but that Damaris has better performance on heavy analytics (Section 5.2). We finally implement a thread isolation mechanism thanks to TBB arenas and we show that TINS with this isolation competes with Damaris on heavy analytics (Section 5.3).

5.1 Evaluation of TINS compared to the Goldrush Process-Based Approach

In our approach, simulation and analytics run inside the same process and we let the TBB scheduler interleave simulation and analytics tasks on the same resources. As already seen in Chapter 2, in situ processing is often seen as a co-scheduling problem [57, 87]. Simulation and analytics run on two distinct processes on the same cores by over-subscribing the compute nodes resources or by dedicating resources to the analytics. In the Goldrush approach, simulation and analytics processes are executed in a way that minimizes the core idleness periods without degrading simulation performance. TINS and Goldrush share a similar problematic and solve it at a different level (OS level for Goldrush and task level for TINS). We compare in this section the two solutions. We first present how Goldrush is used on the Cobalt supercomputer (Section 5.1.1). We then instrument ExaStamp and develop analytics with the Goldrush API (Section 5.1.2). Finally, we compare TINS and Goldrush on three analytics (Section 5.1.3).

5.1.1 Usage of Goldrush on the Cobalt Supercomputer

Goldrush [119] is a C library that can be schematized by Figure 5.1. Simulation and analytics run on two different processes and they share monitoring data through a shared memory segment managed by System V. The principle of the middleware is the following. Simulation and

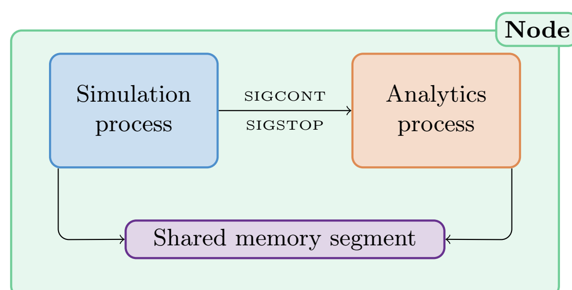

Figure 5.1 | Schematic view of in situ analytics with Goldrush. Simulation and analytics run on their own processes inside the same compute node and share data via a shared memory segment. The cores of the node are over-subscribed: both processes can use all the cores of the node. The simulation process can resume and stop the analytics process by sending SIGCONT and SIGSTOP signals.

analytics processes are launched simultaneously on the node but the analytics process is initially suspended. When the simulation reaches a sequential region, it predicts if the region will be longer than a given threshold. If this is the case, it sends a `SIGCONT` signal to the analytics process to resume. At the end of the sequential region, the simulation sends a `SIGSTOP` signal to the analytics process to suspend it.

Goldrush proposes two scheduling policies: a *greedy* policy where the analytics process runs during simulation sequential regions and a more complex *interference aware* policy where Goldrush also monitors performance counters and determines if the analytics causes interference in the simulation execution. In this situation, PAPI hardware counters [91] are stored in the shared memory segment and Goldrush slows down the analytics execution rate if it causes too much interference to the simulation. Unfortunately, we were not able to access PAPI hardware counters on the Cobalt supercomputer and we decided to use only the greedy policy for our comparisons.

We launch the two processes as a MPMD MPI program on one Broadwell node. Each process can use all the cores of the node and the two codes share the same `MPI_COMM_WORLD` communicator. At initialization, the simulation code creates the shared memory segment and sends contact information to the analytics process so that it can attach it. The `MPI_COMM_WORLD` communicator is split so that the codes can work with distinct communicators.

The transfer of data from the simulation to the analytics is not natively supported by Goldrush. It is made possible by the FlexIO [117] transport in the ADIOS [77] system. Unfortunately, Goldrush is not an open source work and the version of the middleware provided by its authors did not include the necessary functions to use FlexIO. We therefore developed a simple data transfer thanks to the shared memory segment.

5.1.2 Instrumentation of ExaStamp with Goldrush API

To integrate the Goldrush library into ExaStamp, we directly insert the Goldrush API inside the simulation code (see Figure 5.2). The middleware has been developed for MPI+OpenMP applications but it is not bound to any multithreaded programming language because it just needs to know when the simulation enters and leaves sequential regions. When `gr_phase_start` is called, the library identifies the sequential region based on the name of the file and the line number where the function is called. If this sequential region is met for the first time, Goldrush stores the duration of the region as the elapsed time between `gr_phase_start` and `gr_phase_end`. If

the region has already been encountered, the library predicts the duration of the sequential region based on the measurements performed during previous iterations. If the sequential region is predicted to be long enough (more than 1ms), the analytics is resumed. When the simulation calls the `gr_phase_end` function, the analytics process is stopped again. If the sequential region is not predicted to be long enough, Goldrush only monitors the elapsed time between `gr_phase_start` and `gr_phase_end`. The instrumentation of ExaStamp is straightforward except that sequential regions cannot be identified by the file name and the line number. Indeed, some sequential regions are defined once in ExaStamp code but called several times with different inputs. We therefore use a global identifier to differentiate the sequential regions.

```
1  // Initialize Goldrush
2  gr_init(MPI_COMM_WORLD);
3
4  // Begin timeloop
5  gr_mainloop_start();
6  for (int i=0; i<imax; ++i)
7  {
8
9    parallel_region( ... );
10
11   // Sequential region
12   gr_phase_start(__FILE__, __LINE__);
13   sequential_region( ... );
14   gr_phase_end(__FILE__, __LINE__);
15
16   parallel_region( ... );
17
18   // Sequential region
19   gr_phase_start(__FILE__, __LINE__);
20   sequential_region( ... );
21   gr_phase_end(__FILE__, __LINE__);
22
23 }
24 // End timeloop
25 gr_mainloop_end();
26
27 // Finalize Goldrush
28 gr_finalize();
```

Figure 5.2 | Instrumentation of ExaStamp with the Goldrush API.

To compare TINS with the Goldrush approach, we keep the same buffering technique explained in Section 4.3.1. When the simulation reaches an in situ iteration, it copies the data into the shared memory segment in the `ParticleInSitu` data structure and resumes to the next iteration. The data of the shared memory segment can be overwritten only when Goldrush has completed the execution of the analytics for this iteration. It means that parts of the analytics are executed during ExaStamp sequential regions and the remaining are executed synchronously with the simulation if the sequential regions are not long enough between two in situ iterations.

The `statistics_seq` and `statistics_par` analytics are adapted by extracting them from ExaStamp. In the analytics side, we only need to call the `gr_init` and `gr_finalize` functions. The two analytics are also adapted to read data from the shared memory segment.

5.1.3 Comparison of TINS and Goldrush

Table 5.1 shows the total execution time of TINS and Goldrush approaches on an ExaStamp simulation on one Broadwell node. We run three analytics: the parallel statistics performed 100 and 1,000 times after each iteration (`stat_par_100` and `stat_par_1000`) for small and medium analytics parallelized with TBB and the sequential statistics computed 1,000 times at each in situ iteration (`stat_seq_1000`) for a long analytics without TBB.

Table 5.1 | Comparison of Goldrush and TINS on a simulation of 32 iterations with 4,000,000 particles on 1 Broadwell node. For each test, the analytics are executed after each iteration. The total execution times are expressed in percentages of the total execution of ExaStamp alone without in situ analytics and without file output.

	`stat_par_100`	`stat_par_1000`	`stat_seq_1000`
Goldrush	108.25 %	122.56 %	173.84 %
TINS	100.05 %	107.30 %	172.79 %

Goldrush and TINS present similar execution times for the long sequential analytics, with a relative end-to-end execution time of 173%. When executed synchronously, the `stat_seq_1000` relative execution time alone on a Broadwell core was around 170%. For the two approaches, the total execution time is dominated by the analytics execution time and both approaches show a good overlapping of analytics and simulation. In particular, parts of the analytics run during simulation sequential regions, while the remaining are executed at the end of the simulation iteration.

The results are different for the parallel analytics. For both parallel analytics, the overhead of TINS is under 7%, being even negligible for `stat_par_100`. On the other hand, Goldrush has an overhead of 8% (resp. 22%) when computing the statistics 100 (resp. 1,000) times after each iteration. Goldrush can execute several analytics processes concurrently with a multithreaded simulation but it has not been tested with multithreaded analytics [119]. It seems here that there are side effects when running multithreaded analytics. We identified several possible causes to these side effects. A longer context switching for the worker thread or a delay in the moment where the worker threads get the SIGCONT and SIGSTOP signals may lead to a smaller portion of analytics executed during simulation sequential regions. Moreover, the TBB scheduler of the analytics sees N cores in the system and therefore creates $N - 1$ worker threads. When the analytics process is resumed, the analytics master thread and $N - 1$ threads execute analytics task and the simulation master thread computes sequential regions. There are therefore $N + 1$ threads on N cores, leading to core over-subscription and potential performance loss.

For the parallel analytics, TINS manages to interleave simulation and analytics tasks more efficiently than Goldrush. In particular, there is a unique TBB scheduler with $N - 1$ worker threads. There is therefore no need for context switching and there is no core over-subscription. Moreover, both simulation and analytics sequential regions can be exploited, while Goldrush only exploits simulation sequential regions. TINS also exploits the periods when ExaStamp is not efficient enough on the 28 Broadwell cores by adding more tasks to execute. Finally, TINS is easier to use because there is no parameter to use and tune, while the user needs to choose the threshold for the sequential region duration and the size of the shared memory segment in Goldrush.

We have seen in this section that TINS can be up to 12% faster than the Goldrush approach on multithreaded analytics and we suspect that a limitation of the process-based approach proposed by Goldrush may come from the over-subscription of the cores by multithreaded processes that do not share the same TBB scheduler. Another traditional technique of in situ is the *static helper core* approach, which is for example implemented in the Damaris [42] middleware. The cores are in this case split into two groups and each process executes on its own group of cores. We will show in the next section that the process-based static helper core approach of Damaris shows competitive performance that our TINS approach.

5.2 Evaluation of TINS compared to the Damaris Static Helper Core Approach

Another approach to co-schedule simulation and analytics processes on the same resources is to dedicate a set of cores for the analytics process. In this situation, cores are not over-subscribed and the OS scheduler does not have to handle priorities between the two processes. This approach is implemented in the Damaris middleware for example. We compare in this section TINS where we let TBB schedule simulation and analytics tasks on all the cores of a node and Damaris where the cores are split into two groups. We first instrument ExaStamp and develop analytics plugins with the Damaris API (Section 5.2.1) and we compare TINS and Damaris on our set of analytics benchmarks (Section 5.2.2).

5.2.1 Instrumentation of ExaStamp with Damaris API

Damaris is a MPI-based C++ library where simulation and analytics belong to two distinct sets of MPI processes. Each compute node holds a simulation and an analytics MPI processes. Inside a node, data are transferred from the simulation process to the analytics process thanks to a shared memory segment managed by Boost. Damaris architecture is summarized in Figure 5.3.

The instrumentation of ExaStamp is done in two steps. First, we need to initialize Damaris after MPI initialization (see Figure 5.4 left). Damaris splits the `MPI_COMM_WORLD` communicator in two, one for the simulation and one for Damaris server where analytics are executed. ExaStamp thus needs to call the `damaris_client_comm_get` function to get the simulation MPI communicator where it will work. Damaris initialization requires an XML file that gives information for the execution of Damaris, for example the size of the shared memory buffer. This file also describes the data structure used between the simulation and the analytics (Figure 5.5). The XML file also lists the different plugins that can be launched by Damaris upon the reception of events.

The second step of ExaStamp instrumentation is done during the simulation iteration loop. Damaris proposes different ways for the simulation to expose data to the analytics. A *double buffering* technique allows the simulation to allocate its data structures directly inside the shared memory segment. The simulation works inside the shared memory segment and tells when data are ready to be processed and when it does not use the data anymore. This way, no copy is performed when the iteration is over and simulation and analytics can work simultaneously on the same data. This technique proved efficient for simulations that use a double buffering technique [42]. However, as explained in Chapter 3, ExaStamp does not use this double buffering technique. Moreover, due to ExaStamp complex data structure, modifying the code such that it

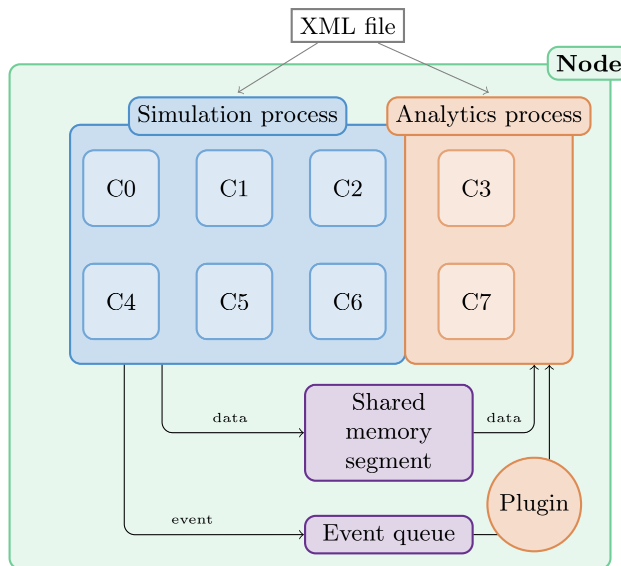

Figure 5.3 | Schematic view of in situ analytics with Damaris on a node composed of 8 cores. The cores are split into two groups: one group of 6 cores for the simulation MPI process and one group of 2 cores for the analytics MPI process. An XML file describes the data that are exchanged in the shared memory segment. An event queue enables to run plugins on the analytics process.

```
int err, is_client;

// Initialize Damaris
damaris_initialize("exastamp.xml",
    MPI_COMM_WORLD);
damaris_start(&is_client);

// If simulation process
if ((err == DAMARIS_OK || err ==
    DAMARIS_NO_SERVER) && is_client
    )
{
  runSimulation();

  // Finalize Damaris
  damaris_stop();
  damaris_finalize();

}
else
{
  // Finalize Damaris
  damaris_finalize();
}
```

```
ParticleInSitu part;
int nbPart = getNbParticles();

// Set number of particles in the
    MPI process
damaris_parameter_set("nbPart", &
    nbPart, sizeof(int));

// Allocate the fields of part in
    the shared memory segment
damaris_alloc("timestep/rx", (void
    **)(&particles.rx));

// Fill the buffer with ExaStamp
    data
fillBuffer(&part);

// Tell that data are ready
damaris_commit("timestep/rx");
damaris_clear("timestep/rx");

// Send the event
damaris_signal("runAnalytics");
```

Figure 5.4 | Instrumentation of ExaStamp with Damaris API: at ExaStamp initialization (left) and at the end of a simulation iteration (right).

```xml
<simulation name="xstamp">

  <parameter name="nbPart" type="int" value="1"/>

  <layout name="intData" type="int" dimensions="nbPart" />
  <layout name="doubleData" type="double" dimensions="nbPart" />

  <group name="timestep">
    <variable name="id" layout="intData" />
    <variable name="type" layout="intData" />
    <variable name="rx" layout="doubleData" />
    <variable name="ry" layout="doubleData" />
    <variable name="rz" layout="doubleData" />
    <variable name="vx" layout="doubleData" />
    <variable name="vy" layout="doubleData" />
    <variable name="vz" layout="doubleData" />
  </group>

  <actions>
    <event name="runAnalytics" action="myAnalytics" library="libanalytics.so" />
  </actions>

</simulation>
```

Figure 5.5 | XML file to describe the `ParticleInSitu` structure shared by simulation and analytics.

allocates its data into the shared memory segment is difficult and we decided to use the second option (Figure 5.4 right). When the simulation reaches an in situ iteration, it allocates resources into the shared memory segment (`damaris_alloc`), copies the data into the allocated buffer, tells Damaris that data are ready to be processed (`damaris_commit`) and that it does not use them anymore (`damaris_clear`) and sends the event to launch analytics (`damaris_signal`).

The development of the plugins is less straightforward than the instrumentation of the simulation code because Damaris does not expose a simple API for the plugins. It is therefore required to go more in depth into Damaris structure as shown in Figure 5.6. A `VariableManager` is in charge of keeping track of the variables defined in the XML file. Data are stored in blocks. On a node, each simulation MPI process copies a block of data per iteration into the shared memory segment. The Damaris servers on a node, and hence the analytics MPI processes of the node, can access all the blocks stored in the shared memory segment of the node. In our situation, each node is composed of one simulation MPI process and one analytics MPI process, as highlighted in Figure 5.3. Thus, the analytics process only needs to retrieve one block of data per iteration. The analytics then performs its computations on the retrieved pointers. Finally, the plugin is in charge of freeing the shared memory segment when data are not necessary anymore.

Damaris does not provide a native way to handle the number of threads that the different processes can access. The user is therefore in charge of splitting the cores into two groups and to allocate a group of cores to each MPI process of the node. We use here the `taskset` Linux command to separate at launch time the N cores of a node into two groups of cores and assign them to each MPI process of the node. When asking n cores for the analytics on a processor with N cores, the n first cores are assigned to the analytics process and the $N - n$ last cores to the simulation, guaranteeing that simulation and analytics processes use different sets of cores.

Damaris is integrated inside the simulation executable. The only difference with ExaStamp alone is that the user asks two MPI processes per compute node. The size of the shared memory

```
void myAnalytics(const char* name, int source, int iteration, const char* args)
{
  // Look for rx variable
  shared_ptr<Variable> v;
  v = VariableManager::Search("timestep/rx");
  if (v)
  {
    // Get block corresponding to this iteration
    shared_ptr<Block> b = v->GetBlock(source, iteration, 0);
    if (b)
    {
      // Get pointer to the data
      void* addr = b->GetDataSpace().GetData();
      double *rx = (double*) addr;
      runAnalytics(rx);
    }
    else
      std::cerr << "Block not found" << std::endl;
  }
  else
    std::cerr << "Variable timestep/rx not found" << std::endl;

  // Free shared memory segment
  VariableManager::iterator var = VariableManager::Begin();
  VariableManager::iterator end = VariableManager::End();
  while (var != end)
  {
    var->get()->ClearAll();
    var++;
  }
}
```

Figure 5.6 | Development of a Damaris plugin. Damaris does not provide an API for plugin development and we have to go more in depth into Damaris structure.

segment has to be set before the execution and we choose it so that only one copy of the data can fit inside.

5.2.2 Comparison of TINS and Damaris

Figure 5.7 compares TINS and Damaris for the four analytics. Damaris uses different numbers of static helper cores (1 to 14 cores out of the 28 Broadwell cores). The results for the four analytics can be divided in three patterns: sequential analytics, lightweight parallel analytics and compute intensive parallel analytics.

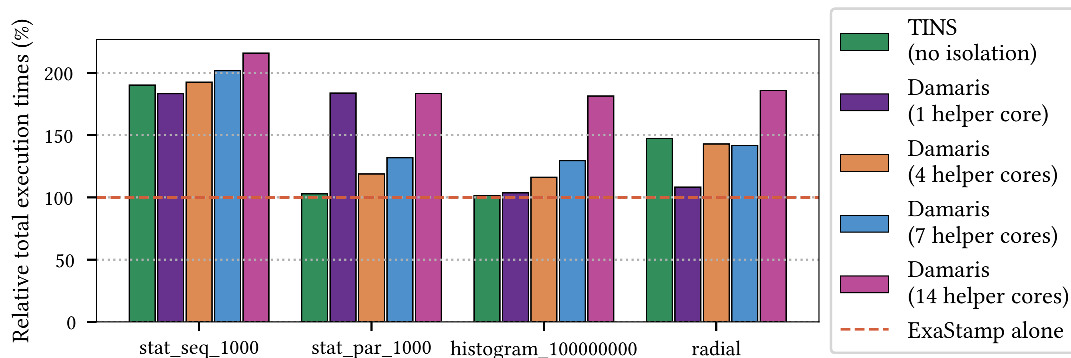

Figure 5.7 | Comparison of the relative total execution times of the in situ execution of 4 analytics with TINS and Damaris. Damaris uses different numbers of static helper cores. The total execution times are expressed in percentages of the total execution time of ExaStamp alone. ExaStamp simulation of 32 iterations with in situ analytics after each iteration on a simulation of 256,000,000 particles on 64 Broadwell nodes.

Sequential Analytics

In a static helper core strategy, allocating more than one core for the `statistics_seq` analytics is useless: it just reduces the simulation execution time because the simulation runs on fewer resources while not speeding up the sequential analytics. The analytics being longer than the simulation iteration, the execution time is dominated by the analytics for both approaches. In TINS, there will also be one thread executing the sequential task while the other threads execute simulation tasks. The difference between the Damaris approach with one helper core is that the helper thread of Damaris will always execute on the core 0 of the processor because of the splitting of the cores into two groups made by the `taskset` function. In TINS, there is no guarantee that the task will be executed on a dedicated core. Broadwell nodes being decomposed in 4 NUMA nodes, this can lead to NUMA effects when the threads that execute the task are not on the same NUMA node from one iteration to the other. This is the reason why Damaris with one helper core is slightly better than the TINS approach for this analytics.

Lightweight Parallel Analytics

`statistics_par` and `histogram` are both analytics parallelized with TBB where small analytics tasks are created. TINS manages to interleave simulation and analytics tasks with an overhead of less than 3% compared to ExaStamp alone.

The choice of the number of static helper cores for Damaris has an influence on the total execution time. When using only one helper core, the analytics runs sequentially, hence showing the same execution time than a sequential computation. For the `statistics_par` analytics, increasing the number of helper cores is necessary to benefit from the analytics parallelization. 4 helper cores seems to be the best trade off for the `statistics_par` analytics. When using more helper cores, too much cores are removed from the simulation process and the total execution time gets dominated by the simulation execution time. On the other hand, the `histogram` analytics is lightweight compared to the simulation iteration, even sequentially. One helper core is therefore the optimal for this analytics. However, even with the optimal numbers of helper cores for the two parallel analytics, Damaris does not manage to be faster than the TINS approach, showing the benefit of the task-based approach for these analytics.

Compute Intensive Parallel Analytics

The result is different for the `radial` analytics. Using one helper core is the best choice for Damaris that shows an overhead of 4% compared to ExaStamp alone while TINS shows an overhead of 47% for this analytics. The `radial` analytics is a computationally intensive analytics with an important number of large tasks. TINS does not manage to interleave efficiently the `radial` tasks because ExaStamp does not present enough sequential regions to be exploited for this heavy analytics. Moreover, if tasks of the `radial` analytics are executed during ExaStamp sequential regions, they can prevent simulation tasks to be executed. Indeed, the threads need to complete the tasks of the `radial` analytics before executing simulation tasks. The tasks can therefore disturb the simulation that may be delayed after the end of the sequential region. By dedicating one core in the Damaris approach, the tasks created by the `radial` analytics do not disturb the simulation execution, hence the difference in the overhead with respect to ExaStamp alone.

We have compared in this section TINS with the Damaris approach where the cores are split into two disjoint groups, one for the simulation and one for the analytics. We have shown that lightweight analytics parallelized with TBB are easily interleaved between simulation tasks by TINS but that having one dedicated helper core can both reduce the NUMA effects in the case of a sequential analytics and avoid perturbations with the simulation execution in the case of a compute intensive analytics. In the following section, we will look at the influence of a thread isolation in TINS and see if it can reduce the overheads for sequential and compute intensive analytics without increasing the overheads for lightweight analytics parallelized with TBB.

5.3 Implementation of a Thread Isolation Mechanism in TINS

We have shown in the previous section that dedicating resources to the analytics can be beneficial for sequential or heavy parallel analytics. We investigate in this section the influence of thread isolation in TINS. To do so, we separate the task execution into two groups of threads (Section 5.3.1) that can be pinned on disjoint cores thanks to the TBB arena and observer features introduced in Chapter 3. We then propose an alternative solution to the analytics task spawned by the simulation by creating an analytics master thread with a different timeloop than the simulation master thread (Section 5.3.2) and evaluate the two versions of TINS with respect

to Damaris (Section 5.3.3). Finally, we highlight the limitations of the static helper core approach (Section 5.3.4).

5.3.1 Separation of the Tasks into Disjoint Arenas

As already seen in Chapter 3, TBB proposes different mechanisms to guide the task execution. In particular, arenas allow to execute tasks in different groups of threads given a *concurrency level* that sets the maximum number of threads that can execute tasks concurrently. The observer intercepts when a thread enters an arena to pin it to a given subset of cores.

Let N be the number of cores in the processor. We use two arenas: one with a concurrency level of n_s for the simulation and one with a concurrency level n_a for the analytics so that $\boldsymbol{n_s + n_a = N}$ to guarantee the thread isolation. The arenas are instantiated during ExaStamp initialization, the analytics arena size being chosen by the user in the input data file. ExaStamp code is modified so that each parallel region is called inside the proper arena (Figure 5.8). To each arena is associated a `task_group` to represent the concurrent execution of the tasks. The fork-join model used by ExaStamp is kept thanks to the `execute` and `run_and_wait` functions. The threads enter an arena to execute a parallel region and leave the arena when the tasks of this arena have all been executed.

```
1  simulationArena->execute( [&]
2  {
3    simulationGroup->run_and_wait ( [&]
4    {
5      tbb::parallel_for( tbb::blocked_range<J>(begin, end),
6      [&](const tbb::blocked_range<J>& r)
7      {
8        lambda(r.begin(), r.end());
9      }
10     );
11   });
12 });
```

Figure 5.8 | Encapsulation of a parallel region into the simulation arena. The task group represents the concurrent execution of the tasks. The fork-join model of ExaStamp is kept thanks to the `run_and_wait` and `execute` functions.

We derive the `arena_observer` class from the TBB `task_scheduler_observer` class and we associate one arena observer to each arena (Figure 5.9). Each arena observer holds a mask, corresponding to n_a cores for the analytics observer and n_s cores for the simulation observer. When a thread enters an arena, it calls the `on_sheduler_entry` method of the corresponding arena observer and the thread mask is applied to the thread thanks to the `pthread_setaffinity_np` function. The threads are not pinned on a unique core each but they execute on a subset of cores. We will see in next section the influence of arena pinning on the total execution time.

5.3.2 Implementation of an Analytics Master Thread

We have seen in Section 5.2.2 that NUMA effects can be reduced when sequential analytics are executed on the same core from one iteration to another. We therefore propose an alternative

```
class arena_observer : public tbb::task_scheduler_observer
{
public:

  arena_observer( tbb::task_arena& _a, cpu_set_t _target_mask) :
    tbb::task_scheduler_observer(_a), target_mask(_target_mask)
  {
    observe(true);
  }

  void on_scheduler_entry( bool worker )
  {
    pthread_t current_thread = pthread_self();
    pthread_setaffinity_np(current_thread, sizeof(cpu_set_t), &target_mask);
  }

  void on_scheduler_exit( bool worker ) { }

  ~arena_observer() {};

private:

  cpu_set_t target_mask;
};
```

Figure 5.9 | Description of the arena observer class. The `on_scheduler_entry` method is called when a thread enters the arena and the `on_scheduler_exit` method is called when the thread leaves the arena.

method to the analytics task created by the simulation master thread that guarantees that sequential analytics will be executed by the same thread. We also study the impact of the binding of the temporary buffer and the arenas on the total execution time.

Analytics Master Thread

Figure 5.10 describes the version of TINS where an analytics master thread is spawned by the simulation during the initialization. The two master threads have different timeloops, the simulation master thread being in charge of simulation tasks creation and the analytics master thread creating analytics tasks.

For the simulation master thread, the difference with the timeloop presented in Figure 4.6 relies on the fact that the simulation master thread does not spawn an analytics task after having copied the data but sends a notification instead. When reaching an *analytics breakpoint*, the simulation master thread copies the data into the temporary buffer and notifies the analytics master thread that data are ready to be processed with the `dataReady` signal. It then resumes to the next iteration. On the other side, the analytics master thread waits for data to be ready. When it receives the `dataReady` signal, it launches the analytics execution inside the analytics arena. Once the analytics tasks have been executed, the analytics master thread notifies the simulation master thread with the `analyticsDone` signal, telling the simulation that data in the temporary buffer can be overwritten. The simulation master thread therefore waits for the `analyticsDone` signal before copying the data into the temporary buffer. This synchronization is necessary to keep only one copy of the particles data at a time. It is disabled for the first analytics breakpoint to avoid a deadlock. In the case where enough memory is available to store several copies, a

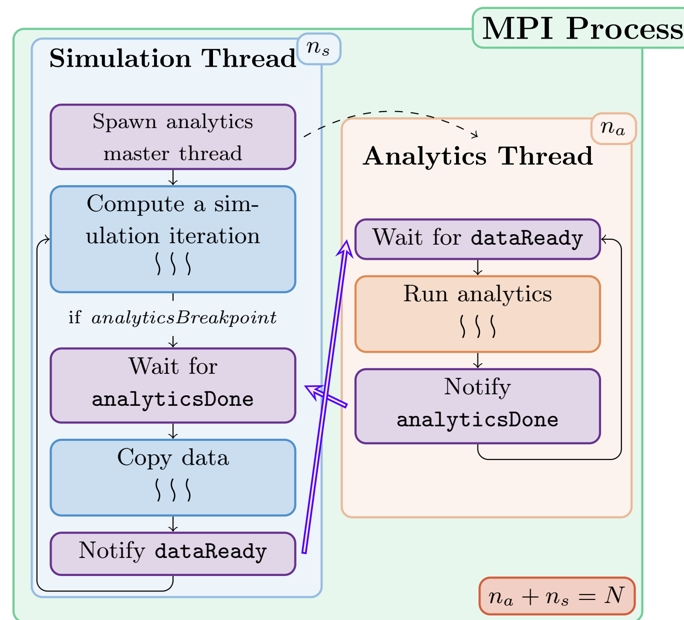

Figure 5.10 | Timeloops of the simulation (left) and analytics (right) master thread. The blue rectangles correspond to portions of codes that can be executed in the simulation arena of size n_s. The orange rectangle corresponds to the execution of analytics inside the analytics arena of size n_a. In the static helper core configuration, $n_a + n_s = N$ where N is the number of cores in the node. The file output is omitted compared to Figure 4.6 because the file writing can be seen as an analytics executed by the analytics master thread.

buffering system could be implemented to reduce the time lost by the simulation master thread, waiting for the analytics to be finished.

Influence of the Arena Binding

We compare here the impact of three different binding strategies:

- `nobinding`: the arenas are not bound and we let TBB schedule the threads on the different cores. The arena observers are disabled;
- `master-binding`: the analytics master thread is bound on core 0 and the simulation master thread on core $N-1$. The binding of the master thread is made once at ExaStamp initialization and the arena observers are disabled;
- `arena-binding`: the analytics master threads are bound and the observers apply a mask when the worker threads enter the arenas. They bind the analytics arena on the first n_a cores (corresponding to the first NUMA nodes of the processor) and the simulation arena on the remaining n_s cores (corresponding to the last NUMA nodes of the processor).

As already seen in Chapter 3, ExaStamp uses an affinity partitioner and using two arenas may disturb its efficiency. Indeed, the affinity partitioner tries to assign tasks to the same threads from one iteration to another to optimize cache affinity. However, in a two-arena system, the threads are not assigned to a particular arena and a thread may be in the simulation arena for some iterations and then enter the analytics arena. This is highlighted in Figure 5.11 where the thread 6 for example is in the simulation arena in the first iteration but in the analytics arena in

the second iteration. Let suppose that in this example the affinity partitioner thinks that the task t should be executed by the thread 6. Because it is not in the simulation arena for the second iteration, another thread executes the task t, potentially leading to cache issues. This effect could be all the more exacerbated when the processor is composed of different NUMA nodes.

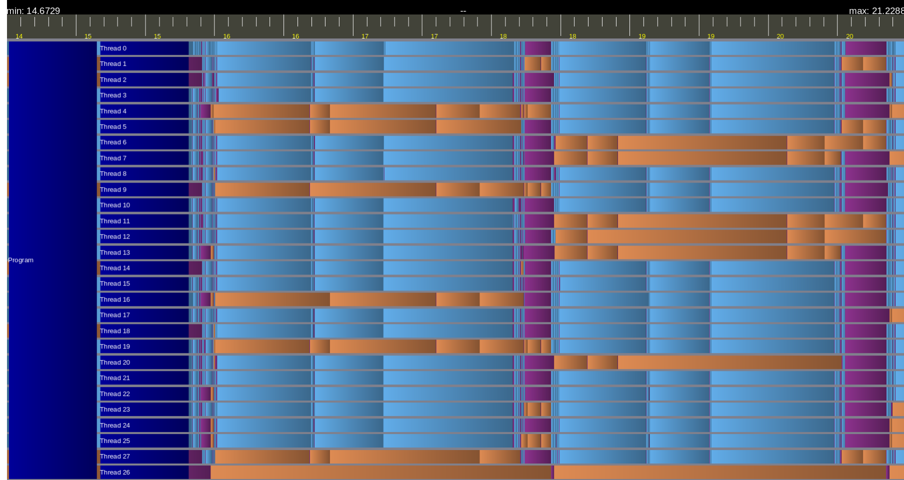

Figure 5.11 | Trace of the in situ execution of `statistics_par` with $n_a = 7$ and $n_s = 21$ on a 28-core Broadwell node for two iterations of ExaStamp. Blue and orange areas correspond respectively to simulation and analytics tasks execution. Purple areas highlight thread idleness periods. Thread 0 (top) is the simulation master thread and thread 27 (bottom) the analytics master thread.

To validate this hypothesis, we measure the total execution time of TINS when computing the parallel statistics with $n_a = n_s = N/2$ and we compare it with the execution time of ExaStamp alone running on $N/2$ cores. We choose the parallel statistics because it is a lightweight analytics so that the total execution time is dominated by the simulation with this arena configuration. The threads are also likely to change of arena from one iteration to the other, highlighting the NUMA effects. We compare the three binding strategies when ExaStamp uses the affinity partitioner but we also look at the impact of the binding when using two other partitioners: simple partitioner and auto partitioner. These partitioners do not optimize the cache affinity and should be less sensitive to the two-arena system.

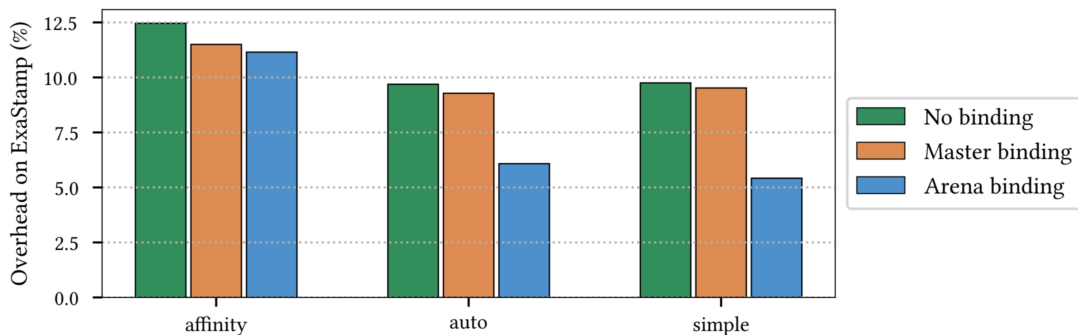

Figure 5.12 | Overhead of three binding strategies (`nobinding`, `master-binding` and `arena-binding`) on ExaStamp alone executed on 14 Broadwell cores for three partitioners (affinity, auto and simple). For the three binding strategies, TINS is used with $n_a = n_s = 14$. ExaStamp simulation of 32 iterations with `statistics_par` being executed after each iteration on a simulation of 4,000,000 particles on 1 Broadwell node.

Figure 5.12 shows the overhead of the different binding strategies on ExaStamp alone executed on 14 Broadwell cores. For the three partitioners, `master-binding` shows smaller overhead than `nobinding` and the overhead is further reduced with the `arena-binding` strategy. The performance gain is more visible for the auto and simple partitioners because the arena binding gives them thread affinity that reduces cache effects. While the overhead of `arena-binding` when using auto and simple partitioners is around 6%, the overhead reaches 12% using an affinity partitioner. This validates our hypothesis that the two-arena system has a negative influence on the affinity partitioner. This latter is not able to decide which thread should execute which task because the threads may change of arena at each iteration.

Influence of the Temporary Buffer Binding

Table 5.2 compares the simulation, copy and analytics execution times of a static helper core strategy for different bindings of the temporary buffer. The comparison is performed on a Broadwell node of 28 cores with arenas of size $n_a = n_s = 14$. The analytics master thread is pinned on the core 0. The analytics arena is pinned on the 14 first cores, corresponding to the NUMA nodes 0 and 1. The simulation arena is pinned on the 14 last cores, corresponding to the NUMA nodes 2 and 3. Compared to a situation where the location of the buffer is not set, the analytics is up to 41% faster when allocating the buffer close to the analytics master thread and up to 22% slower when allocating the buffer close to the simulation master thread. The different binding strategies also show slightly slower execution times for the simulation and the copy.

Table 5.2 | Simulation, copy and analytics execution times of a static helper core strategy with $n_s = n_a = 14$ on a 28-core Broadwell node when binding or not the temporary buffer on a NUMA node. The relative execution times are expressed in percentages with respect to ExaStamp alone executed on 14 cores. ExaStamp simulation of 4,000,000 particles for 32 iterations where `statistics_par` is executed after each iteration.

	Simulation	Copy	Analytics
No binding	113.64 %	0.67 %	0.21 %
NUMA 0	112.55 %	0.41 %	0.13 %
NUMA 1	112.84 %	0.42 %	0.13 %
NUMA 2	112.43 %	0.47 %	0.26 %
NUMA 3	112.30 %	0.47 %	0.34 %

Given all the measurements, we decided to bind the analytics arena on the first NUMA nodes and to allocate the temporary buffer on the NUMA node 0 where the analytics master thread is bound. With these bindings, we expect to reduce the NUMA effects highlighted by some of the analytics.

5.3.3 Comparison of the Two Versions of TINS and Damaris

Figure 5.13 compares TINS without isolation, Damaris and TINS with static helper core strategy (TINS SHC) on the four analytics used in Section 5.2.2. Each bar of the plot corresponds to a strategy with a number of static helper core. To better understand the total executions times of the different strategies, the bars are divided into two parts. The left part corresponds to times measured on the simulation side and the right part corresponds to times measured on the analytics side. For each part, the dashed areas measure the time when the simulation (resp. analytics)

master thread is active and the blank areas measure idle times of the master threads, either when waiting for data to be ready for the analytics master thread or waiting for the analytics completion for the simulation master thread. The total execution times are expressed in percentages of the total execution time of ExaStamp alone.

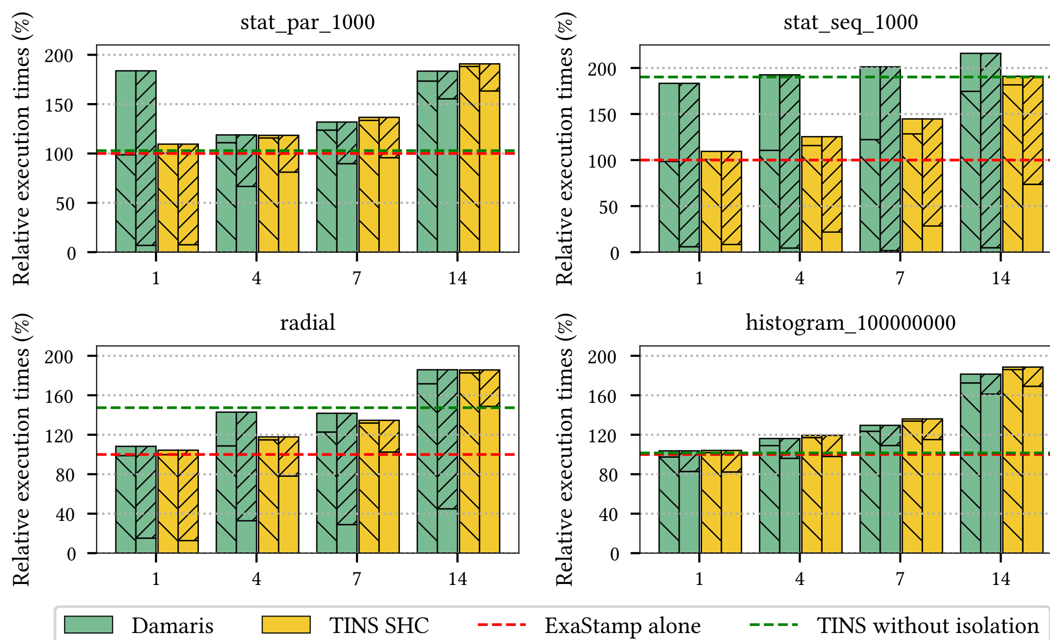

Figure 5.13 | Comparison of the different strategies on 64 Broadwell nodes for a simulation of 32 iterations on 256,000,000 particles. The analytics are performed after each iteration. The total execution times are normalized with respect to ExaStamp alone. The TINS without isolation reference execution times are those discussed in Section 5.2.2 and Chapter 4.

Overall, TINS and Damaris show similar performance, TINS being even better for heavy analytics (sequential statistics and radial). When looking into more details, we notice that TINS with static helper core presents an overhead on the simulation execution time, compared to the Damaris approach. When using one static helper core, the simulation execution time is approximately 2% longer with TINS SHC than with Damaris and this difference reaches 8% when using 14 static helper cores. We identified two reasons that may explain this overhead. First, we explained in the previous section that the two-arena system disturbs the affinity partitioner of ExaStamp. The observers also apply a mask to set the core affinity every time a thread enters an arena in TINS, potentially leading to small overheads. In Damaris, the processes are separated thanks to the `taskset` function and there is only one arena per process. Moreover, Damaris does not use an observer because the cores are split at launch time, leading to fewer interference.

On the other side, while the execution time of `histogram` is equivalent for Damaris and TINS SHC, the execution times of the other analytics are longer with Damaris than with TINS SHC. For example, the execution of `statistics_seq` can be up to 79% longer with Damaris than with TINS SHC with the same number of static helper cores. Measurements with VTune show an important impact of NUMA effects. Indeed, for the same amount of memory accesses, TINS SHC presents 15% of DRAM remote accesses while this proportion reaches 75% for Damaris. This is due to Damaris memory management. Damaris relies on a shared memory segment to exchange data between simulation and analytics processes and this shared memory segment is not bound

to a specific location. The shared memory segment is therefore likely to be interleaved on the different NUMA nodes, leading to major performance penalties when data need to be accessed. In TINS SHC, the temporary buffer is located on the same NUMA node than the analytics master thread, limiting the NUMA accesses when the analytics threads need to access copied data.

For lightweight analytics parallelized with TBB, TINS SHC is slightly slower than TINS without isolation (respectively 3% for `histogram` and 6% for `statistics_par`) because of the overhead of TINS SHC on the simulation execution time and because one thread is removed from the simulation. On the other hand, the isolation leads to better performance for `statistics_seq` and `radial`, except when the number of helper cores is set to 14 because too much threads were removed from the simulation. With one static helper core, TINS SHC is 42% faster than TINS without isolation on `statistics_seq` and 29% faster on `radial`. The isolation and the memory placement strategies therefore prove their benefit on heavy analytics while having a low overhead on lightweight analytics. Moreover, as it will be discussed in Chapter 7, the analytics master thread implemented in TINS SHC has the advantage of separating simulation and analytics codes more efficiently than TINS without isolation.

5.3.4 Highlighting the Limitations of the Static Helper Core Approach

We have seen in the previous section that a thread isolation in TINS can lead to better performance than TINS without isolation on heavy analytics. However, the static helper core approach has still limitations with respect to resource consumption. In particular, it does not exploit the sequential regions as in the TINS without isolation case. Moreover, the static helper core approach adds idleness periods during synchronization points, when the analytics master thread waits for the `dataReady` signal or when the simulation master thread waits for the `analyticsDone` signal.

The time lost during synchronization points depends on the choice of the number of helper cores. Figure 5.14 shows the traces of the execution of `statistics_par` with three different numbers of static helper cores on one Broadwell node. Notice that the trace measurement system has been modified to show when the threads are in the simulation or analytics arenas and does not show the task execution anymore. We did so because the overhead on the simulation execution time was too important when measuring the task execution and the files with task measurements were too heavy to be processed. We therefore instrumented the arena observers to create events when the threads enter or leave the arenas.

When using only one helper core, the analytics is executed on the analytics master thread and the sequential execution is longer than a simulation iteration. The 27 simulation threads are thus idle when the simulation master thread waits for the `analyticsDone` signal. On the other hand, when using 4 static helper cores, the analytics execution is faster than the simulation iteration and the 4 analytics threads are idle when the analytics master thread waits for the `dataReady` signal. The best choice for this particular analytics seems to be 2 static helper cores when the analytics execution time is close to the simulation execution time, hence reducing the thread idleness periods.

We have implemented in this section a static helper core approach in TINS thanks to an analytics master thread spawned by the simulation master thread, an arena system and arena observers. We have shown that TINS with static helper core approach can be up to 79% faster than

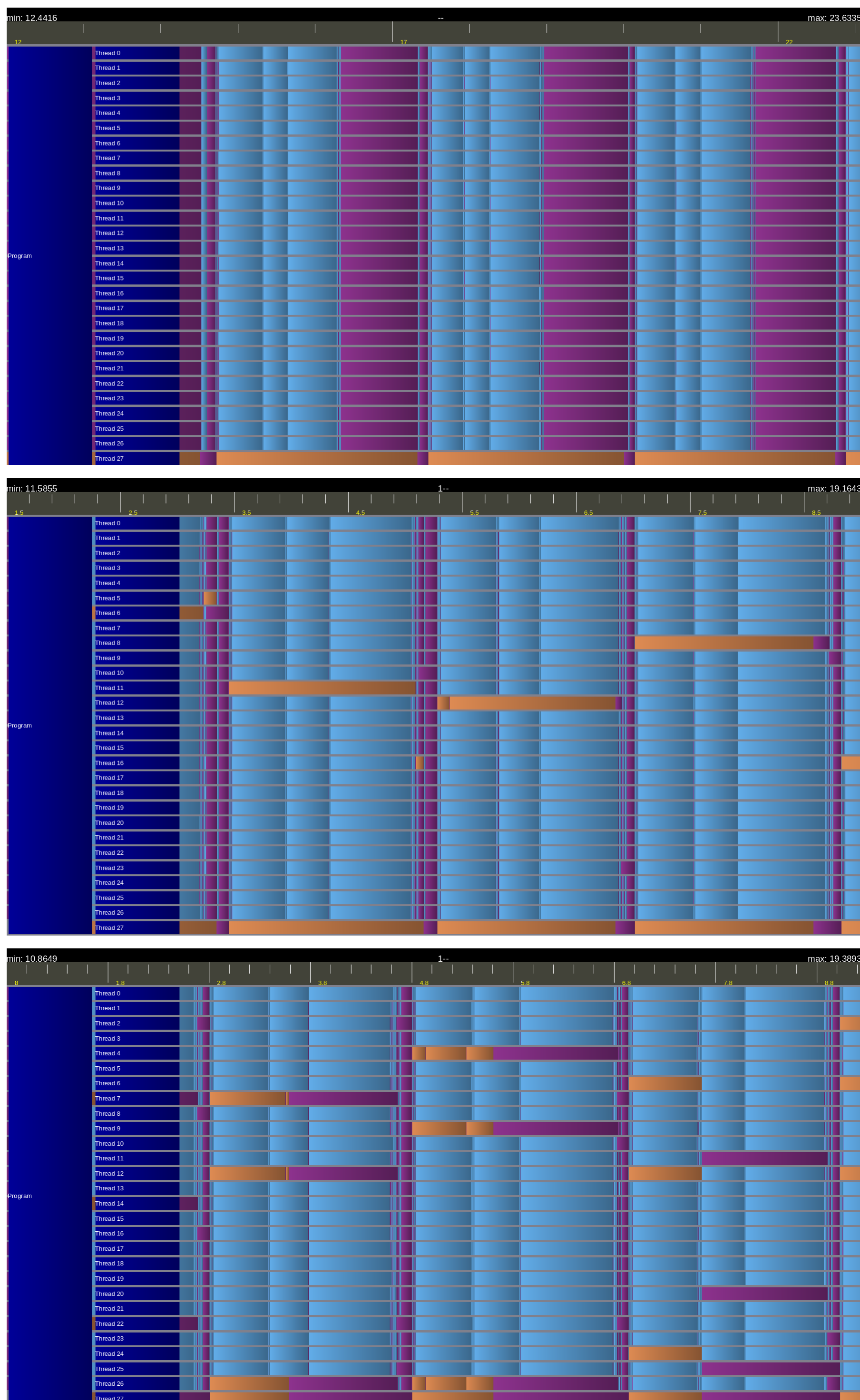

Figure 5.14 | Trace of the in situ execution of `statistics_par` for three numbers of static helper cores: 1 (top), 2 (middle) and 4 (bottom) on a 28-core Broadwell node for three iterations of ExaStamp. The trace shows when the threads are in the simulation arena (blue areas) or in the analytics arena (orange areas). Purple areas highlight thread idleness periods. Thread 0 (top) is the simulation master thread and thread 27 (bottom) the analytics master thread.

Damaris for memory intensive analytics and up to 42% faster than TINS without isolation. The overhead of TINS with static helper core on the simulation execution time is also less sensitive to the analytics than TINS without isolation, given that the optimal number of static helper cores is well chosen at runtime.

5.4 Chapter Summary

We have evaluated in this chapter TINS compared to state-of-the-art process-based middleware: Goldrush and Damaris. We have shown that TINS without isolation can be up to 12% faster than the Goldrush approach on analytics parallelized with TBB. We have also shown that the static helper core approach of Damaris performs better than TINS without isolation for sequential and heavy analytics. We have implemented a static helper core approach in TINS thanks to a dedicated analytics master thread and we have shown that TINS with static helper core approach can be up to 42% faster than TINS without isolation on sequential and heavy analytics. TINS with static helper core approach is also more stable across the different analytics. However, in a static helper core approach, the choice of the number of static helper cores is essential to avoid wasting resources during synchronization points. The optimal number of helper core is found so that the simulation and analytics execution times are equal but this number is analytics dependent and requires a manual calibration step from the user. We will see in the next chapter how to make this choice automatic and we will in particular introduce a dynamic helper core strategy with a temporary thread isolation where the choice of the number of dynamic helper core is less punitive than for the static case.

6 Implementation of a Dynamic Helper Core Strategy with Automatic Sizes

The static helper core approach has mostly two drawbacks. First, it does not harvest the sequential regions of analytics and simulation. Secondly, an optimal number of static helper cores has to be found to reduce resource wasting during synchronization points. This optimal number is analytics dependent and requires a calibration step that is time and resource consuming for the end-user. In this chapter, we propose three different methods to enhance the performance of the static helper core approach and to limit and even avoid the choices that the user has to make. We first propose an adaptive static helper core approach where the arena sizes adjust themselves to find an optimal where the simulation and analytics execution times are roughly equals (Section 6.1). After highlighting the technical limitations to implement such an approach, we design a dynamic helper core strategy with a temporary thread isolation that enables to use the sequential regions of simulation and analytics (Section 6.2) and we implement it in TINS. The dynamic helper core approach still requiring the choice of the analytics arena size, we improve the approach toward an adaptive dynamic helper core strategy with an automatic choice of the analytics arena size (Section 6.3).

6.1 Implementation of an Adaptive Static Helper Core Approach

To limit the calibration step required to run in situ analytics in a static helper core approach, our first idea has been to implement an adaptive static helper core approach where the arena sizes automatically adjust themselves so that the simulation and analytics execution times are roughly equals (Section 6.1.1). However, the implementation of this approach presents technical limitations and we discuss the reasons why we did not choose this solution (Section 6.1.2).

6.1.1 Design of the Algorithm

When executing asynchronously an analytics parallelized with TBB, the execution time of the analytics can be adjusted by the size of the analytics arena. The analytics arena size being linked to the simulation arena size ($n_a + n_s = N$), the analytics arena size also has an impact on the simulation execution time. As we have already seen in Chapter 5, if the analytics arena size increases, more resources are allocated to the analytics and hence the analytics execution time decreases. However, fewer resources are allocated to the simulation and the simulation execution time increases. Similarly, when the analytics arena size decreases, the analytics execution time

increases and the simulation execution time decreases. The analytics arena size can therefore be the parameter to optimize if we want to have simulation and analytics execution times equals. However, because the arena size is an integer and because of jitters in the execution times due to the execution context, finding an analytics arena size such that the execution times are equals is nearly impossible and we try to find the analytics arena size close to the optimal, that is to say when the execution times are roughly equals.

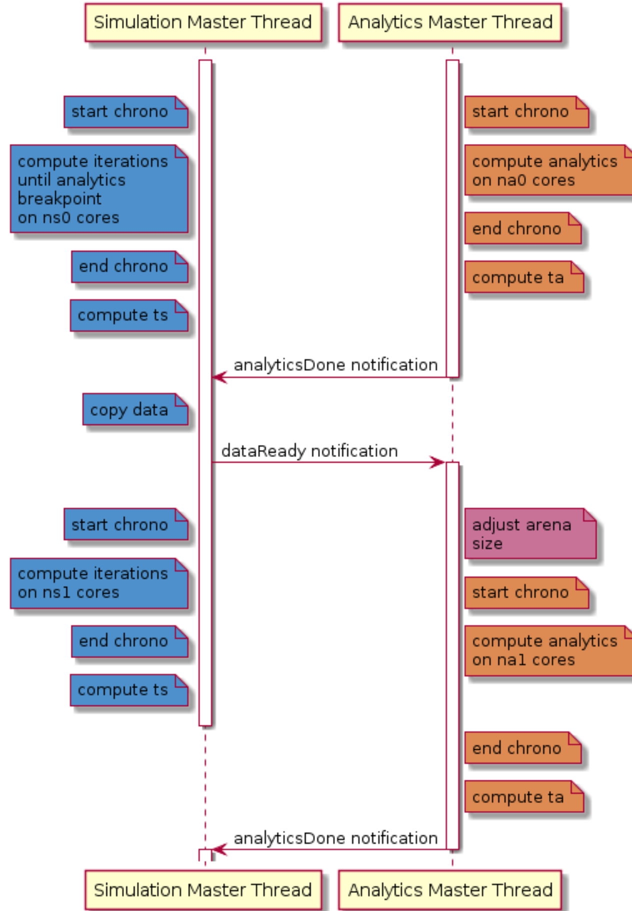

Figure 6.1 | UML sequence diagram of the simulation and analytics master threads time measurements.

We propose to measure the execution times of the two master threads as highlighted in Figure 6.1. When the simulation master thread begins to compute the iterations, it starts a chronometer, computes the iterations with a simulation arena of size n_{s0} and ends the chronometer when it reaches the next analytics breakpoint. On the other side, the analytics master thread starts a chronometer when it gets the `dataReady` notification, launches the analytics execution on an analytics arena of size n_{a0} and stops the chronometer when the analytics ends. t_s and t_a are then the elapsed time between the two chronometers on the simulation and analytics sides respectively. Before resuming to the next iteration, the analytics master thread adjusts the arena sizes by comparing t_a and t_s:

- if $t_a > t_s$, it may mean that the analytics does not have access to enough resources. The next analytics arena size is therefore $n_{a1} > n_{a0}$ (and hence $n_{s1} < n_{s0}$);

- if $t_a < t_s$, it may mean that the simulation does not have access to enough resources. The next analytics arena size is therefore $n_{a1} < n_{a0}$ (and hence $n_{s1} > n_{s0}$).

The algorithm to find arena sizes close to the optimal is similar to a dichotomy. We start from a beginning condition and we try different analytics arena sizes until we find the one that gives simulation and analytics arena sizes roughly equals. Here we propose to start with $n_a = 1$ because we want to remove as few resources as possible from the simulation in order not to slow down too much the simulation. The analytics arena size can vary between 1 and $N/2$ where N is the number of cores in the processor. This upper limit is chosen so that the simulation does not run on fewer threads than the analytics. Such an algorithm may converge in a few iterations only because N corresponds to the number of cores, which does not exceed 72 for modern processors.

6.1.2 Highlighting the Limitations of the Approach

Execution Times Measurements

The first difficulty to implement this approach in TINS is related to the position of the arena adjustment in the analytics master thread timeloop. In Figure 6.1, the analytics master thread adjusts the arena size when it gets the `dataReady` notification. At this point, it can have access to both simulation and analytics execution times and it can apply the adjustment algorithm. It thus computes the next arena sizes and has to update the arena sizes by destroying the arenas and creating them with the new sizes. However, at that point, the simulation has resumed to the next iteration and tasks are already executed in the simulation arena. It is therefore not possible to destroy the simulation arena.

A solution to this issue would be to execute the arena adjustment algorithm on the simulation master thread side, before sending the `dataReady` notification. However, as it will be seen in more details in Chapter 7, we would like TINS to decouple simulation and analytics executions and in particular, we do not want the simulation to be aware of the analytics execution. Making the simulation master thread execute the arena adjustment algorithm goes against this principle.

The only way to execute the arena adjustment algorithm on the analytics master thread while still being able to destroy the simulation arena is to add an extra synchronization between the two master threads. When the data have been copied, the simulation is blocked until the analytics master thread has destroyed and recreated the arenas. However, this solution limits the asynchronous character of our system, potentially leading to performance loss.

Execution Times Comparison

Another difficulty in this approach is the confidence we put in the time measurements and in particular how to compare the execution times. The execution times may present jitters because of the resource usage and the way the worker threads execute the tasks. Moreover, ExaStamp does not perform exactly the same operations at each iteration. For example, the neighbor lists can be executed given refinement criteria and not at every iteration, as it is shown in Figure 6.2. The execution time of the analytics, on the other hand, is expected to be more stable throughout the iterations. These jitters have to be taken into account when comparing the execution times in order not to make mistakes in the choice of the next analytics arena size.

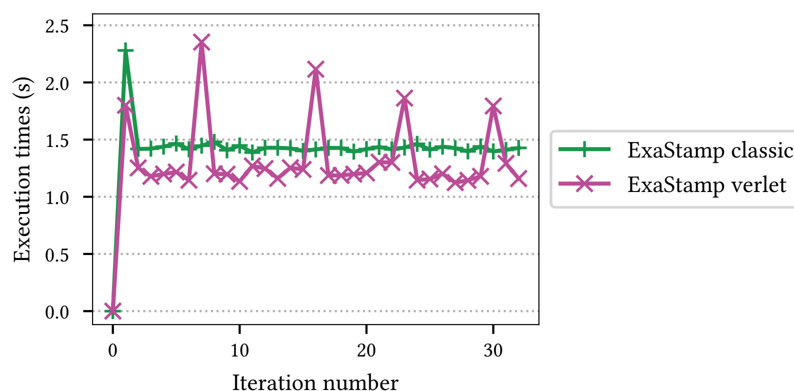

Figure 6.2 | Execution time per iteration of ExaStamp with two configurations: classic configuration where the neighbor lists is updated after each iteration (green) and verlet configuration where the neighbor lists is updated given refinement criteria (magenta). ExaStamp simulation of 32 iterations of 4,000,000 particles on one Broadwell node (28 cores).

We have proposed in this section an approach to find an optimal number of static helper cores without any calibration steps. However, in the perspective where simulation and analytics codes are decoupled, implementing this approach in TINS requires an extra synchronization between the two master threads that would limit the asynchronous properties of TINS. Moreover, this approach still does not harvest the sequential regions of both simulation and analytics. We therefore decided not to implement this method in TINS and to focus on a *dynamic helper core* strategy that seemed to be more promising. We will see in the following section the design of the dynamic helper core approach to harvest the simulation and analytics sequential regions and we will see in Section 6.3 how it offers a more flexible framework for the implementation of the adaptive algorithm.

6.2 Implementation of a Dynamic Helper Core Strategy with a Temporary Isolation

The static helper core approach does not harvest the simulation and analytics sequential regions because the threads are separated into two groups and they cannot execute tasks in the other group, even if no tasks are available in their own group. The goal of this section is to enable the threads to switch of arena when no more tasks are available in their original arena. We first design a temporary thread isolation with TBB that enables a more dynamic execution of tasks (Section 6.2.1) and propose two implementations of this *dynamic helper core* strategy in TINS (Section 6.2.2). We then evaluate TINS with dynamic helper core compared to TINS with static helper core (Section 6.2.3) and highlight the limitations of TBB for some analytics.

6.2.1 Designing a Temporary Thread Isolation with TBB

The idea behind a dynamic helper core strategy is to combine the benefit of both the static helper core approach and the task-based execution without isolation. The idea is similar to the dynamic approach of Callisto [57]: we would like to have two groups of threads when both simulation and analytics tasks are available but to remove the isolation when only one kind of task is available.

The idea is summarized in Figure 6.3. In this example, the 6 threads (T0 to T5) are split into two groups: 4 threads for the simulation (T0 to T3) and 2 threads for the analytics (T4 and T5). In a static helper core strategy (Figure 6.3 left), when the simulation master thread enters sequential regions (grey areas), T1 to T3 are idle because they have no tasks to execute. In this example, the analytics is quicker than the simulation. T4 and T5 are thus idle after the analytics execution, waiting for data to be copied by the simulation. The idle periods are highlighted by the purple areas.

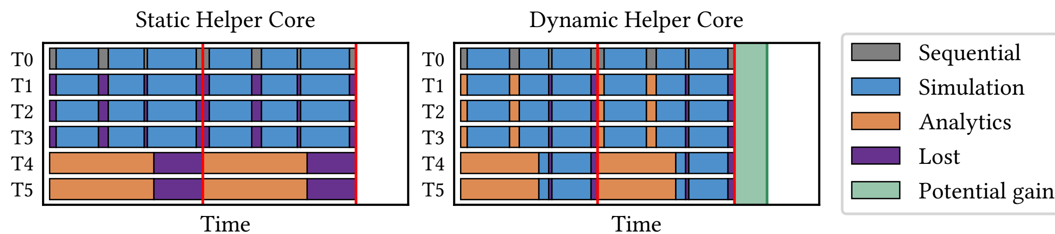

Figure 6.3 | Gantt diagram of the execution of simulation and analytics tasks on 6 threads (T0 to T5) with a permanent thread isolation (left) and with a temporary thread isolation (right). T0 is the simulation master thread and T1 to T5 are worker threads. T0 to T3 are threads dedicated for the simulation and T4 and T5 execute analytics tasks. The diagram shows two iterations of a simulation alternating parallel regions (blue areas) and sequential regions (grey areas). The analytics is composed of one parallel region (orange areas). The purple areas highlight the periods when the threads are idle. The green area highlights the potential gain in execution time of the right approach.

The dynamic helper core approach aims at harvesting the thread idleness periods (Figure 6.3 right). Instead of being idle during simulation sequential regions, T1 to T3 participate in the analytics execution. In the same way, T4 and T5 execute simulation tasks instead of being idle at the end of the analytics execution. The goal is twofold. First, the method aims at reducing the thread idleness periods. Secondly, as the simulation threads can execute analytics tasks (and respectively the analytics threads can execute simulation tasks), the simulation and analytics execution times should be reduced, hence leading to smaller total execution times.

The implementation of a dynamic helper core strategy is made possible by the TBB arenas. So far, we have used two arenas of sizes n_s and n_a such that $n_s + n_a = N$, N being the number of cores in the processor. These sizes prevent the threads to migrate to the other arena when they have no tasks to execute. In the example of Figure 6.3, $n_s = 4$, $n_a = 2$ and $N = 6$. When the simulation enters a sequential region, T1 to T3 do not have tasks to execute in the simulation arena. Instead of being idle, they could enter the analytics arena to participate in analytics execution but T4 and T5 are already involved in the analytics execution and the analytics arena size is set to 2. No extra thread can therefore enter the analytics arena because the arena is at its maximum occupancy. This observation led to the idea of setting arenas such that $\boldsymbol{n_s + n_a > N}$. This way, the simulation threads could enter the analytics arena during simulation sequential regions because the analytics arena would not be at its maximum occupancy.

As already explained in Chapter 3, the TBB scheduler never creates more than $N - 1$ worker threads to avoid core over-subscription. When using two arenas of sizes such that $n_s + n_a > N$, satisfying the request of the two arenas would require more threads than available. Three cases can therefore be distinguished:

- if there are no tasks in the simulation arena (during simulation sequential regions or when the simulation master thread waits for the `analyticsDone` signal), the analytics arena can

contain the requested $n_a - 1$ worker threads (assuming that $n_a \leq N$);

- if there are no tasks in the analytics arena (when the analytics master thread waits for the `dataReady` signal or during analytics sequential regions), the simulation arena can contain the requested $n_s - 1$ worker threads (assuming that $n_s \leq N$);

- if simulation and analytics tasks exist concurrently, the arenas get a number of worker threads proportional to their requests. As TBB can provide $N-1$ worker threads, the arenas will respectively get $(n_a - 1)\,(N - 1)\,/\,(n_a + n_s - 2)$ and $(n_s - 1)\,(N - 1)\,/\,(n_a + n_s - 2)$ worker threads.

Work stealing in this context works as follow. When simulation and analytics tasks exist concurrently, threads are separated into two groups and steal tasks inside their own arena. When there are no simulation tasks to steal anymore, the worker threads involved in the simulation arena can enter the analytics arena, given the maximum occupancy of $n_a - 1$ worker threads. The same holds when there are no analytics tasks anymore. Let assume now that $n_a - 1$ worker threads are involved in the analytics arena because the simulation is in a sequential region. When the sequential region ends, simulation tasks are created again by the simulation master thread. In this situation, TBB provides a migration mechanism so that worker threads can change of arena [4]. Some of the threads will leave the analytics arena to enter the simulation arena so that both arenas get a number of threads proportional to their request.

6.2.2 Implementation of the Temporary Thread Isolation in TINS

Choice of the Arena Sizes

TBB arena feature allows to design a dynamic helper core approach with a temporary thread isolation by setting the arena sizes such that $n_a + n_s > N$. The question is now how to choose n_a and n_s. Our first idea has been to set $n_a = n_s = N$ such that all the threads can execute simulation and analytics tasks and all the threads can enter the other arena instead of being idle. However, when tasks of both types exist concurrently, half of the threads execute simulation tasks and the other half execute analytics tasks. It is therefore as if simulation and analytics were given the same priority. One of the issues of such approach is illustrated in Figure 6.4. In Figure 6.4 (left), the analytics executed on half of the threads is quicker than the simulation. When the analytics ends, all the threads migrate to the simulation arena but a lot of simulation sequential regions cannot be harvested because no analytics tasks may be executed anymore. In Figure 6.4 (right), the analytics runs on fewer threads and is hence longer than in Figure 6.4 (left) but the simulation runs quicker because more threads execute simulation tasks and more simulation sequential regions can be used to run analytics. At the end, these two effects could combine to reduce the total execution time. To test this effect and hence give a higher priority to the simulation, we fix the simulation arena size to $n_s = N$ so that all the threads can execute simulation tasks and the analytics arena size is left as a parameter to be fixed. The dynamic helper core strategy of TINS is summarized in Figure 6.5. We will see in the next section the influence of the analytics arena size on the execution of the different analytics.

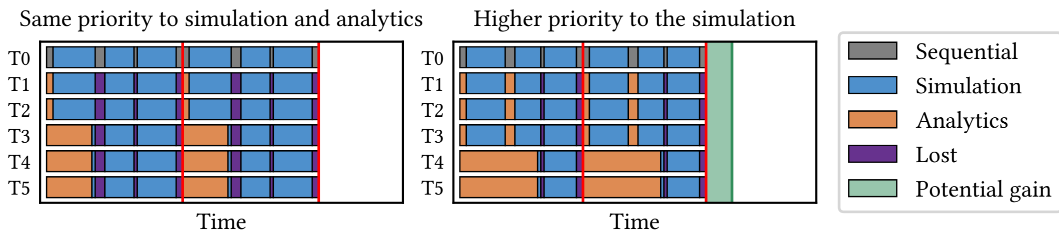

Figure 6.4 | Gantt diagram of the in situ execution of an analytics with dynamic helper core strategy when giving the same priority to simulation and analytics (left) or when giving a higher priority to the simulation (right). In the latter case, the analytics execution time is longer but fewer resources are idle during simulation sequential regions. The green area highlights the potential gain in execution time of the right approach.

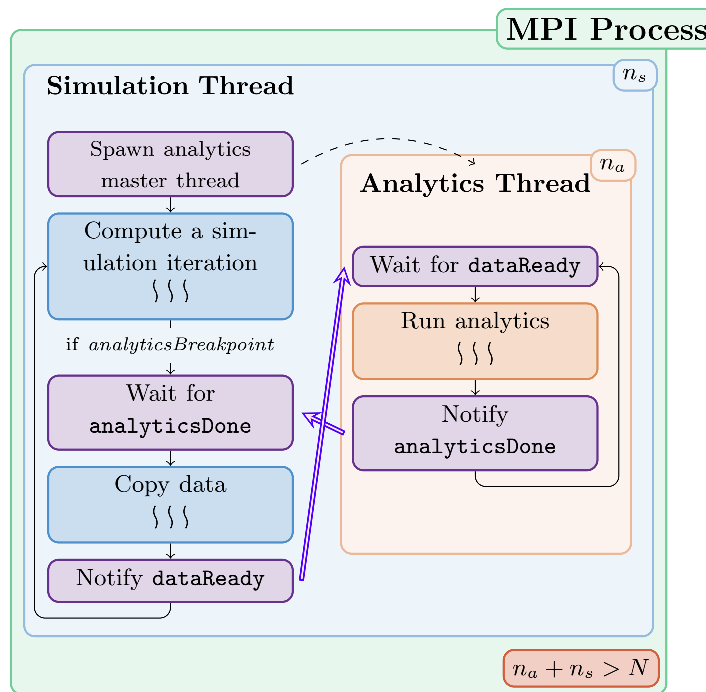

Figure 6.5 | Dynamic helper core strategy in TINS. The simulation arena size is set to $n_s = N$ where N is the number of cores in the processor and the analytics arena size $n_a > 1$ is left as a parameter to be chosen so that $n_a + n_s > N$.

Core Over-subscription or Restricting Resources

In ExaStamp, the TBB scheduler is initialized with default number of worker threads, that is to say $N-1$ on a processor with N logical cores. In the static helper core approach where the arena sizes are set such that $n_a + n_s = N$, the arenas request a total of $N-2$ worker threads. With the two master threads, N threads are thus executed on N cores, without core over-subscription. The situation is different in a dynamic helper core context. Here, the cumulative request of worker threads by the two arenas is larger than the number of worker threads instantiated by TBB and the scheduler allocates a proportional number of threads to each arena, leading to $N-1$ worker threads scheduled to execute simulation or analytics tasks. However, with the 2 master threads existing in TINS, a total of $N+1$ threads run on N cores and the cores are over-subscribed.

We saw two solutions for the implementation of the dynamic helper core strategy in TINS. The first solution is to keep the default behavior of TBB and let TBB and the OS handle the core over-subscription. The second solution is to force TBB to create only $N-2$ worker threads. Before TBB initialization, the analytics master thread is pinned on the core 0 and a mask is created from core 1 to N. This way, the TBB scheduler only sees $N-1$ cores in the system and creates $N-2$ worker threads. We saw advantages and drawbacks in both solutions. The second solution avoids core over-subscription but one thread is removed from the simulation execution. Indeed, as seen in Chapter 3, the analytics master thread cannot execute simulation tasks because it cannot enter the simulation arena and at most $N-1$ threads (the simulation master thread and $N-2$ worker threads) can enter the simulation arena. In this solution, the analytics master thread is often idle for small analytics. On the other hand, the first solution enables the simulation arena to get N threads when the analytics arena is empty but it leads to core over-subscription during the concurrent execution of simulation and analytics tasks.

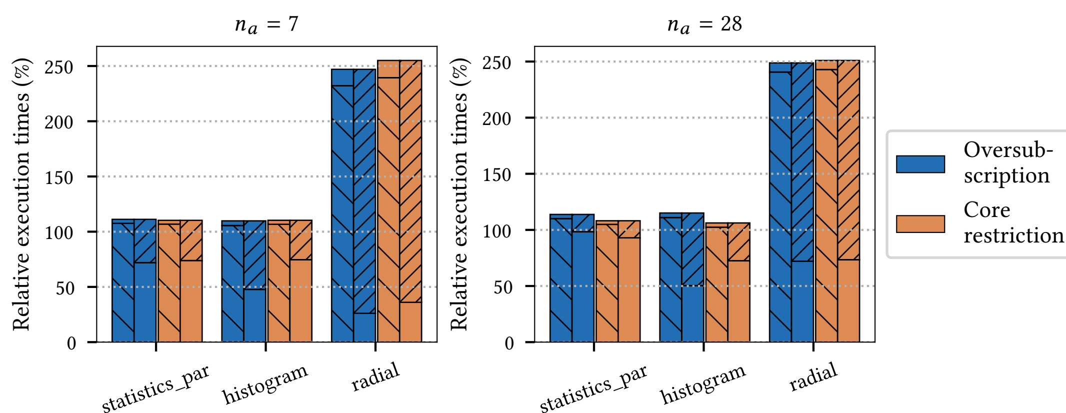

Figure 6.6 | Comparison of the two solutions (core over-subscription or restricting resources) for three analytics and for $n_a = 7$ and $n_a = 28$. The total execution times are normalized with respect to the execution time of ExaStamp alone on 28 cores per node. ExaStamp simulation of 32 iterations on 64 Broadwell nodes (1,792 cores).

We implemented the two solutions in TINS and compared them on a set of analytics. Figure 6.6 summarizes the results for `statistics_par`, `histogram` and `radial` with $n_a = 7$ and $n_a = 28$. When the analytics arena is set to 7, both techniques are equivalent for `statistics_par` and `histogram`. The `radial` execution is 3% slower with the core restriction than with the core over-subscription but this difference decreases with the analytics arena size (less than 1% difference with $n_a = 28$). On the contrary, the core restriction has smaller execution

times for the other analytics when the analytics arena size is set to 28. Restricting the cores is indeed 5% and 7% faster than the core over-subscription method for `statistics_par` and `histogram` respectively. We therefore decided to keep the core restriction approach that seems to be slightly better than the core over-subscription method for our target analytics.

Binding Strategies

In the previous chapter, we have shown that, for a static helper core strategy, binding the arenas to two distinct subsets of the cores leads to better performance than letting TBB schedule them on all the cores. In a dynamic helper core strategy, the simulation arena does not need to be bound because it encompasses all the cores. The analytics arena could on the contrary benefit from a binding close to the analytics master thread. Figure 6.7 compares the execution times of the execution of `statistics_par` with a dynamic helper core approach and with different sizes for the analytics arena when enabling or not the binding of the analytics arena on the n_a first cores. As seen in Chapter 5, the n_a first cores correspond to the first NUMA nodes, the analytics master thread being bound to core 0 and the temporary buffer to NUMA node 0, close to the analytics master thread. Both approaches are equivalent when $n_a = 28$ because the analytics arena encapsulates all the threads. For smaller analytics arena sizes, binding the analytics arena has a negative impact on the total execution times: it is respectively 8% and 13% longer for $n_a = 4$ and $n_a = 7$ compared to a strategy where the analytics arena is not bound. We therefore decided not to bind the analytics arena.

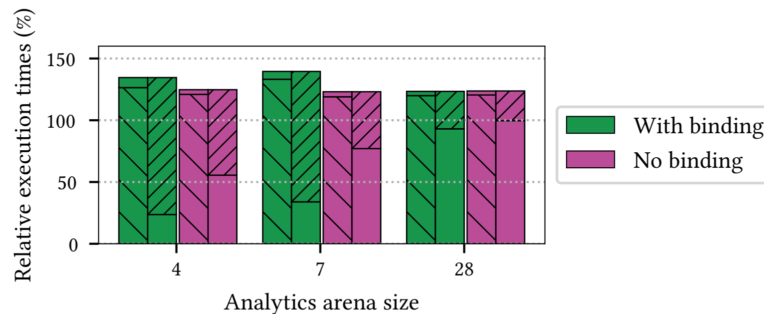

Figure 6.7 | Execution of `statistics_par` in a dynamic helper core context with different sizes for the analytics arena and when enabling or not the binding of the analytics arena on the n_a first cores that correspond to the first NUMA nodes, close to the analytics master thread and the temporary buffer locations. The total execution times are normalized with respect to the execution time of ExaStamp alone on 28 cores. ExaStamp simulation of 32 iterations on 1 Broadwell node (28 cores).

6.2.3 Evaluation of the Dynamic Helper Core Approach

Comparison of Static and Dynamic Helper Core Approaches

Figure 6.8 compares TINS without isolation, TINS with static helper core strategy (TINS SHC) and TINS with dynamic helper core strategy (TINS DHC) on the four analytics used in Section 5.3.3. Each bar of the plot corresponds to a helper core strategy with an analytics arena size. The analytics arena sizes range between 1 and 14 for the static helper core approach and between 1 and 28 for the dynamic helper core approach. The static and dynamic helper core approaches are compared when using the same analytics arena sizes, except for the two right bars. In this

situation, we compare the static helper core approach with an analytics arena of size 14 and the dynamic helper core approach with an analytics arena of size 28. These configurations are comparable because, in the dynamic helper core approach, the two arenas will get 14 threads each if tasks of both types exist concurrently. An analytics arena of size 1 leads to similar execution times for the static and dynamic approaches because they have internally the same behavior: one thread is dedicated to analytics and the remaining $N - 1$ threads execute simulation tasks. This comes from the resource restriction explained in Section 6.2.2.

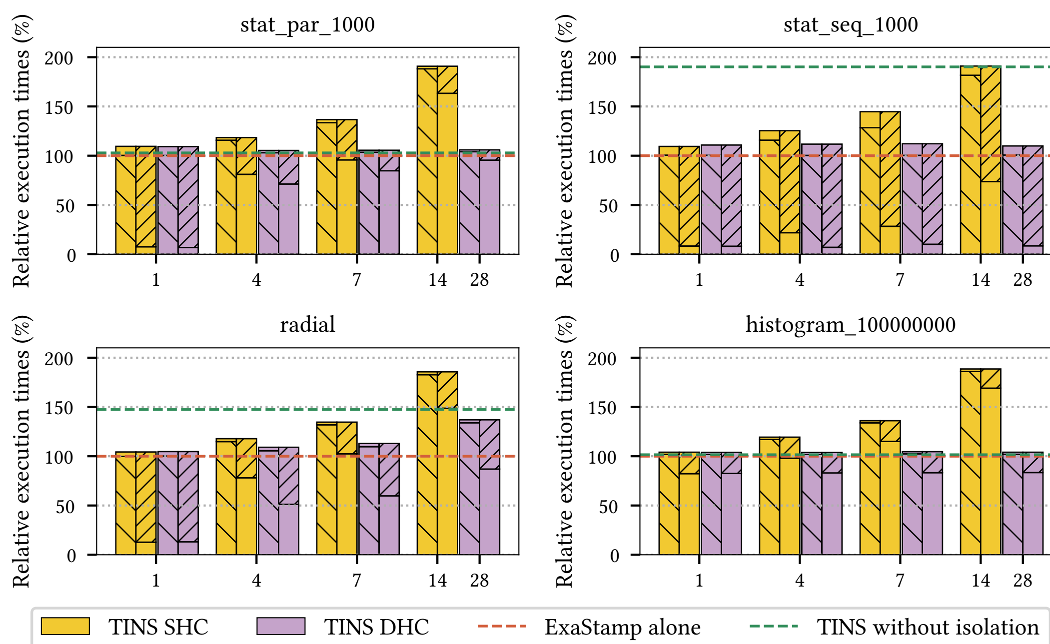

Figure 6.8 | Comparison of the different strategies implemented in TINS (without isolation, with SHC and with DHC) on 64 Broadwell nodes for different analytics arena sizes. The simulation is composed of 32 iterations on 256,000,000 particles, the analytics being performed after each iteration. The total execution times are normalized with respect to the execution time of ExaStamp alone. The TINS without isolation and TINS SHC execution times correspond to the execution times discussed in Chapter 5 and Chapter 4.

For the four analytics, the dynamic helper core execution is faster than the static helper core execution with similar arena sizes. If we leave the `radial` analytics aside at first, we notice also that the dynamic helper core strategy is much less sensitive to the configuration than the static helper core approach. In the `histogram` computation, the execution times are less than 1% different from one configuration to another (the difference being respectively 2% and 3% of difference for `statistics_seq` and `statistics_par`). In a static helper core approach, the total execution time for 14 static helper cores is nearly twice as much as the total execution time for 1 static helper core. In a dynamic helper core context, the parallel analytics (`statistics_par` and `histogram`) can be performed within an overhead of less than 5% with respect to ExaStamp alone, and the sequential analytics (`statistics_seq`) can be performed within an overhead of 10%.

The `radial` analytics shows a different behavior for the dynamic helper core strategy. When the analytics arena size increases, the total execution time also increases. An analytics arena of size 1 induces an overhead of 6% with ExaStamp alone and this overhead reaches 39% with an analytics arena of size 28. TINS DHC with an analytics arena of size 28 is still 7% faster than

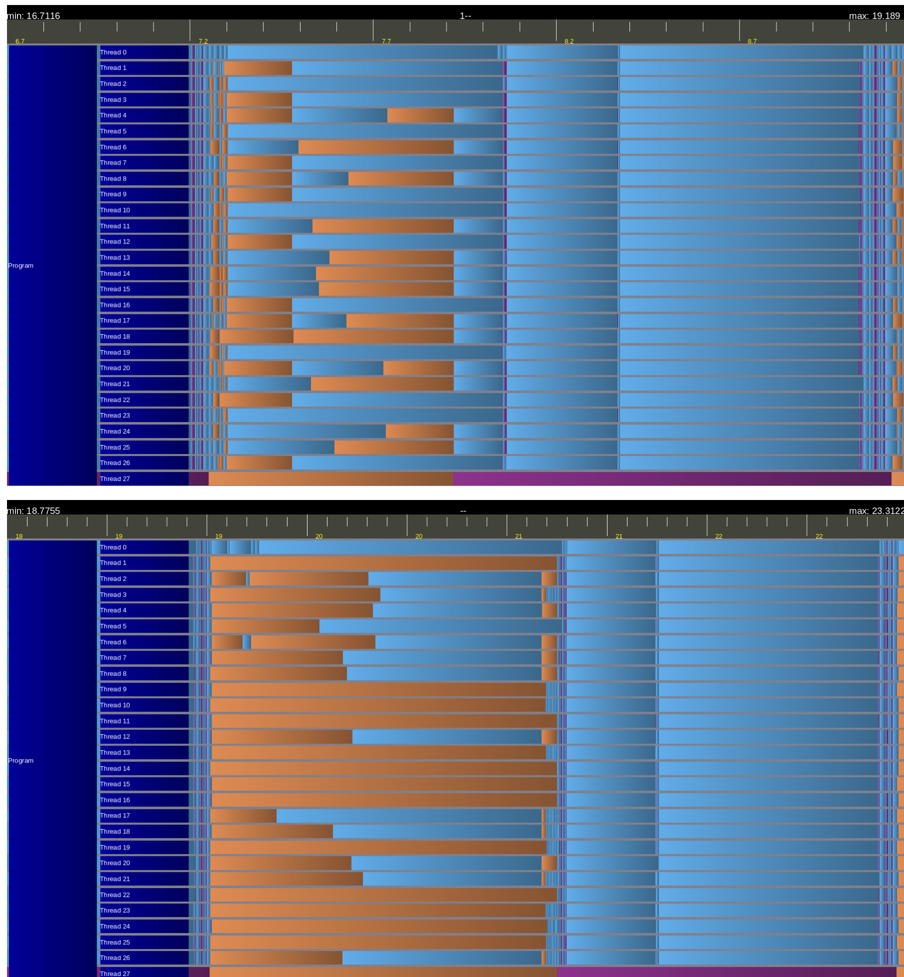

Figure 6.9 | Traces of the in situ execution of `statistics_par` (top) and `radial` (bottom) in a dynamic helper core context where the analytics arena size is set to 28 on a 28-core Broadwell node and for one iteration of ExaStamp. The traces show when the threads are in the simulation arena (blue) or in the analytics arena (orange). Purple areas highlight periods when the threads are not assigned to an arena. Thread 0 (top) is the simulation master thread and thread 27 (bottom) the analytics master thread.

TINS without isolation. To understand what happens with this analytics, we need to go more in depth into the task execution.

Figure 6.9 shows the traces of the executions of `statistics_par` and `radial` during one iteration of ExaStamp with an analytics arena size of 28. As explained in Chapter 5, the trace measurement has been modified to visualize only when the threads enter and leave the arenas. For the `statistics_par` analytics, we see that the beginning of the iteration is characterized by periods where the threads alternate between simulation and analytics arenas. When looking into more details, we observe that there are only few periods when all the threads are in the analytics arena. Most of the time, 14 threads belong to the analytics arena and 14 threads belong to the simulation arena, showing that the TBB scheduler allocates half of threads to each arena when simulation and analytics tasks exist concurrently and migrates threads from the analytics arena to the simulation arena when simulation tasks are created again after sequential regions. The `radial` analytics shows a different pattern. We see in Figure 6.9 that 27 threads are in the analytics arena for a long time before they switch to simulation execution. Simulation tasks

are still created by the simulation master thread but the TBB scheduler does not migrate the threads from the analytics arena to the simulation arena as in the `statistics_par` execution. The simulation tasks are therefore executed only by the simulation master thread, increasing the execution time of the simulation.

This comes from the thread migration mechanism of TBB. Indeed, the worker threads can discover that they need to leave the analytics arena to enter the simulation arena only when they are in their stealing loop and when they do not execute a nested parallel algorithm [4]. The `radial` analytics is precisely composed of a nested parallel loop and is concerned with this arena migration issue. The effect is all the more visible when the analytics arena size increases because more threads get trapped in the analytics arena and cannot migrate to the simulation arena. This effect can lead to performance loss for two reasons. First, if the analytics is not efficient enough on 27 cores, 27 threads are in the analytics arena but the task execution would be longer than in the case where fewer threads were involved in the execution. Moreover, the traces do not show if the threads always have a task to execute when they are trapped in the analytics arena. Secondly, this analytics highlights our intuition in Figure 6.4: the analytics is executed at the beginning of the simulation iteration and the simulation sequential regions of the end of the iteration cannot be harvested because there are no analytics tasks to execute anymore. Moreover, the analytics master thread spends half of the iteration idle. Allocating fewer thread to the analytics increases the analytics execution time but it is beneficial for the total execution time.

To conclude this comparison, TINS with dynamic helper core is less sensitive to the size of the analytics arena than TINS with static helper core. We see in particular that $n_a = 4$ or $n_a = 7$ are a good tradeoff for the analytics execution. It enables to execute the parallel analytics on more than one core and it prevents that all the worker threads get trapped in the analytics arena when the analytics is composed of a nested parallel loop. With these configurations, the in situ execution of the four analytics can be performed with an overhead of less than 12% over ExaStamp alone.

Evaluation of the Different Methods on an Iteration Varying Workload at Scale

So far, we have compared the different approaches with the same analytics being performed after each iteration. During a simulation, the physics of the system evolves and the end-user studies different phenomena. It results in complicated analytics workflows where different analytics are executed with different frequencies. We have seen in Section 6.1 that it is possible to find an optimal number of static helper cores for one analytics but finding an optimal number of static helper cores for several analytics may be difficult because the analytics may have different needs. For example, the optimal number of static helper cores may be 4 for one analytics but this number is of course not optimal for a sequential analytics.

We evaluate in Figure 6.10 the capacity of the dynamic helper core strategy to execute iteration varying workloads at scale. We compare Damaris, TINS SHC and TINS DHC on an ExaStamp simulation of 2 billions atoms using 14,336 Broadwell cores (512 nodes). The simulation is composed of 32 iteration and after the second iteration, an analytics is executed after each iteration in the following scheme:

- iterations 3 - 12: `statistics_par`;
- iterations 13 - 22: `histogram`;
- iterations 23 - 32: `radial`.

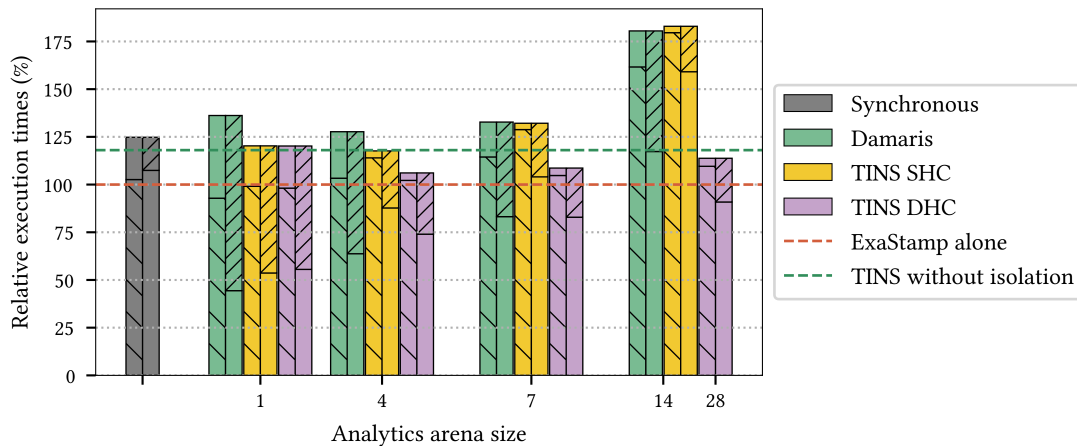

Figure 6.10 | Comparison of Damaris, TINS with static helper core and TINS with dynamic helper core with different analytics arena sizes when executing three different analytics on 14,336 Broadwell cores (512 nodes) on an ExaStamp simulation of 2 billions atoms. The total execution times are normalized with respect to the execution time of ExaStamp alone.

Given that the analytics arena size is greater than 1 so that the analytics can be executed in parallel and smaller than 28 to give a higher priority to the simulation, TINS DHC shows similar execution times, whatever the configuration. In particular, TINS DHC always has the smallest execution time, showing that the static helper core approach fails at finding an optimal number of static helper cores for this iteration varying workflow. TINS DHC can be up to 40% faster than Damaris and TINS SHC. Moreover, TINS DHC enables the in situ execution of analytics with an overhead of 6% with respect to ExaStamp alone.

We have shown in this section that a temporary thread isolation allows to implement a dynamic helper core strategy where the worker threads are free to migrate to simulation or analytics arenas. In particular, the threads are split into two groups when both simulation and analytics tasks exist concurrently but they can enter the other arena instead of being idle when no tasks are available in their arena. We have shown that TINS with dynamic helper core can be up to 40% faster than TINS with static helper core on an analytics workflow where the analytics being executed change over time. The dynamic helper core still requires to choose the analytics arena size but most of the analytics are not sensitive to the choice of the arena size. This is not the case of the `radial` analytics, however, for which the analytics arena size has to be smaller than the number of cores in the system to avoid disturbing the simulation. We will see in the next section how we can adapt the algorithm presented in Section 6.1 in the dynamic helper core context to automatize the choice of the analytics arena size.

6.3 Implementation of an Adaptive Dynamic Helper Core Approach

In a dynamic helper core context, a tradeoff has to be found between the priority we want to provide to the simulation and the parallelism we want to give to the analytics. We have seen in

the previous section that the dynamic helper core approach is less sensitive to the analytics arena size. For most of the analytics, $n_a = n_s = N$ is a good choice because it allows the TBB scheduler to balance the threads and to migrate all of them into one arena if necessary. However, the thread migration mechanism of TBB has some limitations because the threads can discover that they need to migrate to another arena only when they are in a specific state. For some analytics, it leads to too much threads dedicated to the analytics and the simulation execution time is severely impacted. In this section, we propose to adapt the algorithm presented in Section 6.1 in a dynamic helper core context so that the analytics arena size can be chosen automatically (Section 6.3.1). We then validate the approach on two analytics (Section 6.3.2) and highlight the limitations of the approach (Section 6.3.3).

6.3.1 Design of the Algorithm

The dynamic helper core context is more flexible than the static helper core context to implement the arena adjustment algorithm presented in Section 6.1. Indeed, in this context, the simulation arena size is always set to the number of cores and is not modified throughout the execution. The algorithm can be executed after the analytics master thread has received the `dataReady` notification because the analytics arena size can be adjusted no matter of what happens on the simulation master thread side. There is therefore no need for an extra synchronization between the two master threads.

Choice of the Starting Condition

In the static helper core algorithm, we proposed to begin the algorithm with $n_a = 1$ to remove as few resources as possible from the simulation. In the dynamic helper core context, we propose to begin the algorithm with $n_a = N$. We have indeed shown that $n_a = n_s = N$ is a good choice for most of the parallel analytics. The analytics arena size has also no impact on the execution of sequential analytics because one thread executes the analytics and the other threads enter the simulation arena, no matter the size of the analytics arena. The only analytics where $n_a = N$ is not a good choice are the analytics where the thread migration of TBB shows its limitations. The optimal analytics arena size is therefore searched between $n_{min} = 1$ and $n_{max} = N$.

Choice of the Next Analytics Arena Size

The algorithm starts with the highest analytics arena size and decreases it to find an optimal size. The optimal arena size is such that the elapsed time between two `dataReady` signals is the smallest. Given an analytics arena of size $n_0 = n_{max} = N$, the analytics master thread first measures t_0 as the elapsed time between the reception of two `dataReady` signals with an analytics arena of size n_0. The analytics arena size is then reduced ($n_1 = (n_{max} + n_{min})/2 = (N + 1)/2$) and the analytics master thread measures t_1 as the elapsed time between the reception of the two `dataReady` signals with an analytics arena of size n_1. It then compares the two elapsed times:

- if $t_1 > t_0$, the first configuration was better than the second one and the optimal arena size is searched between n_{max} and $n_{min} = n_1$ and the next analytics arena size is set to $n_2 = (n_{min} + n_{max})/2$;

- if $t_1 < t_0$, the second configuration was better than the first one and the optimal arena size is searched between $n_{max} = n_1$ and n_{min} and the analytics arena size is set to $n_2 = (n_{min} + n_{max})/2$.

The algorithm is then applied again with the new analytics arena size and the new upper and lower limit until it finds an optimal analytics arena size.

Execution Times Comparison

As we have explained in Section 6.3, one time measurement is not sufficient to discriminate an analytics arena size. Indeed, ExaStamp does not always perform the same operations at every iteration and the execution times can be disturbed by the execution environment. We therefore propose to measure the execution times 10 times and to compute the mean, the standard deviation and the coefficient of variation of the 10 measurements. We distinguish three cases:

- if the new configuration is more than 5% quicker and the coefficient of variation is low, the new configuration is better than the former and the analytics arena size is decreased for the next 10 measurements;
- if the new configuration is more than 5% longer, the configuration is worst than the former and the arena size is increased for the next 10 measurements;
- if the new configuration is less than 5% longer but the coefficient of variation is high, the configuration is worst than the former and the arena size is increased for the next 10 measurements;

In all the other cases, the algorithm cannot make a decision and leaves this arena configuration for the next 10 measurements.

Breaking Conditions

We distinguish two different breaking conditions. If the algorithm cannot make a decision three times in a row, the algorithm stops and we keep the configuration with the smallest analytics arena size. The algorithm begins with a high analytics arena size and reduces it until it reaches a configuration worse than the former. At that point, the analytics arena size is increased again until it finds an optimal. When the algorithm is in an upward phase, it stops when the computed analytics arena size is the same for two sets of measurements.

6.3.2 Validation of the Approach

Figure 6.11 shows the total execution time of the adaptive method for the `radial` and the `statistics_par` analytics compared with the dynamic helper core approach with different analytics arena sizes for a simulation of 128 iterations. For the `statistics_par` analytics, the execution time is dominated by the simulation and we can see that the adaptive method gives similar execution times. Modifying the analytics arena size does not affect the simulation execution time. The algorithm stops at iteration 32 with an analytics arena of size 14 because it was unable to make a decision three times in a row.

The adaptive method is important for the `radial` analytics because the analytics arena size has an influence on the simulation execution time. We measured that the optimal analytics arena

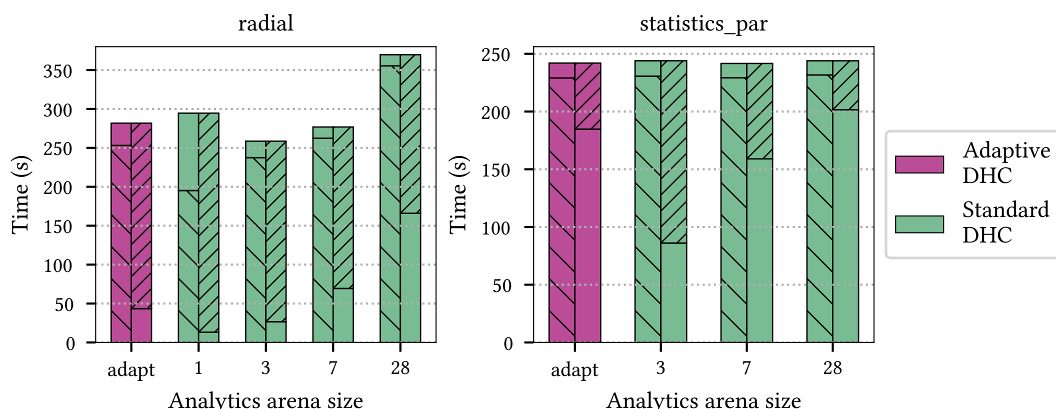

Figure 6.11 | Execution times of the adaptive dynamic helper core approach for the `radial` and the `statistics_par` analytics compared with the dynamic helper core approach with different analytics arena sizes. ExaStamp simulation of 128 iterations with analytics executed after each iteration on 1 Broadwell node (28 cores).

size is 3 for this analytics and we observe that the adaptive method is better than the dynamic helper core approaches with $n_a = 1$ and $n_a = 28$ and equivalent to the dynamic helper core approach with $n_a = 7$. The total execution time is 10% longer with the adaptive method than with the optimal dynamic helper core approach because of the number of tests it takes to find the optimal solution. Indeed, as it can be seen in Figure 6.12, the adaptive method determines $n_a = 3$ to be the optimal analytics arena size but it takes 72 iterations to make this decision. This difference will become even smaller as the number of iterations increases.

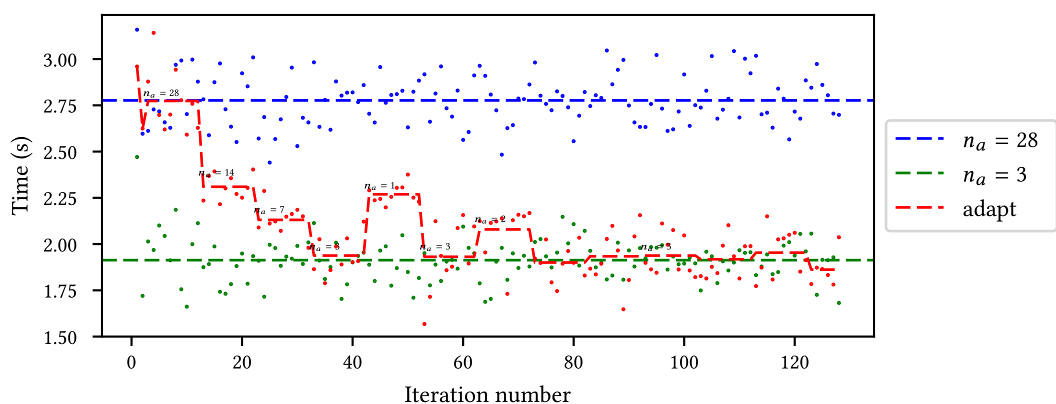

Figure 6.12 | Execution time per iteration of the `radial` execution in three different dynamic helper core configurations: $n_a = 28$, $n_a = 3$ and with adaptive dynamic helper core. The dashed lines represent means of the iteration times to highlight the different stages of the adaptive algorithm.

6.3.3 Highlighting the Limitations of the Approach

The dynamic helper core context offers a more flexible framework for the implementation of an algorithm to adapt iteratively the analytics arena size. In particular, it does not require an extra synchronization between the two master threads. However, there are still limitations to this approach. The algorithm being iterative, it requires a certain number of iterations before finding

the optimal arena size. For example, it required 72 iterations to find that $n_a = 3$ was the best configuration for the `radial` analytics. During the first iterations, the analytics arena size was not the optimal and computation time was lost. This effect would be reduced when increasing the number of iterations.

We have shown in Section 6.2.3 that the dynamic helper core strategy is well adapted for iteration varying workloads where the executed analytics are not the same at each analytics breakpoint. Here, the adaptive algorithm proved to be efficient when the executed analytics is the same at each analytics breakpoint. When the analytics change over time, the workload also changes and the algorithm as such is not applicable. Indeed, the elapsed times measured between two iterations do not match if the analytics are not the same. To circumvent this issue, one idea would be for the analytics master thread to keep track of the iterations where the analytics workload varies. For example, suppose that we execute two analytics (A_1 and A_2) at two different frequencies (f_1 and f_2). The analytics master thread can keep track of the analytics breakpoints when A_1 only is executed, when A_2 only is executed and when A_1 and A_2 are executed together. The analytics master thread would therefore save three analytics arena sizes and update each of them accordingly.

We have adapted in this section the algorithm presented in Section 6.1 for a dynamic helper core context. The dynamic helper core context is indeed less restrictive than the static helper core context. In particular, no extra synchronization is necessary for its implementation. The approach proved to be efficient to find the optimal analytics arena size for the `radial` analytics, hence reducing the execution time compared to the situation when $n_a = N$. This approach still has limitations, in particular the number of iterations it requires to find the optimal arena size.

6.4 Chapter Summary

We have designed in this chapter a dynamic helper core strategy with a temporary thread isolation that uses the TBB arena feature. The worker threads can execute both simulation and analytics tasks and can in particular enter the other arena instead of being idle when no tasks are available in their own arena. We have shown the benefits of the dynamic helper core approach compared to the static helper core approaches implemented in TINS and in Damaris, TINS with dynamic helper core being up to 40% faster than the other methods on a set of analytics. Simulations on up to 14,336 cores of 2 billions particles show that the dynamic helper core approach is able to execute dynamic analytics workflows with an overhead of less than 7% over ExaStamp alone. The dynamic helper core approach needs an analytics arena size to be chosen for setting a priority to the simulation. This approach proved to be weakly sensitive to the analytics arena size, except for some analytics where the migration mechanism of TBB shows limitations. We have thus implemented an adaptive algorithm in the dynamic helper core context and we have shown that it enables to find an optimal analytics arena size in a few iterations, for analytics that are sensitive or not to the analytics arena size.

6.5 Part Summary

In the previous three chapters, we have developed TINS, a task-based in situ method inside ExaStamp. We have measured the performance of TINS on different analytics and we have compared it with state-of-the-art middleware. TINS is based on an analytics master thread spawned by ExaStamp at its initialization. The analytics are coded in ExaStamp but they use a simple data representation as an input and they are not aware of ExaStamp complicated data structure. In the perspective of having a portable in situ solution, we would like the analytics to be coded outside of the simulation code. The next chapter will focus on the architecture of the TINS framework, in particular on how we make use of the dedicated analytics master thread to separate simulation and analytics codes.

Part III

Toward an Evolutive Task-Based Hybrid Framework

7 Design of a Framework to Automatically Orchestrate Analytics Execution

In the three previous chapters, we have focused on the performance of our task-based in situ method. We have in particular shown how the use of a dedicated thread and of TBB mechanisms enables to execute analytics in situ with the simulation using a dynamic helper core strategy. This chapter is more oriented toward the architecture of the TINS framework focused on two characteristics. We want TINS to be a *generic* framework that may be integrated into other simulation codes than ExaStamp and an *intuitive* tool where non expert users can easily develop new analytics, test them and execute them asynchronously with the simulation in a transparent way. The architecture of TINS is designed so that simulation and analytics codes are kept well separated, the analytics being developed as plugins loaded at runtime in the simulation (Section 7.1). Analytics workflows are easily described in an external file and TINS automatically creates a task-based graph of plugins matching the user requirements (Section 7.2). TINS is designed to be an evolutive architecture where new features, such as in transit and post-processing capabilities, can easily be added in the existing framework (Section 7.3).

7.1 Orchestration of Simulation and Analytics Codes

The first step toward the development of our generic and intuitive framework is made by separating simulation and analytics codes. As already seen in Chapter 4, developing analytics inside the simulation code has the advantage of reducing data copy and data transformations but it also shows drawbacks. In particular, the analytics developer needs to know the simulation data structures to integrate analytics routines. Moreover, integrating the analytics inside the simulation code may lead to difficulties when it comes to test the analytics. Indeed, it implies that the simulation code is compiled and started each time a change is performed on the analytics codes. Finally, it limits the genericity of the approach because the data structures are different from one simulation to another and analytics can thus not be easily used by different simulation codes. On the other hand, separating simulation and analytics codes allows to decouple simulation and analytics data representations and thus to use a simpler data structure for the analytics. The analytics being compiled as separated codes, the testing process is also improved and analytics can be used in several simulations without any code modifications.

To decouple simulation and analytics codes, we propose the TINS architecture described in Figure 7.1 in the form of a pseudo-UML. It is organized around a `TINSManager` interface known both by the simulation and the analytics codes. An object `OrchestratorManager` derives from

the interface and is instantiated by the simulation as a singleton (Section 7.1.1). There is one `OrchestratorManager` object per MPI process. The analytics are developed as C++ plugins, holding a pointer to the `TINSManager` instance created by the simulation (Section 7.1.2). The plugins are compiled as separate libraries and loaded at runtime by the analytics master thread spawned by the simulation (Section 7.1.3).

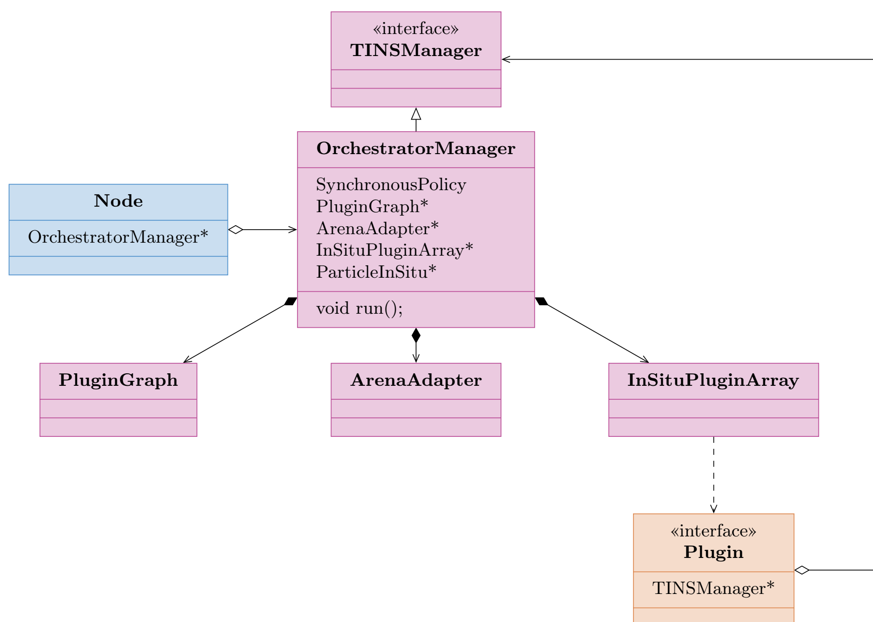

Figure 7.1 | Overview of TINS architecture in ExaStamp.

7.1.1 Integration of TINS Architecture in the Simulation Code

During the initialization, an `OrchestratorManager` object is instantiated by the simulation. The simulation master thread spawns the analytics master thread presented in Chapter 5 as a C++ thread that executes the `OrchestratorManager::run` method. This method corresponds to an infinite `while` loop as sketched in Figure 5.10. It first waits for data to be ready, executes the analytics in the analytics arena when it has received the `dataReady` signal, sends the `analyticsDone` notification when analytics execution is completed and finally resumes to the infinite `while` loop, waiting for the next data to be ready. Data sharing between simulation and analytics master threads are performed in the shared memory. In particular, the notifications are handled by atomic booleans. In the following, the analytics master thread will be called *orchestrator* because the role of this thread is to orchestrate simulation and analytics executions.

The implementation of TINS in ExaStamp corresponds to the addition of an `OrchestratorManager` object into the `Node` object and to the modification of the end of the `doComputeWork` method presented in Figure 3.9. The `OrchestratorManager` object is created during ExaStamp initialization based on the input data file of the simulation code. The input data file is read by the `OrchestratorManager` object to retrieve information about the in situ mode it should use and eventually the analytics arena size chosen by the user. We support the different in situ modes introduced in the last three chapters: synchronous and asynchronous without isolation (Chapter 4), asynchronous with static helper core (Chapter 5) and asynchronous with dynamic helper core

with or without adaptive analytics arena size (Chapter 6). In the latter case, the `ArenaAdapter` class implements the adaptive method introduced in the previous chapter. The synchronous and asynchronous without isolation modes are slightly adapted compared to the approaches presented in Chapter 4. The analytics master thread is present in both cases and simulation and analytics tasks are submitted in the same arena. An extra synchronization is also added in the synchronous case so that the simulation does not resume to the next iteration immediately after having copied the data but waits for the analytics completion.

The primary goal of the simulation being to solve the numerical equations, it should not be aware of the analytics being executed or the frequency with which they are performed. This is the role of the orchestrator and the `doComputeWork` method is modified to meet this requirement. In particular, the `writeIO` function is replaced by a `copyParticle` function. At any time, the orchestrator knows the iteration number of the next analytics breakpoint, based on the analytics graph it created (see Section 7.2 for more details). The next analytics breakpoint iteration is then made available to the simulation master thread in shared memory. When the simulation enters the `copyParticles` function, it checks whether the iteration corresponds to the next analytics breakpoint. If it is not the case, it resumes to the next iteration. If it is the case, it copies its data into a `ParticleInSitu` structure hold by the orchestrator, sends the appropriate notification and resumes to the next iteration. Notice that the copy is performed by the simulation on a location specified by the orchestrator. To reduce the intrusion into the simulation code, an alternative would have been to let the orchestrator perform the data copy. However, we wanted TINS to be generic and not to know the simulation data structure. Making the orchestrator perform the data copy would have necessitate the description of the AOSOA data structure of ExaStamp in TINS, limiting the genericity of the approach. Works like Conduit [2] have been released during this thesis and could yet be used to easily describe the AOSOA structure of ExaStamp in TINS, enabling the data copy on the orchestrator side. The development of such approach and the comparison with the existing one will be performed in a future work. The checkpointing capacities of the simulation are also not modified by our framework. The simulation still writes checkpoint files periodically to the filesystem but the file output for later post-processing is moved to the plugin system.

7.1.2 Development of Analytics Outside of the Simulation as TINS Plugins

In TINS, the analytics are developed as C++ classes compiled as shared libraries and loaded at runtime by the orchestrator. The classes derive from the `Plugin` interface (Figure 7.2) that defines the mandatory members and methods that a plugin must implement. The main element of the `Plugin` class is the pointer to the `TINSManager` object that is the building block of our intuitive system. Each plugin owns a pointer to the `TINSManager` singleton instantiated by the simulation and the object acts as a bridge between simulation data and the analytics by providing methods to get access to the attributes of its `ParticleInSitu` data structure. The `TINSManager` object thus hides both the synchronization between the simulation and the analytics and the way data are stored by TINS. This way, analytics developers do not need to care about the storage layout of TINS and this latter can be modified without any modifications in the analytics codes.

A plugin has three compulsory members: a name (`m_name`) used by the user and by TINS to create the analytics graph, an iteration from which the analytics has to be executed (`m_itbegin`) and a frequency (`m_freq`). The execution of the plugin is then decomposed in three functions.

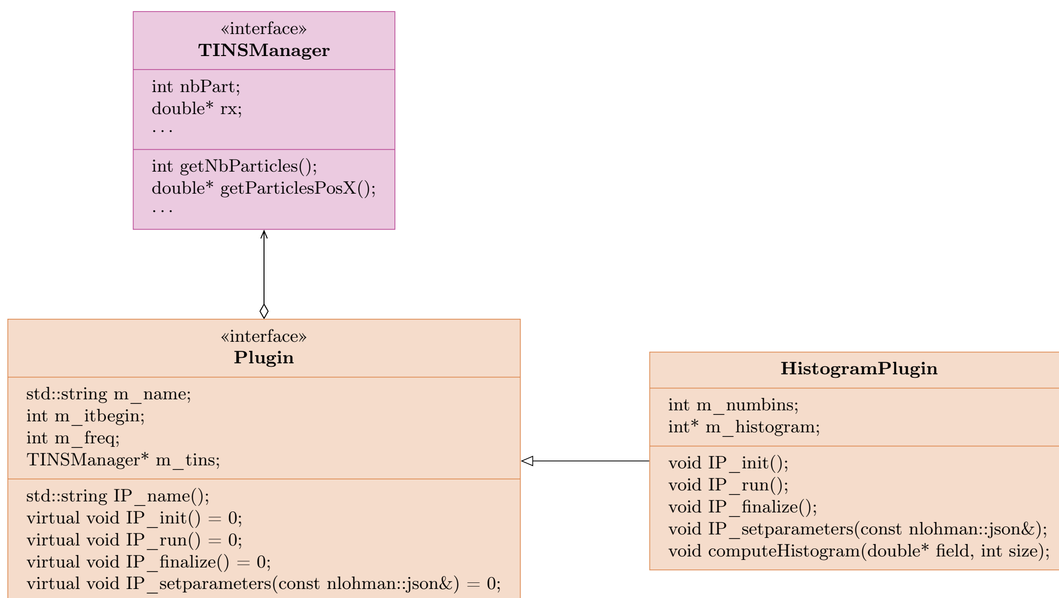

Figure 7.2 | Class diagram of the `Plugin` interface and the `HistogramPlugin` that derives from it.

`IP_init` is called once at the `m_itbegin` iteration and corresponds to the initialization of the plugin (allocation of the class members for example). `IP_run` is executed according to the frequency of the plugin and corresponds to the analytics execution. `IP_finalize` is called during TINS finalization to free the allocated data. As an example, the `HistogramPlugin` class presented in Figure 7.2 is implemented to compute an histogram of the positions along the `x`-axis. The `IP_init` and `IP_finalize` methods are used to allocate and free the histogram array (`m_histogram`) that will be filled at each iteration, reducing the allocation costs. The number of particles and the positions along the `x`-axis are retrieved thanks to the `TINSManager` members (`getNbParticles()` and `getParticlesPosX()`). To benefit from the dynamic helper core strategy implemented in TINS, the `IP_run` method should be parallelized with TBB whenever possible. As we will see in more details in Section 7.2.1, the `IP_run` methods of the desired analytics will indeed be executed in the analytics arena.

The members of the plugins are defined in an external JSON data file. Two kinds of members are distinguished: the compulsory ones (the frequency and the iteration when the analytics should begin) and the members specific for a class. In the `HistogramPlugin` example, the user can set the number of bins (`m_bins`) in the JSON file. This allows to modify the parameters of the analytics without recompiling the plugins. The `IP_setparameters` method is used to set the members according to the JSON input file or to define default parameters.

The main advantage of this class structure is that it allows to keep persistent data throughout the execution of the simulation. For example, writing data with the Hercule library developed at CEA [111] requires to open a Hercule base in an initialization phase and to use this object every time data are output. With this class structure, such an object can be instantiated in the `IP_init` method, destroyed in the `IP_finalize` method and used at each call to the `IP_run` method. The development costs are also kept minimal because the developer just needs to implement the `IP_run` method as if the analytics was a post-processing code. The in situ execution of the plugin

is transparent to the user.

7.1.3 Compilation and Loading of TINS Plugins

The classes corresponding to the different plugins are compiled as shared libraries with one shared library per defined class. The shared libraries are loaded in the simulation during the `OrchestratorManager` initialization thanks to the `dlopen`, `dlsym` and `dlclose` functions. The orchestrator is then in charge of the instantiations of the plugins. However, the `dlopen` and `dlsym` functions retrieve pointers to functions in a shared library by using the symbols associated to the functions and they show limitations when it comes to loading C++ functions and classes. While the symbol of a C function corresponds to the name of the function, C++ compilers use different symbols for the functions to handle function overloading for example. This is called the *name mangling*. The name mangling being compiler specific and even version specific, it is nearly impossible to retrieve a pointer to a function based on the function name. To solve this issue, one solution is to define the functions that need to be loaded as C functions thanks to `extern "C"` pieces of codes. While this solves the problem of name mangling, it is not possible to declare a class as a C function. Moreover, we do not want a pointer to a class but an instance of it. The solution we adopted is to add two functions for each plugin (see Figure 7.3): a *creator* creates an instance of the plugin and a *destructor* destroys it. A plugin is therefore considered as *valid* only if it declares a `create` function (line 8) that returns a pointer to the class instance, a `destroy` function (line 14) that deletes a class instance and a `pname` variable (line 6) that defines the plugin name.

```
1  typedef Plugin* create_t(TINSManager*);
2  typedef void destroy_t(Plugin*);
3
4  extern "C"
5  {
6    const char* pname = "histogram";
7
8    Plugin* create(TINSManager* tins)
9    {
10     std::string name(pname);
11     return new HistogramPlugin(name, tins);
12   }
13
14   void destroy(Plugin* p)
15   {
16     delete p;
17   }
18 }
```

Figure 7.3 | Definition of the class creator and destructor as C functions to ease the loading of the plugins.

During the orchestrator initialization, the `InSituPluginArray` object of the orchestrator scans a specific folder where the available plugins have been installed and loads the different plugins. It then checks whether the plugin is valid by looking at the `create`, `destroy` and `pname` symbols. If one of the three symbols is missing, the plugin is considered as not valid and is discarded. If the three symbols are present, the plugin is stored in an array of plugins later

used by the orchestrator for the graph creation.

We have detailed in this section the design of the TINS architecture where simulation and analytics are kept well separated. The analytics are developed as C++ plugins loaded at runtime by an orchestrator thread spawned by the simulation. The simulation is not aware of the analytics execution and a shared object allows data sharing between simulation and analytics. The available plugins are loaded during simulation execution and the valid ones are stored in an array of plugins. We will see in the next section how the user can describe their analytics workflow and how TINS automatically creates a graph of plugins based on the user requirements.

7.2 Automatic Creation of a Graph of Plugins

To analyze the data produced by a simulation, the user needs to define complex analytics workflows. The analytics workflows are composed of several components: analytics codes are used to extract information from the simulation data, visualization routines are used to visualize the states of the system and file output are used to save important data into the filesystem. Some components may need data produced by other components, leading to dependencies between the different tasks. Analytics workflows can thus be seen as a directed graph and we detail in this section how TINS creates such a graph of plugins. The users describe their analytics workflow in a JSON file (Section 7.2.1) that is automatically transformed into a TBB flow graph by TINS (Section 7.2.2). The analytics graph is executed at each analytics breakpoint and tools are provided to limit data copy and to allow time dependent analytics (Section 7.2.3).

7.2.1 Definition of the Analytics Workflow

As seen in Chapter 2, several methods are used in the literature to define analytics workflows, Python scripts and XML files being the most widely used. In TINS, analytics workflows are defined thanks to an external JSON data file as described in Figure 7.4. No recompilations of the simulation or the plugins are necessary when the user wants to change the analytics workflow. The JSON file corresponds to a list of plugins with their compulsory and additional attributes. We chose the JSON format for expressing the workflow because it has a simple key value system that looks like the text input files used by most of the simulation codes but it is more flexible. Indeed, it is possible to define parameters for each analytics without hard-coding all the configurations in the input data reader. Thanks to JSON libraries for C++, handling JSON objects looks like manipulating C++ objects, which eases the integration into our C++ plugins. Finally, writing a JSON file is easier than an XML file for non expert users.

During the initialization of the `OrchestratorManager` class, the orchestrator reads the JSON file containing the analytics asked by the user. For each asked analytics, the orchestrator uses the plugin name to check whether the plugin is available in its array of plugins. If this is not the case, the orchestrator issues a warning and goes to the next asked analytics. If the asked plugin is available, the orchestrator calls the corresponding creator and sets the class parameters by calling the `IP_setparameters` method. The class instance is then added to an array hold by the `PluginGraph` object, the role of the `PluginGraph` class being to create the analytics graph.

```
1  [
2      {
3          "name": "statistics_par",
4          "itbegin": 8,
5          "freq": 4
6      },
7      {
8          "name": "histogram",
9          "itbegin": 8,
10         "freq": 8,
11         "numbins": 1000
12     },
13     {
14         "name": "write",
15         "itbegin": 0,
16         "freq": 16,
17         "path": "/tmp"
18     }
19 ]
```

Figure 7.4 | Example of JSON input file to describe the analytics workflow.

7.2.2 Construction of a Graph of Plugins

We decided to rely on the TBB flow graph feature to define and execute the analytics graph. As seen in Chapter 3, the TBB flow graph feature allows to describe graphs as a set of tasks linked together by explicit dependencies. In TINS, each node of the graph is a plugin and the edges are data dependencies between the plugins. We chose the TBB flow graph feature for the graph construction because it connects well with our task-based in situ framework. Each plugin is executed as a TBB task submitted in the analytics arena by the orchestrator. If the `IP_run` method of the plugin is parallelized with TBB, children tasks are spawned inside a plugin task. All the tasks are then available in the analytics arena, providing a good potential for the analytics to be interleaved with the simulation execution.

Node Construction

The nodes of the graph are implemented as `continue_node` objects defined in the TBB flow graph API. We made this choice for several reasons. First, data are not explicitly passed from one node to another because the `TINSManager` object is used to retrieve the particles attributes. If a plugin needs to modify a particles attribute, it submits the modifications to the `TINSManager` instance that is in charge of updating the stored information. Secondly, we want the node to wait for its predecessors completion before starting its computations. Indeed, even if data are not explicitly passed from one node to another, a plugin may compute or update a parameter used by its successor plugins and the `continue_node` class guarantees that a node waits for its predecessor completion. The `PluginGraph` object holds a flow graph object and creates the different `continue_node` in this graph object thanks to lambda functions, as described in Figure 3.4. The lambda function calls the `IP_init` and `IP_run` methods of the plugin at the first execution and the `IP_run` method only for the following iterations.

Automatic Edge Construction

Now that we have constructed the nodes of the graph, the second step consists in the definition of the edges between the nodes. The solution the most widely used in the community is to let the user describe the analytics dependencies in a Python script. In TINS, the edge construction is automatized so that our framework is as automatic and as dynamic as possible. This is done thanks to extra parameters defined in the JSON input file (Figure 7.5). The user describes for each analytics the parameters required for the plugin execution (`input`) and the parameters that are computed and/or modified during the plugin execution (`output`). The object-oriented design of the JSON input allows to easily add the input and output information, either with an array of fields or an empty array. The input and output fields can be fields produced by ExaStamp (`rx`, `ry`, `rz`), fields produced by the plugins and made available through the `TINSManager` object (`rx_filtered`) or virtual fields to enforce a dependency between two plugins (a).

```
1  [
2      {
3          "name": "plugin_a",
4          "input": [ "rx" ],
5          "output": [ "rx_filtered" ],
6          "mpi": false
7      },
8      {
9          "name": "plugin_b",
10         "input" : [ "rx", "ry", "rz" ],
11         "output" : [ "a" ],
12         "mpi": true
13     },
14     {
15         "name": "plugin_c",
16         "input": [ "rx_filtered", "a" ],
17         "output": [ ],
18         "mpi": false
19     }
20 ]
```

Figure 7.5 | Modification to the JSON input file to describe the plugin inputs and outputs and to tell the orchestrator whether a plugin performs MPI communications or not.

The graph is constructed by comparing the input and output of each pair of plugins. If the intersection between the two sets is not empty, an edge is added between the first plugin and the second plugin. In the case a plugin modifies a field produced by the simulation, the output field must have a different name than the input field. In the example of Figure 7.5, `plugin_a` applies a transformation to the `rx` field and its output is the `rx_filtered` field. We add this rule so that some plugins can be executed on the data produced by the simulation while other can be executed on the data transformed by the plugins and so that non expected dependencies are not added because of name matching.

The algorithm leads to the construction of a graph as the one sketched in Figure 7.6 (left). However, the graph defined as such may lead to deadlocks because of the plugins that perform MPI calls (communications or explicit barriers). In the example of Figure 7.6, purple nodes are plugins that perform MPI calls. For the first stage of the graph, the only dependencies are from ExaStamp to the three nodes, A, B and C. As soon as data are made available by the simulation, the

three tasks corresponding to the three successor plugins are spawned. There is no guarantee in the order of execution because we do not know which children task is spawned first and because of the work stealing mechanism. In particular, when using several MPI processes, there is no guarantee that every process will execute the tasks in the same order. While this is not an issue for local tasks like the plugin *B*, this can lead to deadlocks when the plugins perform MPI calls. Indeed, let us consider that one MPI process begins with the execution of the plugin *A* while another MPI process begins with the execution of the plugin *C*. At some point, the first process may wait for the processes involved in the MPI call of the plugin *A* while the second process may wait for the processes involved in the MPI call of the plugin *C*, leading to a deadlock.

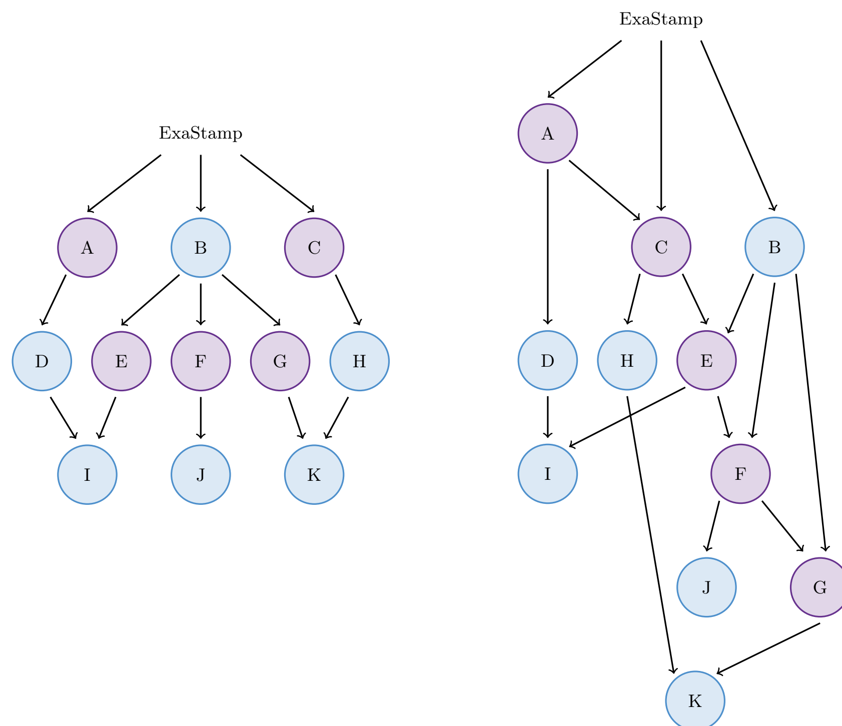

Figure 7.6 | Graph of plugins without taking care of MPI plugins (left) or by adding extra dependencies between the plugins that perform MPI communications to prevent two plugins to perform MPI communications concurrently (right). The purple nodes correspond to plugins that perform MPI communications.

To solve this issue, the user indicates in the JSON input file the plugins that perform MPI communications thanks to the `mpi` flag (Figure 7.5). The graph construction is then performed in two steps. The first step is to create a graph without using the `mpi` flag by scanning the input and output of all pairs of asked plugins. In a second step, a breadth-first search algorithm is performed to serialize the nodes that perform MPI calls by adding extra dependencies between them. The first node with the `mpi` flag (*A* in the example) is kept in memory. When the algorithm traverses another node with the `mpi` flag (*C* in the example), it creates an edge between the two nodes, removes the first node from its memory and keeps the second node. The algorithm stops when all the nodes have been traversed. At the end, the MPI nodes have extra dependencies, as sketched in Figure 7.6 (right). The plugins performing MPI communications are linked to each other so that there is only one plugin performing MPI communications at a time. They can be executed simultaneously with plugins that do not perform MPI communications because there

is no risk of deadlock. Every MPI processes generate the same graph because they use the same JSON input file and they execute the same deterministic algorithm for the graph creation.

The different nodes of the graph are not necessarily executed from the same iteration, nor at the same frequency but only one graph is created. During the TBB flow graph execution at a particular analytics breakpoint, only the nodes whose frequency corresponds to that analytics breakpoint are executed, the other being discarded. When reaching an analytics breakpoint, the orchestrator scans the frequency of the different nodes to compute the next analytics breakpoint. This iteration number is then made available to the simulation so that it knows when it must copy data.

Every time data are made available by the simulation master thread, the orchestrator is in charge of launching the graph execution inside the analytics arena. The orchestrator can either perform computations asynchronously with the graph execution or take part in the graph execution. The `analyticsDone` notification is sent when the graph execution is over. For the moment, the graph construction is made once during initialization and it is not updated during the simulation lifetime but the system can easily be extended to dynamically add and remove nodes between two graph executions. The only requirements are to apply a deterministic algorithm when adding the nodes in the graph so that they are added at the same position in the graph on every MPI processes and to execute the breadth-first algorithm to serialize the MPI nodes.

The description of the input and output fields of a plugin is an extension of the notion of contract introduced by Mommessin et al. [85]. In their work, the user describes the nodes and edges between the nodes in a Python script and they describe the fields produced by the producer and consumed by the consumer. It enables to reduce the data copy but the user is still responsible to define the edges between the nodes. Here, we propose to use a similar mechanism to automatically create the edges between the nodes. We will see in the next section how this feature can be used to also reduce the data copy.

7.2.3 Management of Simulation and Analytics Data

TINS is intended to support a wide range of analytics, from simple analytics that only monitor a parameter of the simulation to more complex analytics that require information about the internal organization of simulation data. The primary goal of the TINS framework is to ease the development of new analytics plugins by hiding the synchronization with the simulation and the way data are stored by the orchestrator. The plugins just call the accessor methods provided by the `TINSManager` object and perform the analytics computation based on the provided data. To guarantee this ease of development, TINS must manage data in an efficient way and must provide tools to help with data representation. We detail in this section two features of TINS regarding data management: a mechanism to copy only the necessary data and the development of helper plugins to ease the development of complex analytics.

Copy of the Necessary Data Only

The time to copy data into the `ParticleInSitu` structure has not been shown so far because it was not predominant in the total execution time of TINS for the various analytics presented in Table 4.1. Indeed, for the various experiments presented in Chapters 5 and 6, the copy time has always been two orders of magnitude smaller than the simulation iteration. Optimizing the data copy may not seem important when the data being copied are the attributes of the `Parti-`

`cleInSitu` data structure but in the perspective of an evolutive framework, optimizing it may become important. For example, we will see in Chapter 8 that cells and ghost information can be copied to help the execution of some analytics, leading to copy times much more important, and even of the order of magnitude of a simulation iteration. It is therefore essential to copy these additional data only when necessary and not at every analytics breakpoint, to avoid loosing time and memory for data that will not be used at this analytics breakpoint.

We therefore decided to generalize the graph creation to indicate to the simulation what data should be copied at each analytics breakpoint. The system is based on a dictionary that gives the data that can be output by the simulation and the data that can be produced by the analytics and stored by the orchestrator. The input and output fields given for each plugin in the JSON input file must match entries in the dictionary. If a field does not match any dictionary entry, it is considered as virtual and is only taken into account for the graph creation. As already explained in Section 7.1.1, the orchestrator knows at each time the iteration of the next analytics breakpoint and makes it available to the simulation master thread. To copy only the necessary data, the orchestrator also computes the list of the data fields that should be copied for the next analytics breakpoint and transmits it to the simulation so that only the necessary data are copied.

For the implementation of the approach in ExaStamp, we rely on a new feature of ExaStamp where lambda functions can be used in the `Node` level to apply operations on ExaStamp internal data. The computations are still performed in the `Cell` level, and in particular ExaStamp still holds the data in a AOSOA data structure but high level representations of the data are provided to ease the manipulation of a set of ExaStamp attributes. To copy only the necessary data, a lambda function corresponding to the data fields to copy just needs to be constructed based on the orchestrator information. The dictionary is intended to be enriched in the future with new parameters. For the moment, we only support the attributes of the `ParticleInSitu` structure and ghosts and cells information as later discussed in Chapter 8. It is possible to add new entries in the dictionary as ExaStamp can provide more data output (molecules and polymers information, mechanical computations, ...). The only requirements are to add the structure in the dictionary and to provide the high level representation of the data to allow the partial copy with lambda functions.

Development of Helper Plugins to Handle Data Representation

So far, we have mostly presented analytics that use the particles attributes stored in the `ParticleInSitu` data structure for one iteration. For these analytics, the data representation does not have an influence, these analytics mostly iterating on the attributes arrays to compute parameters of interest. The data representation is yet important for some analytics. For example, we will see in Chapter 8 that the neighbor search on the particles can be speed up by using the cell information present in ExaStamp. In this section, we focus on temporal analytics that need the particles attributes of different iterations.

In TINS, data are copied at each analytics breakpoint and are overwritten at the next analytics breakpoint. The orchestrator does indeed not keep more than one iteration of the data in its buffers. To keep attributes of a given iteration, the analytics developer must add members to their plugin class and manage the data copy of simulation data to the plugin members. However, special care must be taken when comparing the attributes arrays of two iterations, as illustrated in Figure 7.7. In this example, 11 particles (indices 0 to 10) are distributed over two MPI processes.

Between the two iterations, the number of particles hold by each MPI process has changed and the order of the particles copied in the array has also changed. In particular, the 4th element of the array (highlighted in orange in Figure 7.7) corresponds to the 10th particle at iteration t_1 and to the 5th particle at iteration t_2. Data therefore need to be exchanged between the MPI processes in order to compare the arrays between the two iterations.

	Processus 0	Processus 1
array at t_1	0 4 1 10 7	8 2 6 9 3 5
array at t_2	0 4 1 5 10 7	1 8 2 3 9

Figure 7.7 | Data representation of 11 particles distributed across two MPI processes for two iterations.

A solution to this problem is to let the analytics developer discover by themselves the modifications in the data representation and perform the necessary MPI communications, but this breaks the rules of an intuitive and generic approach. Another solution is to modify the way the data are stored by the orchestrator so that the ith element of the arrays always corresponds to the same particle from one analytics breakpoint to the other. However, this approach means to reorganize data at every analytics breakpoint, even when the graph executed at this analytics breakpoint does not need this reorganization.

We therefore decided to externalize the data reorganization in a *helper plugin* that can be used in analytics workflows where one or several plugins need data reorganization. At each analytics breakpoint, the helper plugin stores the array representing the indices of the particles, compares them with the indices stored at the previous analytics breakpoint and computes a redistribution pattern so that the previous array matches the new array. The redistribution pattern includes internal reorganization of the array and the necessary MPI communications that need to be performed. The redistribution pattern can then be used transparently by the plugins that need reorganization of their arrays.

In the future, we intend to provide more helper plugins to the analytics developers. The goal of these helper plugins is to extract common patterns regarding data representation so that the analytics developer can directly use these tools and focus on the analytics implementation. Other helper plugins may include filter plugins to extract data corresponding to a particular region or conversion plugins to retrieve data in a desired physical unit.

We have seen in this section how TINS automatically creates a graph of plugins as a TBB flow graph executed in the analytics arena at each analytics breakpoint. TINS is designed to hide to the plugins the synchronizations with the simulation codes. This way, the plugins can be developed outside of the simulation code without any knowledge of the simulation or of the way data are stored. So far, we have mainly construct TINS as an in situ framework but some analytics are not well suited for an in situ mode. In particular, some analytics require MPI splittings different than the ones provided by the simulation and would require a lot of MPI communications that would disturb the simulation execution. Some other analytics have also an execution time much higher than the simulation execution time that makes it impossible to execute them in situ without

increasing the end-to-end execution time. We will see in the next section how TINS can be extended to enable an in transit execution of analytics, without any modifications in the plugin codes and with only a few modifications in TINS.

7.3 Extension of TINS with an In Transit Mode

As already discussed in Chapter 2, in transit processing has the advantage of executing heavy analytics concurrently with the simulation without using the resources of the simulation. Analytics are executed on a set of dedicated nodes, called the staging nodes and the in transit framework has to manage data transfer between the simulation nodes and the staging nodes, in a way that minimizes the impact on the simulation execution time. In this section, we integrate an in transit mode in TINS by using the orchestrator and the graph capacities already present in TINS (Section 7.3.1). We then show the benefit of the approach by implementing a prototype and by validating it on the `radial` analytics (Section 7.3.2). We finally show that the TINS extension for in transit processing can be used to execute analytics plugins in a standalone mode, offering a testing platform and a post-processing mode in TINS (Section 7.3.3).

7.3.1 Design of an In Transit Mode

Figure 7.8 shows the design of the in transit mode of TINS. It relies on the `TINSManager`, the `PluginGraph` and the `InSituPluginArray` classes and on two separate codes. The first code corresponds to the simulation code that implements the `OrchestratorManager` class as already seen. The only difference is that the `OrchestratorManager` class implements a `sendData` method for data transfers to the staging nodes. The second code is the in transit code that implements an `InTransitManager` class derivating from the `TINSManager` object and using the `PluginGraph` and `InSituPluginArray` classes for the graph construction. The in transit code is kept very simple. It first initializes the TBB scheduler on the staging nodes, the TBB schedulers being different on the different nodes. Then, for each in transit analytics breakpoint, the program waits for simulation data received through the `recvData` method implemented by the `InTransitManager` object, launches the in transit analytics graph execution and notifies the simulation nodes when the in transit computation is finished. The in transit code shows a similar timeloop than the orchestrator used in the simulation, except that data are received from distant nodes.

The in transit code uses the same mechanisms that were implemented for the in situ mode. In particular, the simulation and in transit codes share the same JSON file for the graph construction and no recompilation of the plugins is necessary to execute them in transit. To distinguish the plugins that are executed in situ or in transit, we added an `intransit` flag in the JSON file. In the in transit side, the `InTransitManager` object only takes into account for its graph construction the plugins with the `intransit` flag defined. In the simulation side, the `OrchestratorManager` object constructs the nodes based on all the plugins. For the in situ plugins, the nodes consist in calling the `IP_run` method of the plugin as already explained above. For in transit plugins, two nodes are created, one for transferring data to the staging nodes thanks to the `sendData` method of the `OrchestratorManager` class and one to wait for in transit analytics completion if necessary. In transit plugins are decomposed in two nodes because they need MPI communications

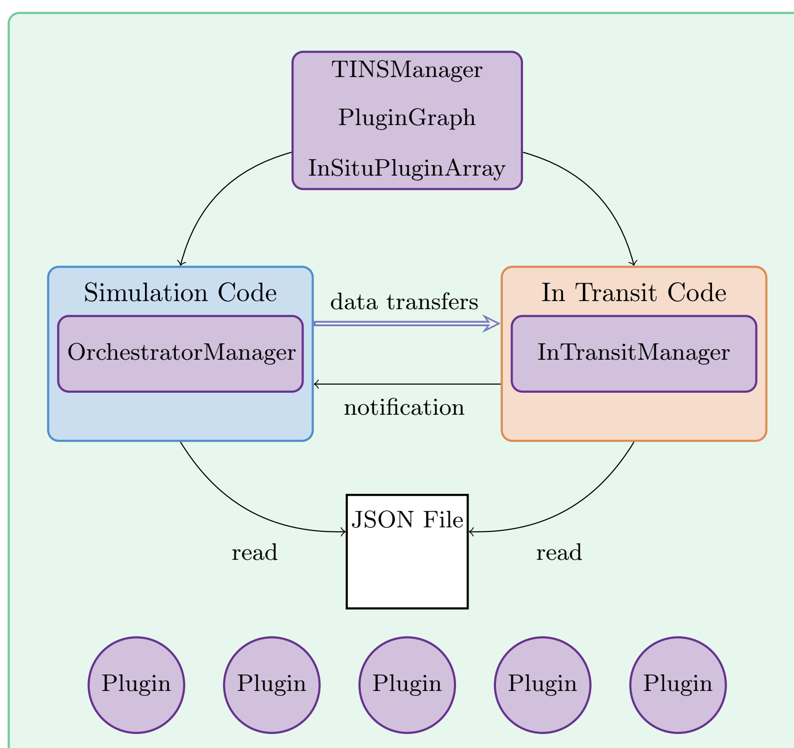

Figure 7.8 | Design of the in transit mode of TINS.

for the data transfers and we want to be able to execute in situ analytics concurrently with in transit processing.

The `OrchestratorManager` and the `InTransitManager` classes need respectively to implement the `sendData` and `recvData` methods to enable data transfer and to store the received data in the `ParticleInSitu` data structure of the `InTransitManager` object. These methods can be implemented thanks to existing in transit middleware such as Damaris [42], FlowVR [44] or Decaf [46] for example. This would indeed allow to take benefit from their work on data redistribution and data transfer. The strength of our framework is that the plugin implementation does not depend on the data transfer method. Indeed, data transfer is managed by the objects derivating from the `TINSManager` interface and data are retrieved thanks to the accessor methods that they provide, no matter how these objects get and store the data. The plugins are therefore exactly the same for an in situ or an in transit execution.

7.3.2 Implementation of a Prototype and Preliminary Results

We have seen in the previous section that the in transit mode of TINS is made possible by implementing the `sendData` and `recvData` methods to enable data transfer and to store received data in the `ParticleInSitu` data structure of the `InTransitManager` object. As already seen in Chapter 2, in transit analytics are usually executed on fewer nodes than the simulation and complex redistribution patterns must be computed to know how data should be sent from simulation to staging nodes. We want in this section to make a proof of concept of our in transit framework and to study the potential gains of the approach over an in situ mode alone. We do not want to deal with complex redistribution patterns and we therefore made the choice of implementing a

prototype with one staging node only. Future work will include the support of several staging nodes by implementing our own redistribution patterns or by using existing libraries such as Bredala [45].

In our prototype, data transfers between simulation and analytics nodes are performed thanks to the MPMD support of MPI. As already explained in Chapter 2, two codes launched in a MPMD world share the same `MPI_COMM_WORLD` communicator. During the simulation initialization, a MPI communicator is created for the simulation communications and the rank of the in transit MPI process is found by comparing the simulation and the `MPI_COMM_WORLD` communicators. Communications between the simulation nodes and the staging node are then performed in the `MPI_COMM_WORLD` communicator thanks to `MPI_Gather` communications performed by the simulation orchestrators and the `InTransitManager` instance in the staging node. The simulation nodes first send their number of particles thanks to `MPI_Gather` collective communications. This step is necessary because the number of particles per MPI process may vary over time. Necessary data are then serialized based on the JSON file that describes the input fields of the analytics. Data are then sent to the staging node thanks to `MPI_Gatherv` collective communications. The `InTransitManager` instance is finally in charge of unserializing the received data and filling its `ParticleInSitu` data structure with the received data. The shared JSON data file is used here in two ways. First, it enables the `InTransitManager` to know which data are received from the simulation, thanks to the dictionary defined for in situ processing. Secondly, the in transit code uses the shared JSON file to know the number of calls to `MPI_Gather` to perform in order to avoid deadlocks between simulation and staging nodes.

To test the in transit prototype, we compared the end-to-end execution times when executing the `radial` analytics in situ with the dynamic helper core strategy or in transit. We have chosen this analytics for two reasons. First, we have shown in Chapter 6 that the `radial` analytics has a significant impact on the simulation execution time because of its nested parallel loops. Executing it in transit may thus reduce the impact of the analytics on the simulation execution time. Secondly, we have explained in Chapter 4 that the `radial` analytics is actually a kernel extracted from the Radial Distribution Function (RDF) computation. The RDF computes an histogram of all the pairs of atoms in the entire simulation. Computing it requires a complex communication pattern so that every MPI process get access to the positions of the particles hold by all the other MPI processes. The `radial` analytics does not implement this communication pattern and does thus not compute the true RDF when executed in situ. On the contrary, this analytics does not require the communication pattern when it is executed in transit on one staging node because it has already access to the positions of all the particles. The in transit execution of the `radial` analytics returns thus the true RDF.

Table 7.1 reports the end-to-end execution times when executing the `radial` analytics in situ or in transit. We compare the in transit execution when ExaStamp is executed on 4 nodes and the analytics on one node (`intransit`), an in situ execution on 4 nodes (`insitu-4`) and an in situ execution on 5 nodes (`insitu-5`). For the three cases, the simulation corresponds to 256 iterations and the analytics is executed every 32 iterations. We observe that `intransit` reduces the end-to-end time of `insitu-4` by 11%, mostly because the in transit execution has a lower impact on the simulation execution time than the in situ execution. However, `intransit` uses 20% more resources than `insitu-4`. When comparing the execution time of `intransit` and `insitu-5` that uses the same amount of resources, we observe that `intransit` is this time 10% longer than `insitu-5`. However, we have explained earlier that `intransit` returns the

true RDF while a communication pattern is necessary for `insitu-5` to return the true RDF. Implementing such a communication pattern would have an important impact on the in situ execution time, while the in transit execution time would remain unchanged, reducing or even reversing the execution time difference. Another aspect also needs to be taken into account: the development time of the analytics. For an in transit execution, the development time is smaller than for an in situ execution because it does not require the design and implementation of an efficient communication pattern. The in transit mode added in TINS therefore allows to execute some complex analytics at a lower development cost than the in situ mode.

Table 7.1 | Comparison the in situ and in transit execution times of the `radial` analytics for a 256-iteration simulation of 16,000,000 particles on 4 or 5 nodes, analytics being performed every 32 iterations.

Mode	Total	Simulation	Graph execution (simulation side)
In transit (4 + 1 nodes)	428.82 s	428.32 s	0.706 s
In situ (4 nodes)	480.69 s	476.71 s	33.32 s
In situ (5 nodes)	389.11 s	386.26 s	21.26 s

Table 7.2 shows the end-to-end execution time of an in situ analytics workflow compared with the end-to-end execution time of the same in situ analytics workflow with the in transit execution of the `radial` analytics. The in situ workflow is executed every 4 iterations while the `radial` analytics is executed in transit every 32 iterations. We notice that the in transit node has a small impact on the in situ analytics execution time, the data transfer corresponding to 2 % of the in situ analytics execution time and requiring only one thread.

Table 7.2 | Comparison the in situ and hybrid execution times of an analytics workflow for a 256-iteration simulation of 16,000,000 particles on 4 nodes. The in situ workflow is executed every 4 iterations while the `radial` analytics is executed in transit every 32 iterations.

Mode	Total	Simulation	Graph execution (simulation side)
In situ	458.38 s	455.14 s	29.47 s
Hybrid	458.96 s	455.70 s	28.47 s (transfer 0.608 s)

The approach has still a limitation when the in transit analytics execution time is greater than then execution time between two analytics breakpoints. In this case, the orchestrator in the simulation side is ready to perform the `MPI_Gather` collective communications but the in transit code is still performing the analytics computation. The orchestrator is then blocked until the in transit code is ready to call the `MPI_Gather` function. To solve this issue, non-blocking communications or one-sided communications could be used to prevent the orchestrator to be blocked when the in transit analytics is not ready to receive data. More complex data management mechanisms can also be used to store several iterations of simulation data.

7.3.3 Execution of Analytics Plugins in a Standalone Mode

Developing new analytics and integrating them in an analytics workflow is a complex process that is generally done in several steps. Usually, the physicists develop analytics codes that are first tested and validated in a post-process way, the analytics codes reading data written by the simulation. It is only after having validated the analytics results and checked the coherence of

the analytics for a particular phenomenon that the analytics can be integrated into an analytics workflow executed in situ or in transit.

Based on this observation, we were convinced that a standalone mode was important to execute the analytics plugins outside of the simulation code. However, it is not possible as such because each plugin is compiled as a shared library and needs to use an instance of the `TINSManager` class to retrieve data for their computations. Conceptually speaking, a standalone mode is not really different than an in transit mode, except that data are not received from distant nodes but read from files. The developments already performed for the in transit mode were therefore reused to offer a standalone mode by simply adding in the `InTransitManager` class a `readData` method that reads data from a user-defined file and that stores the data in its `ParticleInSitu` data structure. Thanks to this reader, the plugins are exactly the same for in situ, in transit and standalone modes, no recompilations of the plugins being necessary to change the mode.

We therefore use the same in transit code for in transit processing and standalone execution of analytics. The JSON input file of the standalone mode is modified to tell the location of the files and to express the number of iterations to process. The standalone code instantiates the TBB scheduler to take benefit from the TBB parallelization of the plugins and of the graph capacities provided by TINS. The standalone mode can then be used by analytics developers as a testing platform to test the analytics execution, validate the results of the analytics and check the validity of the analytics inside their analytics workflow. The analytics workflows can also be tested in a standalone mode before being executed in situ and/or in transit with the simulation.

We have in this section extended TINS with an in transit capacity by using most of the tools already developed for the in situ mode. Data are transferred from the compute nodes to the staging nodes thanks to extra nodes added in the analytics graph. Data transfers can be implemented thanks to existing middleware and we developed a small prototype where we use the MPMD support of MPI to transfer data to one staging node. We have shown that the in transit mode allows to execute analytics that were not possible in an in situ context, with a low cost on the simulation execution. We have also extended this work to add a post-processing mode in TINS so that analytics can be executed transparently in situ, in transit or in a post-processing way.

7.4 Chapter Summary

We have presented in this chapter the design of the TINS framework, whose primary goal is to offer an intuitive and evolutive environment to develop analytics outside of the simulation code. The analytics are developed as TINS plugins, the data management and the synchronizations with the simulation being hidden by TINS. The analytics workflow is described externally by the user and automatically transformed into a TBB flow graph for an in situ execution. The framework of TINS can be extended to enable the in transit execution of analytics, by using existing middleware or by implementing our own data transfer. The strength of the TINS framework is that the analytics are developed as if they were post-processing codes but they can be executed in situ, in transit or in a post-processing way without any code modification or recompilation. In the next chapter, we will validate the robustness of the TINS framework and its capacity to execute complex analytics workflows in a production environment.

8 Validation of TINS on a Production Run

One of CEA field of studies is the propagation of shocks through metallic crystals. It can be used to study micro-jetting [48], micro-spallation [104] or phase transitions for example. We focus in this chapter in the phase transition of tin[1] material under shock. This kind of simulations requires a complex potential that usually limits the size of the systems that can be considered. Typical simulations of this phenomenon are usually performed on a few million particles but the understanding of the phenomenon requires one or two orders of magnitude more particles. ExaStamp and the Tera-1000-2 supercomputer allow to perform simulations of 300 millions atoms, leading to a better understanding of the physics at stake. In this chapter, we use the TINS framework implemented in ExaStamp to analyze in situ the data produced by a production run, greatly reducing the amount of data being stored to persistent storage and the end-to-end execution time of simulation and analytics. We first describe the physics at stake and the analytics workflow being considered (Section 8.1). We then insist on the limitations of the new Tera-1000-2 supercomputer and the adjustments that were performed to execute TINS on the supercomputer (Section 8.2). We conclude this chapter with a physical validation of TINS and preliminary performance measurements (Section 8.3).

8.1 Description of the Physics and the Analytics Workflow

When a shock is initiated, for example when a high-velocity projectile hits a target, a shock wave propagates inside the material. If the intensity of the shock remains below a certain threshold, the shock wave propagates until reaching the free surface where it is reflected and becomes a rarefaction wave. The propagation does not induce any deformation, the shock is called elastic. On the contrary, if the shock intensity is large enough, the shock wave is made of an elastic wave and a plastic wave, traveling at different velocities. A two-wave structure is therefore observed. The first wave always corresponds to an elastic shock wave but the second plastic wave is more difficult to identify. By analyzing the local structure of the matter after the second wave, it is possible to highlight structural changes that indicate a phase transition after the passage of the second wave. The analysis of the local structure of the matter is made thanks to one or several order parameters that reflect the local environment around each particle. We use here the Steinhardt parameters (Section 8.1.1) to study the structure of the matter and we create

[1] Notice that *tin* refers here to the chemical element while *TINS* refers to our task-based in situ method.

an analytics workflow that couples the computation of these parameters with several tools to determine the phase transition of the matter (Section 8.1.2).

8.1.1 Computation of the Steinhardt Parameters

The Steinhardt parameters [71] Q_n are used in material sciences to study the local structure of the matter. The parameters are computed for each particle based on the distances and angles between a particle and its closest neighbors. Because it studies the local structure of the matter, the analytics is applicable when the matter is sufficiently dense so that a particle has at least a dozen neighbors within its cutoff radius. By deriving quantities from these parameters, such as mean values or histograms on a given region and by comparing these quantities with reference cards, it is possible to determine the structure of the region.

For a particle i, the computation of the $Q_n(i)$ parameters consists in the projection of the vectors joining i and its $N_v(i)$ nearest neighbors on a sphere of unit radius. The Steinhardt parameters are defined by Equation 8.1.

$$Q_n(i) = \sqrt{\frac{4\pi}{2n+1} \sum_{m=-n}^{n} |q_{nm}(i)|^2} \tag{8.1}$$

The $q_{nm}(i)$ coefficients are expressed in Equation 8.2. They are computed based on the spherical harmonics Y_{lm} and the solid angle Ω_j.

$$q_{nm}(i) = \frac{1}{N_v(i)} \sum_{j=1}^{N_v(i)} Y_{lm}(\Omega_j) \tag{8.2}$$

Two parameters need to be chosen when computing the Steinhardt parameters: the value of n and the number of neighbors to consider for each particle $N_v(i)$. High values of n give more information about the structure but necessitate more computations. $N_v(i)$ is either the same for all the particles, corresponding to the k nearest neighbors of the particles, or dependent on i, corresponding to the particles inside a sphere of radius r_v, smaller than the cutoff radius, around the ith particle. In the following, we consider only the k nearest neighbors.

The algorithm to compute the Q_n parameters is decomposed into two steps. For all the particles i, we first need to find the k nearest neighbors based on the distance between the particle i and the other particles. When the neighbors have been found, the Q_n parameters can be computed for each particle based on the discretization of spherical harmonic differential equations [101]. A TBB version of this analytics has been developed at CEA during an internship supervised during this thesis, and we detail here the adjustments performed to execute it in situ with TINS.

Neighbor Search

The Q_n computation (hereafter named `qparam` analytics) aims at adding $n+1$ attributes to each particle. We can consider it as a local analytics without MPI communications given that TINS gives access to enough information. The analytics needs the position of the particles owned by the MPI process but also the neighbors of each particle. The neighbors may be inside the MPI process or belong to other MPI processes. As we have already seen in Chapter 3, ExaStamp keeps a layer of ghost cells that correspond to the cells hold by other `Domains`. We made this ghost layer available through TINS so that the `qparam` analytics does not need to perform MPI

communications. Here we consider $k \leq 12$ and the k nearest neighbors can be found within the ghost layers because the `Cells` in ExaStamp are constructed to have a size slightly greater than the cutoff radius.

A naive approach is to search the k nearest neighbors by iterating on every particles hold by the MPI process and on every ghost particles. However, this approach leads to execution times orders of magnitude longer than the execution time of an ExaStamp iteration (Table 8.1). This is all the more true for high numbers of particles. ExaStamp being usually used with several millions particles per MPI process, an acceleration data structure has to be used to reduce the complexity of the neighbor search.

Table 8.1 | Comparison of the time to find the neighbors and the time to compute an iteration of ExaStamp for different numbers of particles.

Number of particles	Neighbor search	ExaStamp iteration
16,000	0.083 s	0.004 s
250,000	1.950 s	0.046 s
2,000,000	1,220.0 s	0.301 s

During the data copy, ExaStamp copies the particles attributes into the `ParticleInSitu` structure of arrays. The analytics therefore retrieve the particles attributes as arrays of length the number of particles inside the MPI process. However, ExaStamp internally implements a cell structure and the arrays are filled according to the cell structure, as highlighted in Figure 8.1 (left). In the arrays, the cells are represented by contiguous indices. Knowing the array index where a particular cell begins and the number of particles in the cell, it is possible to determine where the particles of a given cell are located in the arrays.

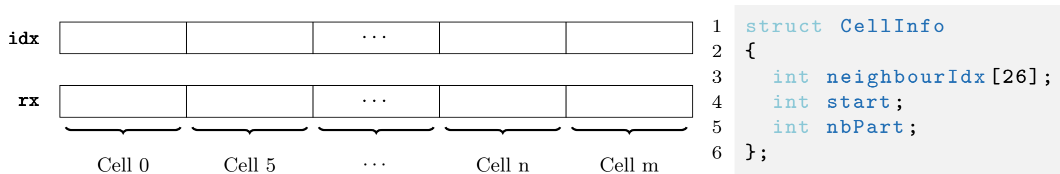

Figure 8.1 | Cell structure inside the arrays of particles attributes (left) and `CellInfo` structure to easily retrieve cell information in TINS plugins (right).

The neighbor search can be speed up by using the cell information. Instead of iterating through the pairs of particles, we perform a loop on the cells. For each cell `c`, we look for the $N_v(i)$ nearest neighbors of the ith particle inside the cell `c` and inside the 26 neighboring cells of `c`. To do so, we introduce the `CellInfo` structure described in Figure 8.1 (right) where the 26 neighboring cells indices, the starting index and the number of particles of the cells are stored by ExaStamp. With this technique, the neighbor search is reduced by 4 orders of magnitude for 2,000,000 particles.

TBB Parallelization

To take benefit from TINS capabilities and in particular of the dynamic helper core strategy proposed by the framework, the `qparam` analytics has been parallelized with TBB. The analytics is decomposed in two functions, one for the neighbor search and one for the Q_n computation. The neighbor search function is based on a `parallel_for` loop on the `CellInfo` array provided by TINS. A task corresponds to the neighbor search of the particles inside one or more cells. The Q_n computation on the other hand does not need the cell information and corresponds to a `parallel_for` loop on the particles. For each particle i, the distances and angles between i and its nearest neighbors are computed and used to determine the spherical harmonics components. The performance of the `qparam` analytics are sketched in Figure 8.2. The analytics shows a good scaling on up to 64 threads, reaching 58 on 64 threads. The analytics is mainly compute intensive and does not take benefit from hyperthreading. The `qparam` analytics is an efficient parallel analytics that we will be able to use in TINS to be efficiently interleaved with the simulation execution.

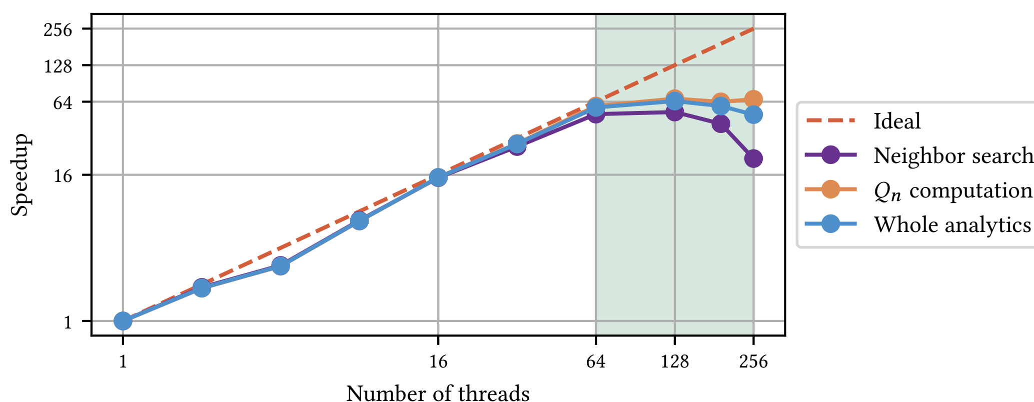

Figure 8.2 | Strong scaling of the `qparam` analytics on a KNL node for 2,000,000 particles. The cells information have been provided to speed the neighbor search. The green area highlights the number of threads from which we pass into the hyperthreading area.

Execution Times

Figure 8.3 shows the execution time of an ExaStamp iteration compared with the execution times of data copy, neighbor search and Q_n computation for a perfect crystalline structure. The computational cost of the analytics is linear with the number of particles and follows the same trend than the computational cost of ExaStamp. The neighbor search and Q_n computation parts have similar execution times and the whole analytics has an execution time of the same order of magnitude than ExaStamp. The execution time and the good scaling of the analytics therefore make it a good candidate to be used in an in situ context with TINS.

The copy time is here more significant than the copy times discussed in Chapter 7. This comes from the fact that the `qparam` analytics requires the cells information to speed up the neighbor search. For a simulation of 2,000,000 particles, it requires the index and positions of approximately 200,000 ghost particles and 300,000 cells, leading to 40MB of additional data compared to the 112MB of data necessary to copy the particle attributes only. This motivates our

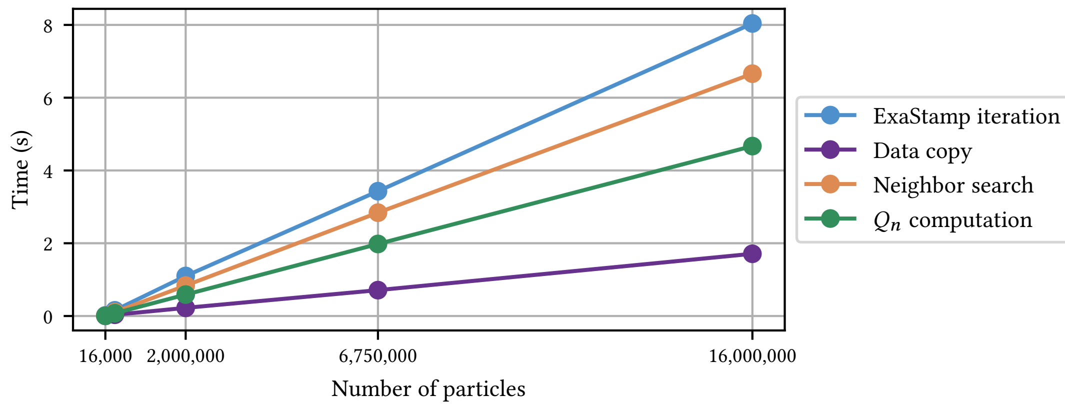

Figure 8.3 | Comparison of the time to perform an ExaStamp iteration with the times to copy data, perform the neighbor search and compute the Q_n parameters ($n = 8$ here) for different numbers of particles. Simulation performed on one node with one MPI process and 64 cores per MPI process.

work to copy only necessary data to avoid the expensive copy of data that may not be used by all the analytics, as we have already discussed in Chapter 7.

8.1.2 Definition of the Analytics Workflow

The analytics workflow is composed of 5 plugins and the analytics graph is sketched in Figure 8.4. We first compute the Q_n parameters for all the particles. We choose here $n = 10$ and $k = 8$. Four plugins can be executed after the Q_n parameters computation. Extra dependencies between plugins performing MPI communications are automatically added as explained in Chapter 7.

VTK File Output

A VTK file is written by each process for later being visualized with ParaView. The file output is sequential and relatively long because it writes the positions, velocities and the 11 Q_n parameters per particle. It is well suited for being executed asynchronously with the other plugins.

Mean Computation of Q_n

For each Q_n parameter, the mean value is computed on the global domain. It uses TBB for the local mean computation and a MPI reduction for the global mean computation. This analytics is used to see global changes in the structure of the matter. By looking at the mean values along time, it is possible to determine whether the structure is evolving and how fast.

Slice Histogram

For a shock propagating along one direction (here, the x-axis), the domain is split along the x-axis into N bins of equal size. Here, we choose $N = 1{,}000$. The mean of the velocities along the three axes and the mean of the Q_n parameters are computed for each bin, locally thanks to a TBB reduction and globally thanks to a MPI reduction. This analytics allows to have a better insight in the simulation data than the mean computation. In particular, it is possible to see the two-wave structure, to follow the propagation of the shocks and rarefaction waves and to see the structural changes along the waves propagation.

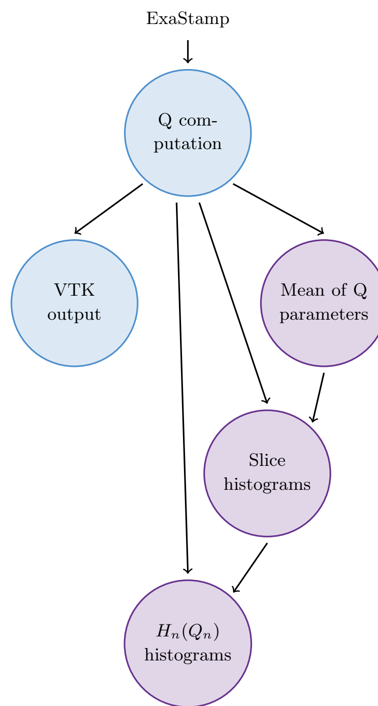

Figure 8.4 | Analytics graph for the study of a shock propagation into a tin material. The purple nodes correspond to plugins that perform MPI communications.

$H_n(Q_n)$ Histograms

For this analytics, the domain is also split along the x-axis into N_x bins of equal sizes. Inside each bin, an histogram is computed for each Q_n parameter on N_q bins. Here we use $N_x = 200$ and $N_q = 100$. This analytics has a similar behavior than the slice histogram analytics (one TBB parallel reduction and one MPI reduction) but it gives more information for the Q_n parameters than just the mean. The $H_n(Q_n)$ histograms have different behaviors. For an ideal crystal structure at equilibrium, all the particles have the same Q_n parameters and each $H_n(Q_n)$ histogram is a Dirac on the Q_n value. The Q_n parameters being affected by the temperature, their histograms have more often a gaussian shape that reflects the gaussian distribution of the particles around their equilibrium value. When several structures are present in the region, the $H_n(Q_n)$ histograms are more complicated and show several peaks. By looking at the Q_n values corresponding to the peaks and by comparing them with reference histograms obtained for different structures, the physicists can deduce which phases are present in the region. The proportion of each structure is then computed thanks to a linear combination of the reference histograms that match the different structures.

For the last three analytics, the results are written into text files that physicists can later process thanks to Python scripts for example. This is not a major performance bottleneck though because the histogram files correspond to a few MB of memory only while VTK files require more than 50GB of memory per iteration.

The complete study of the behavior of tin under shock requires a simulation of 300,000,000 particles on more than 1,000,000 iterations, taking appropriately 1 month with ExaStamp on the Tera-1000-2 supercomputer. The analytics workflow presented in this section is therefore intended to be used on the Tera-1000-2 supercomputer. However, the Tera-1000-2 supercomputer is a new supercomputer released in early 2018 and it is still under development. We will see in the following section the limitations of the supercomputer and the adjustments that were made to execute TINS.

8.2 Limitations of the Tera-1000-2 Supercomputer

The Tera-1000-2 supercomputer is a KNL-based supercomputer that relies on the new BXI interconnect. There are mostly two limitations for executing TINS on this supercomputer: the disabling of the OS scheduler on the KNL nodes (Section 8.2.1) and the temporary absence of the `MPI_THREAD_MULTIPLE` threading support (Section 8.2.2).

8.2.1 Disabling of the OS Scheduler on the KNL Nodes

As seen in Chapter 3, the KNL nodes of Tera-1000-2 are composed of 68 physical cores but 4 cores are exclusively dedicated to the system. When isolating the system cores, the system administrators have disabled the OS scheduler on the 64 cores allocated to the applications. TBB relying exclusively on the OS scheduler to map the threads on the cores, we adapted TINS so that the threads are bound to the cores.

Usually, TINS relies on the TBB scheduler to map the threads on the available cores in a way that optimizes the cache affinity. The only configurations where we gave information to the TBB scheduler for the thread mapping was for the static helper core strategy. In this case, we gave a mask to the threads entering the arenas so that the analytics were executed on the first NUMA nodes and the simulation on the last NUMA nodes, as we have seen in Chapter 5. Even in this configuration, we only gave masks to the TBB scheduler and the scheduler was in charge of mapping the threads according to the masks.

The TBB scheduler exclusively relying on the OS scheduler and the OS scheduler being disabled in the Tera-1000-2 KNL nodes, we have to help the TBB scheduler to map the threads to the cores, by assigning to each core a single thread. We do so by using the TBB observer class introduced in Chapter 3. The observer holds an array with the indices of the cores that can be used by the application. Every time a thread is scheduled for the first time by the TBB scheduler, it enters the `on_scheduler_entry` method and the observer binds it to the first core that has not already been assigned to a thread.

8.2.2 Temporary Absence of the `MPI_THREAD_MULTIPLE` Threading Level

The version of MPI suitable for the BXI interconnect, Bull MPI, is still under development and does not support `MPI_THREAD_MULTIPLE` yet. The absence of this threading level is only temporary and we are in contact with Bull to test future versions of Bull MPI that will include it. However, as explained in Chapter 4, TINS relies on `MPI_THREAD_MULTIPLE` because MPI communications are performed in tasks that can be executed concurrently with MPI communications performed by the simulation master thread. This threading support being unavailable at the time

this thesis has been written, we first thought of modifying TINS to support a less constraining threading level, namely `MPI_THREAD_SERIALIZED`.

In the `MPI_THREAD_SERIALIZED` threading level, several threads can perform MPI communications but the developer has to guarantee that the communications are not performed simultaneously. The MPI communications have thus to be serialized. To do so, we decided to externalize the communicator manager of ExaStamp presented in Chapter 3. Inside a `Node`, the simulation and the plugins share the same communication manager and every MPI communication is performed thanks to this object. A naive implementation has been to use a mutex to guarantee that only one thread performs MPI communications at a time but it leads to the same difficulty than the one explained in Chapter 7. A thread in one MPI process may take the mutex for a simulation MPI communication while a thread in another MPI process may take it for an analytics MPI communication, leading to a deadlock. More advanced thread management being still under development at the time this thesis has been written, we decided to restrain our use of TINS on synchronous executions for the whole analytics workflow of Figure 8.4 and asynchronous executions of analytics workflows composed of plugins that do not perform MPI communications. In the following section, we will perform a numerical validation of TINS and present preliminary performance measurements on the Tera-1000-2 supercomputer under these restrictions.

8.3 Validation of TINS

The primary goal of TINS is to allow the physicists to study physical phenomena with an end-to-end execution time as small as possible. The validation of TINS is performed in two steps. First, we perform a numerical validation to check whether the physical phenomena at stake are well reproduced when executing the analytics workflow in situ (Section 8.3.1). We then give preliminary results concerning the performance of TINS on the Tera-1000-2 supercomputer (Section 8.3.2). We finally present the possible gain of the TINS framework for the physicists (Section 8.3.3).

8.3.1 Numerical Validation

To validate numerically the analytics and the execution of TINS, we first computed the Q_n parameters, the slice histograms and the $H_n(Q_n)$ histograms during a few iterations, based on checkpoint files produced by ExaStamp. The simulation corresponds to a bar of tin material (Figure 8.5) with a shock propagating along the x-axis. In Figure 8.6, we show the slice histogram of the x-velocity and of the Q_6 parameter for three different iterations. We see the two-wave structure (left), the reflection of the shock wave on the free surface (middle) and the moment where the rarefaction wave crosses the phase transition wave (right). The phase transition is highlighted by the slice histogram of Q_6. The values of Q_6 are fluctuating after the passage of the phase transition wave while it is more stable before.

Figure 8.7 highlights the structural changes by showing the $H_6(Q_6)$ histograms in three different regions. The regions before the passage of the phase transition wave (green and purple) present gaussian $H_6(Q_6)$ histograms. Each region is composed of one phase, the phases being slightly different due to the passage of the shock wave. The region after the passage of the phase transition wave (orange) exhibits a two-wave structure typical of the cohabitation of several phases inside the region. This is also highlighted by the different colors in Figure 8.5.

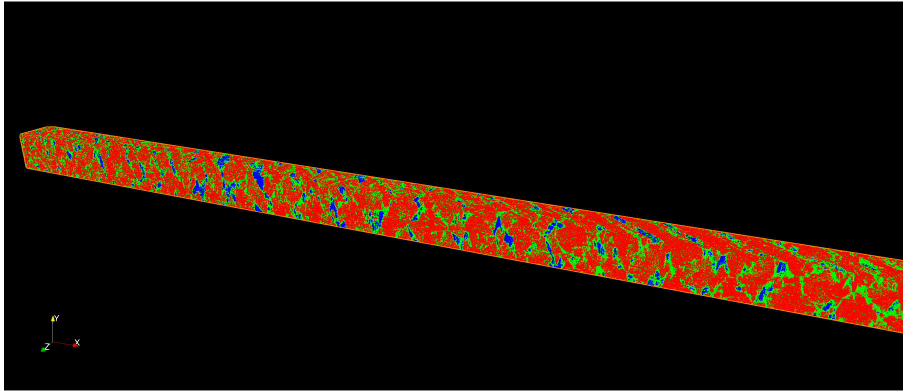

Figure 8.5 | Visualization of the Q_6 parameter in a region after the passage of the phase transition wave. The different colors highlight the coexistence of different phases in this region. This visualization is made possible thanks to the VTK file output plugin.

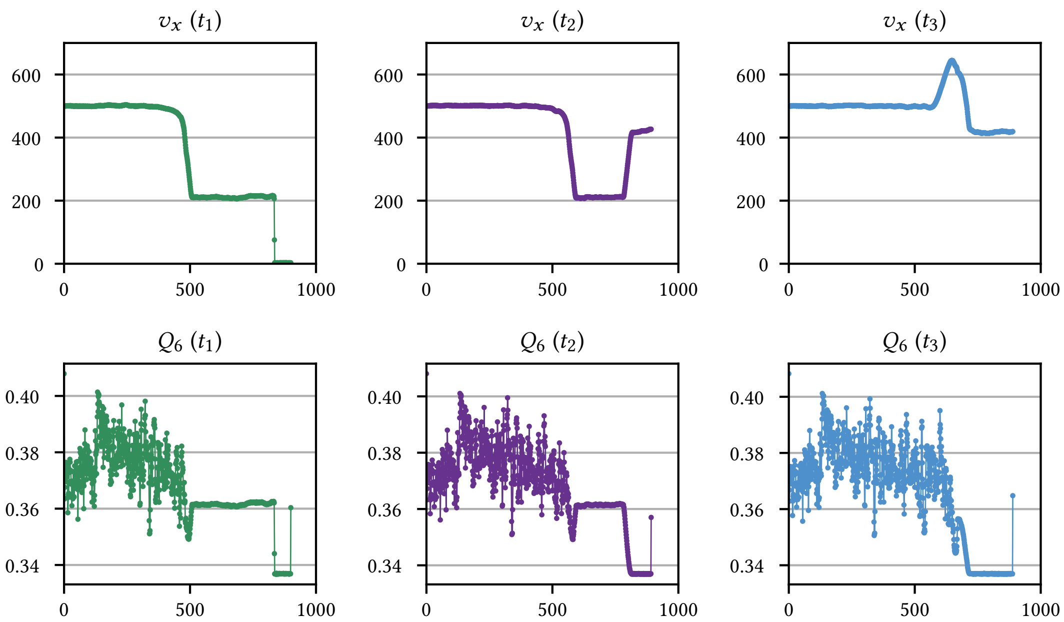

Figure 8.6 | Slice histograms of vx and Q_6 for three different iterations. The first iteration shows the two-wave structure with a shock wave and a phase transition wave (left). The shock wave is then reflected at the free surface (middle) and the third iteration corresponds to a time after the rarefaction wave has crossed the phase transition wave (right). The fluctuations in the Q_6 slice histogram highlight the coexistence of several phases. The histograms are sketched thanks to the files output by the slice histograms plugin.

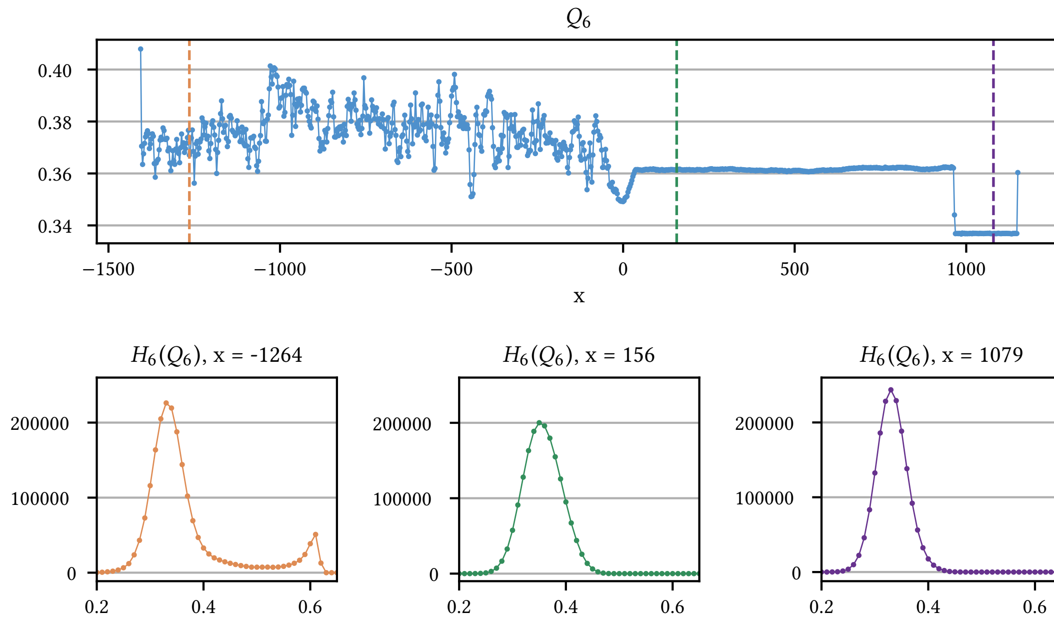

Figure 8.7 | $H_6(Q_6)$ histograms of three different regions along the wave propagation. The histograms are sketched thanks to the files output by the slice histograms and the $H_n(Q_n)$ histogram plugins.

The data produced by the analytics workflow executed with TINS are in adequation with the physical phenomena at stake and with the data produced by other tools developed at CEA. This validates numerically the analytics and the TINS execution.

8.3.2 Preliminary Performance Measurements

As already explained in Section 8.2, the Tera-1000-2 supercomputer does not provide a `MPI_THREAD_MULTIPLE` support, necessary for the asynchronous execution with TINS of analytics workflows where analytics perform MPI communications. The asynchronous execution of the analytics workflow presented in Figure 8.4 has been tested on small systems on other supercomputers, including the Tera-1000-1 supercomputer hosted by CEA but the large-scale simulation of tin material under shock is only possible on the Tera-1000-2 supercomputer. We therefore present in this section preliminary results of the synchronous execution of the complete analytics workflow with TINS and the asynchronous execution of smaller analytics workflows where the plugins do not perform MPI communications.

Synchronous Execution of the Complete Analytics Workflow

The production run simulates a tin material composed of 300,000,000 particles during 100,000 iterations and using 256 KNL nodes (16,384 cores). The simulation lasts for three days and is composed of checkpoint files written every 10,000 iterations and load balancing with the Zoltan library [24] every 5,000 iterations. The load balancing is necessary because of the heterogeneity of the simulation and induces a change in the number of particles owned by every MPI processes.

TINS is used to execute the analytics workflow in a synchronous way. All the analytics are executed every 500 iterations, except the VTK file output that is performed every 2,000 iterations. The first observation is that TINS proves to be a robust method that manages to execute the

analytics workflow during several days. In particular, its data management layout is efficient to handle the change in the number of particles owned by the different MPI processes.

Table 8.2 summarizes the end-to-end execution time of the execution with TINS of the analytics workflow in situ with ExaStamp. The copy time is negligible compared to the simulation time because the mean number of particles per MPI process is around 1 million. The execution time of the analytics graph corresponds to 9% of the total execution time. To see the benefits of the approach, we compared the execution times of ExaStamp/TINS and the current analytics workflow used by the physicists at CEA (Table 8.3). Periodically, MPI-IO files are output by ExaStamp to be read by a custom analytics tools developed at CEA. Writing an MPI-IO file requires roughly 120s, reading it corresponds to 540s and the Q_n computation needs an extra 600s because the analytics code does not have access to the cells information provided by ExaStamp and is only parallelized with MPI. Analyzing one iteration with 300,000,000 particles therefore accounts for 1,260s and an MPI-IO file of 52GB stored into the filesystem, while the in situ computation with the `qparam` analytics and the TINS framework only requires a few seconds and no data stored into the filesystem. The gains are expected to be even more important with an asynchronous execution of the analytics workflow.

Table 8.2 | Execution times of the analytics workflow executed in situ with a simulation of 100,000 iterations.

	Total	Simulation	Copy	Analytics
Time (s)	301,150	275,650	8	25,830

Table 8.3 | Execution time of one iteration of the `qparam` analytics executed in situ compared to a custom post-processing tool developed at CEA. I/O time corresponds to the time to write and read a MPI-IO file with the particles information.

	I/O	Analytics
`qparam` in situ	0 s	2.5 s
custom post-processing tool	660 s	600 s

Even in the synchronous mode, there is an interest in using the TINS framework for analyzing simulation data. In particular, TINS provides a graph capacity to transform the analytics workflow into a graph where several analytics can be executed concurrently. In the analytics workflow used for this study, the graph feature allows to write the VTK file concurrently with the other plugins.

Asynchronous Execution of the `qparam` analytics

To measure the gain of the asynchronous execution of an analytics workflow compared to the synchronous execution, we constructed an analytics workflow composed only of the `qparam` analytics. This analytics does indeed not perform MPI communications and can be executed asynchronously with TINS on the Tera-1000-2 supercomputer. The asynchronous execution of the analytics is performed with the dynamic helper core strategy with an analytics arena of size n_a. For the simulation of the behavior of tin material under shock, the physicists usually compute the Q_n parameters every 500 iterations. In this case, the gain between a synchronous and an asynchronous execution is of 6.75%. We tested different analytics arena sizes but the best

configuration is when $n_a = n_s = 64$ to take benefit from the analytics parallelization. For this analytics, the adaptive dynamic helper core approach does not reduce the end-to-end execution time because the best configuration is when $n_a = n_s$.

The gain of the asynchronous approach is all the more important as the frequency of the analytics increases, as highlighted in Table 8.4. In an extreme case, when the analytics is executed after each iteration, the asynchronous execution of the `qparam` analytics is 22% faster than the synchronous execution. This trend will also be true for the complete analytics workflow because the other plugins mainly perform MPI communications that have been proved in Chapter 6 to be well interleaved by the dynamic helper core strategy.

Table 8.4 | Gain of the asynchronous execution of the `qparam` analytics compared to a synchronous execution of the analytics for different frequencies (the frequency increases from left to right).

	500	16	4	1
Gain (%)	6.75	7.08	10.07	22.47

TINS and the dynamic helper core strategy allow in situ data analytics at a higher frequency than the synchronous approach with a small overhead on the simulation execution time. The approach can thus be used to significantly reduce the end-to-end time of complex analytics workflows.

Limitations of the Graph System

The execution of TINS for this production run let us envision two limitations of the graph system. The first limitation is related to the serialization of the analytics node that perform MPI communications. To prevent the potential deadlocks that may occur when executing a graph composed of analytics that perform MPI communications, we added a flag in the JSON file so that the plugins that define this flag are serialized. However, this limits the asynchronous aspect of the graph when most of the plugins perform MPI communications, as it is highlighted in Figure 8.4. A solution to this limitation would be to split the `IP_run` method into several methods: some methods with local computations without MPI communications and some other composed of MPI communications or any other global communications that require a synchronization from several MPI processes. The graph could be modified so that the analytics is decomposed into several dependent nodes. We highlight this in Figure 8.8, where the nodes are decomposed into two nodes, first a local one and then a global one. This way, only the global nodes would need to be serialized, which increases the asynchronous aspect of the graph. Most advanced features could also support the decomposition into more than two nodes to execute more complex analytics.

The second limitation comes from the unique graph where all the nodes are not executed at the same frequency. In the example of Figure 8.4, all the nodes are executed every 500 iterations, except the VTK file output that is performed every 2,000 iterations. The reason behind this choice is that the VTK file output execution time corresponds to more than 1,000 iterations of ExaStamp. By executing it every 2,000 iterations, we hope to overlap the execution time of the plugin with the simulation. However, our graph system shows a limitation in this case. At the 2,000th iteration, the orchestrator launches the graph execution on data copied by the simulation and sets the next analytics breakpoint at iteration 2,500. However, when the simulation reaches the 2,500th iteration, it cannot copy its data because the VTK file output has not completed yet.

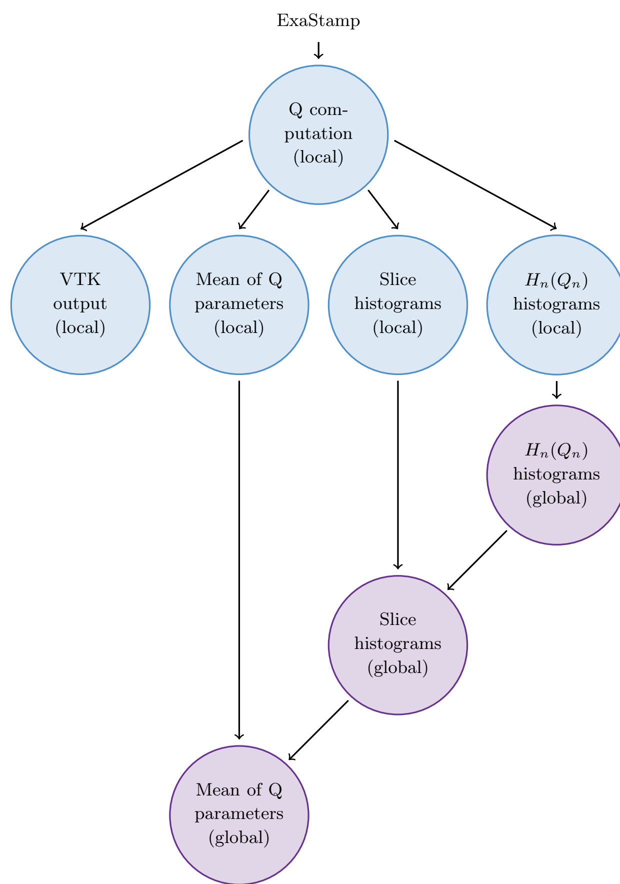

Figure 8.8 | Modification of the graph of Figure 8.4 where the analytics are decomposed into two methods, a local one without MPI communications and a global one with MPI communications. The serialization is necessary only between the global nodes.

A solution to this issue is to create several graphs depending on the analytics frequency. In the example of Figure 8.4, the graph would be decomposed into two graphs, one without the VTK file output node and executed every 500 iterations and one consisting in the whole graph executed every 2,000 iterations. These two graphs coupled with a mechanism that stores several copies of the simulation data would reduce the waiting time in the synchronization point and would allow to execute heavy analytics at a low frequency without disturbing the execution of lightweight analytics at a higher frequency.

8.3.3 Gain of TINS for the Physicists

In situ processing has been recognized in the computer science field as a mean to significantly reduce the end-to-end execution time of complex analytics workflows compared to traditional post-processing tools. However, in situ processing is still under-used by the physicists who see this approach as more restrictive than the post-processing approach. During this thesis, we conducted a survey on the data analytics habits of the physicists at CEA. Several reasons were given to explain why the physicists still prefer post-processing tools. First, the major reason is that the analytics workflow must be known in advance for in situ processing while this is usually not the case because it is mostly constructed and refined thanks to an exploratory process based on data stored by the simulation. Secondly, the physicists fear that in situ processing will totally remove file output and that they will need to re-execute long simulations if they want to execute analytics that were not initially planned. Finally, physicists prefer to use synchronous analytics integrated into the simulation codes instead of complex architectures that would require them to follow specific trainings to master these tools.

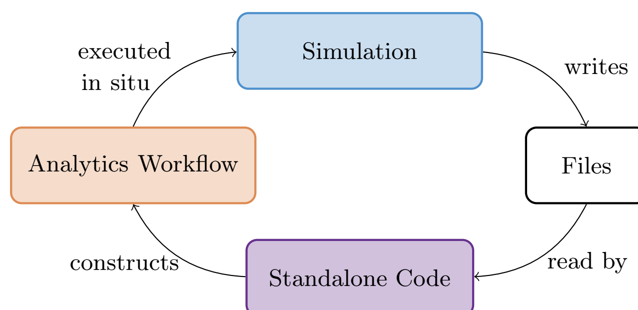

Figure 8.9 | Iterative construction of an analytics workflow with TINS.

With TINS, we provide an intuitive environment integrated inside the simulation to offer the physicists the ability to create and execute complex analytics workflows in situ or in transit. TINS only relies on TBB and does not add extra dependencies in the simulation, greatly reducing the learning curve to use the framework. Moreover, TINS is not intended to replace the file output but rather to complement it, as it is highlighted in Figure 8.9. The framework proposes a standalone mode that can be used to develop analytics that explore data saved by the simulation and to construct complex analytics workflows as if they were post-processing codes. The analytics workflows can then be executed in situ and/or in transit without any modifications in the analytics codes. Data can still be output, potentially at a lower frequency, so that the physicists can use them to refine their analytics workflows. This way, TINS can be used to iteratively construct analytics workflows and reduce the need of file output. The analytics workflows can

then be used transparently in a production environment, hence reducing the end-to-end time to scientific discovery.

8.4 Chapter Summary

We have seen in this chapter that TINS allows the construction and the execution of complex analytics workflows in a production environment. TINS is a robust and intuitive tool where the physicists can easily develop and test their analytics workflows before being executed in a production environment. TINS shows promising results on the asynchronous execution of analytics at a high frequency. Due to technical limitations of the target production supercomputer, we were not able to execute TINS at its full capacities, that is to say with the dynamic helper core strategy. We still show that the synchronous execution of a complete analytics workflow is performed with an overhead of 9% over the execution time of ExaStamp. We have also shown that TINS present promising results for the asynchronous execution of analytics at a high frequency.

9 Conclusion and Perspectives

Traditionally, the analysis of simulation data is performed in a post-processing step. Simulation data are periodically written into the filesystem by the simulation and read back by analytics applications to extract information about the physics at stake. This post-processing of data will become more and more difficult because of the growing gap between data generation rate and the time to write and read data to and from the filesystem, which calls for new data processing methods. The in situ paradigm proposes to reduce the need to write data by directly analyzing them while resident in the compute node memory. Several techniques exist, either by executing simulation and analytics on the same nodes (in situ), by dedicating a set of nodes to the analytics (in transit) or by using a combination of the two (hybrid). Many works have focused on implementing in situ middleware that optimize the resource usage of the nodes. They generally execute simulation and analytics as different processes, either by exploiting the unused resources of the simulation to execute analytics processes or by dedicating cores, called static helper cores, to the analytics. While these techniques have proved their performance for multicore architectures, they do not target manycore processors nor simulation codes optimized for these architectures. In particular, task-based programming models are expected to become a standard for manycore architectures and would allow to finely optimize the execution of in situ analytics but few in situ techniques have been developed with this emergent programming model in mind.

9.1 Contributions

In this thesis, we studied the design and integration of a novel task-based in situ framework inside a task-based molecular dynamics code designed for exascale supercomputers. Our target code is ExaStamp [29], a molecular dynamics code developed at CEA for the last 5 years. ExaStamp shows quasi-linear speedup on multi and manycore processors, but the fork-join model it employs induces small sequential regions that reduce the performance of the simulation code. Such simulation cannot take the best of existing in situ techniques, either because the sequential regions are too short to be exploited without degrading the simulation performance or because confiscating cores has a significant impact on simulation performance. To solve the issues of traditional in situ techniques on such code, we proposed to leverage the task-based programming model and the work stealing concept to create and execute analytics tasks concurrently with simulation tasks and with a low overhead on simulation execution time.

During this thesis, we introduced TINS, a Task-based IN Situ framework that relies on a task-based programming model to create simulation and analytics tasks concurrently and on a work stealing scheduler to efficiently interleave them in situ. Our implementation of TINS uses Intel® TBB [5] as the work stealing scheduler and is integrated into ExaStamp. In a first version, the simulation spawns an analytics task at a given frequency and resumes to the next iteration without waiting for the task completion. Simulation and analytics are therefore executed asynchronously thanks to the TBB scheduler. This approach proved to be up to 39% faster than a synchronous execution and up to 12% faster than the Goldrush middleware [119].

The comparison with the Damaris middleware [42] motivated the implementation of a thread isolation mechanism in TINS to add a static helper core feature in our framework. It relies on a dedicated thread, the orchestrator thread, and on an arena system that guarantees the strict separation of the threads into two disjoint groups. The simulation is just in charge of creating simulation tasks in a simulation arena and to copy data into a temporary buffer at a given frequency. The orchestrator synchronizes with the simulation master thread to launch analytics execution in the analytics arena on the data copied by the simulation. We have shown that the static helper core strategy implemented in TINS is equivalent or outperforms Damaris for analytics with important NUMA effects and we have shown that TINS is up to 42% faster with the thread isolation than without.

To reduce the impact of the choice of the number of static helper cores, we designed a dynamic helper core strategy with a temporary thread isolation. By extending the arena system introduced for the static helper core approach, the threads are split into two groups when both simulation and analytics tasks exist concurrently and steal work inside their own group. When the simulation enters a sequential region or when the analytics of an iteration is completed, the threads involved in these computations can move to the other group and steal tasks of the other group, hence reducing the thread idleness periods. This technique proved to be less sensitive to the number of helper cores than the static helper core strategy. We have shown that the dynamic helper core strategy implemented in TINS can be up to 40% faster than the static helper core strategy and Damaris. Simulations on up to 14,336 cores have shown that TINS with dynamic helper core executes dynamic analytics workflows with an overhead of less than 7% over ExaStamp alone. These results have led to the publication of a paper in an international conference [37].

TINS is not only a task-based in situ method but also an intuitive and evolutive framework where simulation and analytics codes are decoupled. Analytics are coded outside of the simulation code as if they were post-processing codes. The data management and synchronizations with the simulation are hidden by TINS and a standalone mode allows to test the analytics without executing the whole simulation. The end-user can easily describe analytics workflows that are transformed by TINS in TBB flow graphs and that can be executed in situ with the dynamic helper core strategy, in transit or with a combination of the two without any code modifications. We have in particular shown that executing analytics in transit adds a negligible overhead on the in situ execution of other analytics and that the in transit mode allows to execute analytics that were not possible in an in situ mode at a low development cost.

TINS has been tested and validated on a production run, proving its robustness on runs of several days with periodic load balancing. Due to technical limitations of the target supercomputer, we were not able to execute TINS with its full capacities but the synchronous execution of a complex analytics workflow with TINS was made with an overhead of 9% over ExaStamp execution time alone. Moreover, the in situ computation of a molecular dynamics parameter was

performed in a few seconds while post-processing tools usually used by the physicists require more than 10 minutes. Preliminary measurements on the asynchronous execution of a simpler analytics workflow show promising results and makes us confident in the capacity of TINS to efficiently execute complex analytics workflows at a high frequency and in a production environment.

9.2 Perspectives

TINS has shown its robustness in a production environment, its ease of use by non-expert developers and its performance compared to existing middleware but it also opens new perspectives and challenges.

Adding Support to Other Task-Based Runtimes Our implementation of TINS relies on the TBB library that provides a task-based programming model and a work stealing scheduler to interleave simulation and analytics tasks. This work aims at being generalized to support other task-based runtimes. We think in particular of OpenMP because a great number of simulation codes are developed using a MPI+OpenMP programming model. Recent versions of OpenMP support task-based programming but one difficulty will be to find an equivalent of the TBB arenas. The approach could also be generalized to distributed task-based programming models such as Legion [20] or HPX [65] to reduce the limitations of our graph system when analytics perform MPI communications. Finally, the supercomputing environments are becoming more and more heterogeneous and CPUs are now more and more often mixed with accelerators that provide more potential for in situ processing [59, 54]. In this context, it could be interesting to extend this work for heterogeneous computing thanks to StarPU [15] for example or by using the extended flow graph API of TBB to execute tasks on accelerators such as GPUs for example.

Reducing Idleness Periods of the Orchestrator Thread In TINS, an orchestrator thread is spawned during the initialization of the simulation. The simulation master thread and the orchestrator thread have their own timeloop, both being in charge of creating tasks in their corresponding arenas. To avoid core over-subscription, we removed one thread from the worker thread pool. Because the orchestrator thread is not allowed to execute simulation tasks, there will therefore be at most $N-1$ threads executing simulation tasks on a processor with N cores, even if the orchestrator thread has no analytics tasks to execute. To reduce the orchestrator thread idleness periods, the idea would be either to add a mechanism so that the orchestrator can execute simulation tasks instead of being idle or to add more work to the orchestrator to benefit from the presence of this extra thread.

Steering of Analytics One example of work that can be added to the orchestrator thread is the steering of analytics. The computational steering traditionally consists in modifying the simulation parameters at runtime, to interactively test the influence of a given parameter for example. This feature has been used in the past but was abandoned because it prevented code reproducibility. The idea behind the integration of steering capacities in TINS would be to execute small analytics on the parameters of the simulation, to modify the frequency of the analytics or to add new analytics in the analytics workflow at runtime. The end-user could therefore modify the analytics workflow at runtime based on the observation of simulation parameters

and without impacting the simulation code reproducibility. A prototype has been developed in a project with our partner Paratools. It embeds a Python shell managed by the orchestrator thread. Knowing the data structure used by the orchestrator, it is possible to define Python commands that retrieve simulation data and apply small computations on them. The prototype is still under development and should also include the possibility to modify the analytics workflow at runtime. The difficulty here will be to add or remove analytics that perform blocking operations such as MPI communications. It will thus be necessary to implement deterministic algorithms so that each MPI process inserts or retrieves analytics nodes at the same position in the graph to avoid deadlocks.

Implementing More Advanced Graph and Data Management The analytics workflow is described as a TBB flow graph where a node corresponds to the computation of an analytics at a given frequency. At an in situ iteration, the whole graph is executed based on the copy of the simulation data kept in memory. If the frequency of an analytics in the graph does not match the in situ iteration, this analytics is not executed. We envision two enhancements of the graph management. We have seen that a limitation of our graph system is that the analytics that perform MPI communications must be serialized to avoid deadlocks during the graph execution. An idea to soften this issue would be to split the analytics into local parts and parts that actually perform MPI communications, or more generally any blocking computations that could induce a deadlock. The nodes of the graph would thus be split into several nodes linked with a dependency, allowing a more fine-grained serialization. Another idea would be to split the graph into several graphs based on the frequencies of the analytics. This way, heavy analytics could be executed at a low frequency without impacting the execution of more lightweight analytics at a higher frequency. Implementing this approach would require a more advanced data management system where several copies of the simulation data would be kept in memory but it could greatly reduce the synchronizations induced by TINS.

Improving In Transit Capacities We have proposed in this work an extension of the TINS framework to support in transit processing. We have implemented a prototype of in transit execution with only one staging node that let us show the benefit of the approach and the low overhead on the simulation execution time. Future works will include the development of more advanced in transit capacities with several staging nodes. The difficulty here is to manage data transfers efficiently from the compute nodes to the staging nodes, the two sets of nodes being usually of different sizes. The idea would therefore be to use existing libraries such as Bredala [45] to compute redistribution patterns and to transfer data thanks to non-blocking or one-sided communications in a MPMD context.

9.3 Towards Advanced Uses of TINS

In situ processing is an emerging way to analyze data at a high frequency by reducing the amount of data stored into the filesystem and the end-to-end time to scientific discovery. Unfortunately, this approach is not currently the preferred way to analyze data because the end-users often see it as more restricted than post-processing approaches, particularly because the analytics workflow has to be known in advance. However, the exascale era will certainly force the end-users to dramatically change their habits because of the poor I/O performance that exascale machines

will exhibit. It is therefore of the utmost importance to propose intuitive tools and to promote their use to smooth the passage to the exascale era.

With TINS, we set the bases of a hybrid framework to reduce the need to write data into the filesystem and we propose a task-based in situ method to optimize the resource usage on a compute node transparently for the user. TINS does not require complex dependencies and adding analytics in TINS consists in writing C++ plugins parallelized with the TBB library. The plugins can be developed as if they were post-processing codes thanks to a standalone mode and executed in situ or in transit without any code modifications. TINS is for the moment integrated into ExaStamp but we have begun to extract it so that it becomes a library that can be used by other simulation codes. In particular, TINS has been developed for a molecular dynamics code but it can be used by other communities, given that the simulations use a task-based programming model and MPI. In this thesis, we have also focused on simulation codes that do not use all the available memory of the nodes and for which copies of the data can be stored in the compute node memory but the task-based approach implemented in TINS leaves the opportunity for copying the data per block to reduce the memory footprint and hence reach a broader community.

The next step is to work with the physicists to design advanced in situ analytics scenarios. This way, we could identify potential shortcomings or missing features and improve the capabilities of our framework accordingly. The challenge is to make sure that physicists can be autonomous in using TINS and to guarantee that in situ processing reduces the end-to-end time to scientific discovery compared to post-processing approaches, without being a constraining tool difficult to use. To that end, we have already begun to work with physicists to improve the computation of numerical potentials thanks to machine learning methods. Today, these studies require to perform heavy computations synchronously with the simulation to find configuration samples that are added to a database used by a machine learning process. Instead of performing these heavy computations synchronously, TINS can be used to perform them asynchronously with the simulation, hence producing more configurations within a smaller end-to-end time and increasing the precision of the machine learning process and of the numerical potential. More generally, we are collaborating with physicists to design and deploy new analytics workflows to analyze in situ the behavior of matter under shock.

Part IV

Additional Content

10 Résumé de la Thèse en Français

10.1 Introduction

Pour comprendre et mettre en avant des phénomènes physiques, les physiciens créent des modèles et les valident, les invalident ou les raffinent grâce à des expériences, physiques ou numériques. Les expériences physiques se font dans des laboratoires qui reproduisent les conditions physiques du phénomène en jeu. Ces expériences physiques permettent aux physiciens de comprendre finement les phénomènes en jeu mais elles présentent un certains nombre d'inconvénients. Le premier est financier, car l'utilisation d'installations de pointe rendent les expériences physiques très onéreuses. Le deuxième est l'incapacité de reproduire certains phénomènes physiques en laboratoire, comme par exemple le mouvement des planètes en astrophysique ou les interactions entre océans et atmosphère en climatologie.

Au cours des dernières décennies, la simulation numérique est devenue un outil important pour pallier à certains des inconvénients des expériences physiques. Les phénomènes physiques sont modélisés par des modèles mathématiques résolus numériquement par des simulations numériques s'exécutant sur des *supercalculateurs*. Un supercalculateur peut être vu comme un ensemble d'ordinateurs, communément appelés nœuds de calcul, connectés entre eux par un réseau haute performance. Les nœuds de calcul sont aussi reliés à un système de fichiers pour sauvegarder des données sous forme de fichiers.

La puissance d'un supercalculateur se mesure en Flop/s, c'est-à-dire au nombre d'opérations flottantes (*floating point operations* en anglais) qu'il peut effectuer en une seconde. Alors que les premiers supercalculeurs effectuaient quelques opérations flottantes par seconde, la machine la plus puissantes en juin 2018, Summit [1], atteignait 120 PFlop/s, correspondant à 10^{17} Flop/s, ou encore cent millions de milliards d'opérations flottantes par seconde. Cette augmentation de la puissance de calcul n'est pas prête de s'arrêter et les constructeurs de machines ont déjà entamé une course vers l'*exascale* qui devrait mené, aux alentours de 2020, à des supercalculateurs capables d'effectuer un milliard de milliards d'opérations par seconde (10^{18} Flop/s).

Jusqu'aux années 2000, les constructeurs ont augmenté la puissance de calcul notamment grâce à l'augmentation de la fréquence des processeurs constituants les nœuds de calcul. En 1965, Gordon Moore a observé que le nombre de transistors dans un circuit intégré doublait à peu près tous les deux ans. En diminuant la taille des transistors, il était alors possible d'augmen-

[1] `https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/`

ter la fréquences des processeurs. Les codes de simulation pouvaient directement bénéficier de l'augmentation de la fréquence, s'exécutant de plus en plus vite sans modifier la moindre ligne de codes. Néanmoins, la loi de Moore s'essoufle depuis les années 2000, notamment parce que l'augmentation de la fréquence des processeurs entraîne une augmentation significative de la température des processeurs.

Pour s'adapter à cette contrainte, les constructeurs ont créés des processeurs *multi-cœurs*. Au lieu d'avoir une seule unité de calcul par processeur, les processeurs sont désormais équipés de plusieurs *cœurs* permettant d'effectuer des calculs en parallèle et réduisant l'enveloppe thermique associée. Les processeurs multi-cœurs actuels peuvent aller de quelques cœurs à quelques dizaines de cœurs. Le nombre de cœurs par processeur est en augmentation constante et les processeurs *many-cœurs*, avec plusieurs dizaines de cœurs à faible fréquence, sont de plus en plus prédominants. L'ère de l'exascale sera très certainement atteinte grâce à ce genre de technologies.

Pour pouvoir bénéficier de cette puissance de calcul, les codes de simulations doivent s'adapter aux différents niveaux de parallélisme et de mémoire offerts par les supercalculateurs actuels. Les codes optent de plus en plus pour une programmation dite *hybride*, où un modèle de programmation en *mémoire partagée* exploite les cœurs disponibles dans un nœud de calcul et un modèle de programmation en *mémoire distribuée* permet d'utiliser plusieurs nœuds de calcul. Le modèle de programmation en mémoire distribuée le plus utilisé est MPI (Message Passing Interface). Des *processus* MPI sont déployés sur les différents nœuds et s'échangent des données via des communications explicites.

Il existe plusieurs modèles de programmation en mémoire partagée. Ils se basent sur un ensemble de *threads* (fils d'exécution en français) qui sont exécutés sur les différents cœurs disponibles. Par exemple, la *programmation à base de tâches* propose une interface haut niveau où le programmeur décrit son programme sous forme de tâches qui sont distribuées de façon transparente par un ordonnanceur de tâches sur un ensemble de threads qu'il a créés. La programmation par tâches est susceptible de devenir le modèle de programmation standard pour les machines multi et many-cœurs. Des bibliothèques telles que Intel® Threading Building Blocks (TBB), proposent un modèle de programmation à base de tâches et un ordonnancement à base de *vol de tâches* pour équilibrer les charges de calcul exécutées par les différents threads.

Une simulation numérique s'effectue en deux étapes. Les ressources de calcul sont dans un premier temps utilisées pour résoudre les équations mathématiques. Ensuite, les données générées par la simulation sont périodiquement extraites et écrites sur le système de fichiers sous forme de fichiers de sorties. Ces fichiers de sorties ont vocation à être relus par des codes d'analyses qui, à partir des données brutes générées par la simulation, extraient des métriques utilisées par les physiciens pour comprendre les phénomènes physiques. On parle dans ce cas de *post-traitement* des données.

L'évolution des supercalculateurs permet d'effectuer des simulations de plus en plus précises. Par exemple, en dynamique moléculaire appliquée aux matériaux, l'étude du changement de phase d'un matériau sous choc était jusqu'à présent limitée à des systèmes de quelques millions de particules. Grâce aux nouvelles architectures, il est désormais possible de simuler des systèmes de plusieurs centaines de millions de particules, améliorant grandement la compréhension des phénomènes en jeu. Mais cette augmentation du nombre de particules signifie aussi une augmentation de la quantité de données à écrire sur le système de fichiers. Néanmoins, les capacités de transfert et d'écriture n'évoluent pas aussi rapidement que les capacités de calculs. Les

simulations sont donc de plus en plus pénalisées par les sorties de données sur le systèmes de fichiers et le temps total jusqu'à la découverte scientifique est grandement augmenté.

Afin de diminuer le besoin d'écrire des données sur les systèmes de fichiers, les techniques de l'*in situ* proposent d'analyser les données directement là où elles sont produites. Il en existe trois types : le *in situ* à proprement parler où simulation et analyses sont exécutées sur les mêmes nœuds de calcul, le *in transit* où les analyses sont exécutées sur des nœuds dédiés et l'approche *hybride* où des analyses peu coûteuses sont exécutées in situ et des analyses plus coûteuses in transit.

Les méthodes traditionnelles de l'in situ (sur les mêmes nœuds de calcul) sont adaptées pour des codes hybrides MPI+X (où X correspond à un modèle de programmation en mémoire partagée) qui n'arrivent pas à bénéficier pleinement des cœurs d'un processeur multi-cœurs. La plupart des codes de simulation hybrides peuvent en effet tirer profit de quelques cœurs mais ne sont souvent pas efficaces sur les dizaines de cœurs proposés par ces architectures. Les bibliothèques d'in situ proposent donc d'utiliser ces ressources sous-exploitées par la simulation pour exécuter des analyses de manière asynchrone. Il existe deux grandes techniques : les analyses peuvent être exécutées pendant les régions séquentielles de la simulation [119] ou des cœurs, appelés *helper cores* en anglais, peuvent être confisqués à la simulation pour être dédiés à l'analyse [41, 44]. Ces techniques sont peu adaptées aux architectures many-cœurs et encore moins aux codes qui ont été optimisés pour ces architectures. De plus, bien que la programmation par tâches aient été identifiée comme propice au développement de systèmes in situ [93], peu de travaux se sont concentrés jusqu'à présent sur le développement de systèmes in situ à base de tâches pour les codes MPI+X.

Cette étude s'inscrit dans ce contexte et consiste à étudier l'intégration d'une architecture in situ à base de tâches au sein d'un code de dynamique moléculaire optimisé pour les architectures multi et many-cœurs. Le code cible est ExaStamp, un code hybride MPI+TBB développé au CEA et qui atteint des performances presque optimales sur des supercalculateurs à la pointe de la technologie. Les techniques traditionnelles de l'in situ ne sont pas efficaces sur un code tel qu'ExaStamp, soit parce que les régions séquentielles sont trop courtes pour être exploitées sans perturber l'exécution de la simulation, soit parce que confisquer des cœurs à la simulation a un impact important sur le temps de la simulation. Compte tenu des bonnes propriétés d'équilibrage de charge du vol de tâches, l'idée est donc de créer des tâches de simulation et d'analyse de façon concurrente et de tirer parti du mécanisme de vol de tâches pour les entrelacer efficacement sur les mêmes cœurs. Plus généralement, le but de la thèse est de tirer profit du modèle de programmation à base de tâches pour proposer une architecture hybride où les mêmes analyses peuvent être exécutées in situ, in transit ou de manière hybride de façon transparente pour l'utilisateur.

10.2 Organisation du Manuscrit

Ce manuscrit rend compte de la démarche adoptée pour proposer une architecture hybride de traitement des données in situ qui tire profit de la programmation par tâches et du vol de tâches pour entrelacer de façon efficace simulation et analyses. Ce manuscrit s'organise en trois parties. Dans un premier temps (Chapitres 2 et 3), nous étudions plus en détails le contexte de cette étude, les solutions existantes pour le traitement in situ, in transit et hybride des données et nous présentons plus en détails la bibliothèque TBB et le code de dynamique moléculaire ExaStamp. Une deuxième partie (Chapitres 4, 5 et 6) est consacrée à la mise en place d'une méthode dynamique

de traitement des données in situ. Finalement, une troisième et dernière partie (Chapitres 7, 8) est dédiée au développement d'une architecture évolutive, intuitive et générique pour le traitement des données in situ. Ce document étant rédigé en anglais, nous résumons dans cette section les idées principales et les résultats principaux des cinq chapitres de démarche en français. Nous encourageons vivement le lecteur à lire les chapitres en anglais pour de plus amples informations sur la démarche et les résultats obtenus.

ExaStamp est un nouveau code qui ne possède pas nativement de mécanismes pour analyser les données in situ. Nous proposons donc dans le Chapitre 4 d'intégrer des analyses représentatives de la physique à l'intérieur du code et de les exécuter dans un premier temps de façon *synchrone* : périodiquement, la simulation est stoppée et les ressources de calcul sont utilisées pour exécuter une analyse à la place. Ceci nous permet dans un premier temps de vérifier l'intérêt en terme de temps de calcul de l'in situ par rapport à la technique traditionnelle de post-traitement. Grâce au développement d'un outil pour visualiser l'exécution des tâches, nous mettons en évidence les régions séquentielles du code de simulation, qui représentent une opportunité pour l'implantation d'un système in situ plus efficace. Nous proposons donc de mettre en place une exécution asynchrone à base de tâches que nous appelons TINS (pour *Task-based IN Situ* en anglais). Périodiquement, la simulation crée une tâche d'analyse, la soumet à l'ordonnanceur de tâches et continue l'exécution de la simulation sans attendre que la tâche d'analyse n'ait été exécutée. Simulation et analyses s'exécutent donc de manière concurrente grâce à l'ordonnanceur de TBB et nous montrons que TINS peut être jusqu'à 39% plus rapide qu'une exécution synchrone.

Dans le Chapitre 5, nous comparons TINS avec deux bibliothèques existantes, Goldrush et Damaris. Nous montrons que TINS est jusqu'à 12% plus rapide que Goldrush, notamment en raison du fait que les régions séquentielles d'ExaStamp sont trop courtes pour être exploitées mais aussi à cause de l'approche à base de processus de Goldrush qui induit deux ordonnanceurs de TBB sur un nœud alors qu'il n'y en n'a qu'un seul dans TINS. Nous montrons ensuite que le temps d'exécution avec Damaris est plus faible qu'avec TINS pour de grosses analyses et des analyses séquentielles, ce qui motive le développement d'un mécanisme d'isolation des tâches dans TINS. Pour ce faire, nous utilisons des mécanismes natifs de TBB pour créer un système d'arènes attachées sur des ensembles disjoints de cœurs. La création de tâches d'analyses se fait grâce à un thread d'analyse créé à l'initialisation par le thread principal de la simulation. Notre système est donc composé d'un thread de simulation qui crée des tâches de simulation dans une arène de simulation et qui copie périodiquement des données dans un buffer temporaire, d'un thread d'analyse qui se synchronise avec le thread de simulation pour créer des tâches d'analyse dans l'arène d'analyse une fois que des données sont disponibles dans le buffer temporaire et de $N - 2$ *worker threads* (N étant le nombre de cœurs dans le processeur) qui sont assignés aux deux arènes (n_a threads dans l'arène d'analyse et n_s threads dans l'arène de simulation avec $n_a + n_s = N$) pour exécuter les différentes tâches. Nous montrons que TINS avec ce mécanisme de *helper cores* statique est équivalent voire même meilleur que Damaris sur des analyses avec beaucoup d'accès mémoire. TINS avec *helper cores* statique est de plus jusqu'à 42% meilleur que TINS sans isolation. Cependant, nous montrons une limitation de l'approche par *helper cores* statique : il faut choisir le nombre de *helper cores* (n_a) suffisamment grand pour que l'analyse puisse bénéficier de sa parallélisation et suffisamment petit pour que le temps d'exécution de la simulation ne soit pas trop impacté.

Le Chapitre 6 est dédié à la mise en place de méthodes dynamiques et adaptatives pour pallier

aux inconvénients du *helper cores* statique. Nous proposons dans un premier temps une méthode de *helper cores* statique adaptative qui détermine automatiquement le nombre de *helper cores* optimal, c'est-à-dire n_a tel que le temps d'exécution de la simulation soit à peu près égal au temps d'exécution de l'analyse. Cependant, nous montrons que l'implantation du telle méthode dans TINS rajouterait une synchronisation entre les threads de simulation et d'analyse. De plus, cette méthode ne permet toujours pas de tirer profit des régions séquentielles de la simulation et de l'analyse. Nous mettons donc en place dans un second temps une méthode de *helper cores* dynamique où l'isolation entre les threads n'est plus permanent comme dans le *helper cores* statique mais temporaire. Les threads sont séparés en deux groupes lorsque des tâches de simulation et d'analyse sont disponibles en même temps mais cette restriction est relevée quand la simulation rentre dans une région séquentielle ou que l'exécution de l'analyse est terminée. Cette méthode utilise le système d'arènes introduit précédemment mais avec $n_s = N$ et $n_a > 0$ de sorte à avoir $n_a + n_s > N$. L'utilisateur doit choisir n_a, selon le degré de priorité qu'il veut accorder à la simulation, mais nous montrons que le choix de n_a dans le cas dynamique est beaucoup moins punitif que le choix de n_a dans le cas statique. Nous montrons en particulier que la méthode dynamique est jusqu'à 40% plus rapide que la méthode statique sur un ensemble de scenarios, incluant une simulation de 2 milliards de particules sur 14,000 cœurs avec des analyses aux besoins différents à chaque itération. Finalement, nous proposons d'implanter la méthode d'adaptation de n_a dans le cas dynamique pour automatiquement trouver la meilleure configuration. Nous montrons que l'algorithme permet de trouver la meilleure configuration pour un ensemble d'analyses mais qu'il faut pour certaines analyses un nombre conséquent d'itérations pour y arriver.

Nous présentons dans le Chapitre 7 l'architecture de TINS qui est conçue de sorte à séparer les codes de simulation et d'analyse. Jusqu'à présent, les codes d'analyses étaient directement intégrés dans le code de la simulation mais ceci pose des problèmes, notamment pour la généricité de l'approche. En effet, coder les analyses à l'intérieur du code de la simulation impose des contraintes sur le développement des analyses, contraintes qui sont différentes d'un code de simulation à un autre. Nous développons donc un système de plugins où les analyses sont développées sous forme de plugins chargés dynamiquement lors de l'exécution de la simulation. Un objet est partagé par la simulation et le plugins, le but de cet objet étant de masquer les synchronisations entre la simulation et les analyses et de masquer la façon dont sont stockées les données en mémoire. Le développeur utilise cet objet pour récupérer des pointeurs vers les données en mémoire et développe les analyses comme si c'était des analyses de post-traitement, sans se soucier du fait qu'elles vont être exécutées in situ avec la simulation. Nous développons aussi un système de graphe à base de tâches pour la création de *workflows* d'analyses complexes. L'utilisateur décrit les analyses qu'il veut exécuter dans un fichier externe et TINS en déduit un graphe où chaque nœud représente une analyse et les liaisons entre les nœuds des dépendances de données entre les analyses. Chaque analyse est vue comme une tâche à gros grain et la tâche d'analyse peut à son tour créer des tâches qui vont être intercalées avec la simulation dans une exécution in situ. Nous ajoutons finalement dans TINS la capacité d'exécuter les analyses in transit et nous développons un prototype qui nous permet de valider l'intérêt de cette approche. Le plus gros avantage de l'architecture de TINS est que les analyses peuvent être exécutées in situ, in transit ou même en post-traitement de manière totalement transparente et sans aucune modification des codes d'analyses. Ceci en fait une architecture flexible, intuitive et générique qui peut être utilisée par d'autres codes de simulation qu'ExaStamp.

Finalement, le Chapitre 8 valide l'architecture de TINS en l'utilisant pour analyser in situ les résultats d'une simulation de production. Nous présentons d'abord le cas test qui étudie le changement de phase d'un matériau d'étain sous choc. Nous implantons une analyse structurelle sous forme de plugin et nous montrons que l'architecture de TINS est suffisamment souple pour extraire facilement plus de données de la simulation afin d'accélérer le calcul de l'analyse. Nous présentons ensuite le supercalculateur cible et en particulier les limitations techniques de cette machine qui nous empêchent d'utiliser toutes les fonctionnalités de TINS. Nous arrivons néanmoins à exécuter un *workflow* d'analyses complexe avec TINS en quelques secondes seulement alors qu'il nécessite une dizaine de minutes avec les outils de post-traitement traditionnellement utilisés. De plus, nous mettons en évidence la robustesse de TINS sur un calcul de plusieurs jours et nous montrons un surcoût de TINS sur le temps de la simulation de moins de 10%. Outre le gain de temps de calcul permis par TINS, nous montrons aussi que l'architecture est intuitive et peut permettre aux physiciens habitués aux outils de post-traitement de transiter vers des techniques in situ malheureusement encore peu utilisées par cette communauté.

10.3 Conclusion

Traditionnellement, la découverte scientifique s'effectue en deux temps. Des simulations numériques sont tout d'abord calibrées et exécutées sur des supercalculateurs. Les simulations produisent des données qui sont périodiquement écrites sur les systèmes de fichiers des supercalculateurs. Des codes d'analyses sont ensuite utilisés pour lire ces données et en extraire des métriques utilisables par le physicien pour comprendre les phénomènes physiques. Avec l'évolution actuelle des supercalculateurs, il est possible de simuler des phénomènes physiques de plus en plus complexes grâce à des systèmes numériques de plus en plus grand. Cependant, les capacités d'écriture et de lecture n'évoluent pas aussi rapidement que les capacités de calcul et la technique traditionnelle de post-traitement entraîne un temps de plus en plus long jusqu'à la découverte scientifique. Les techniques de l'in situ voient donc le jour pour diminuer le besoin d'écrire des données sur les systèmes de fichier en les analysant directement là où elles sont produites. De nombreuses études ont été réalisée pour fournir aux utilisateurs des techniques performantes pour l'analyse in situ des données mais ces études se focalisent majoritairement sur les codes qui n'arrivent pas à tirer profit du nombre croissant de cœurs par processeur. Elles sont en particulier moins performante pour des codes qui sont optimisés pour les architectures multi et many-cœurs. De plus, peu de méthodes in situ se basent sur la programmation par tâches qui est de plus en plus prédominante sur les machines actuelles.

Dans cette thèse, nous étudions l'intégration d'une architecture de traitement des données in situ à base de tâches pour un code de dynamique moléculaire optimisé pour les architectures multi et many-cœurs. Nous proposons l'architecture TINS (pour *Task-based IN Situ* en anglais) qui utilise un modèle de programmation à base de tâches pour créer simultanément tâches de simulation et d'analyses et un ordonnanceur à base de vol de tâches pour entrelacer efficacement simulation et analyses. TINS est une architecture hybride où les mêmes analyses peuvent être exécutées de façon transparente in situ, in transit ou en mode post-traitement. L'exécution in situ se fait grâce à une méthode innovante de *helper core* dynamique où les threads sont séparés en deux groupes uniquement lorsque des tâches des deux types existent, la restriction étant levée dans le cas contraire. Ceci permet d'exécuter in situ des analyses jusqu'à 40% plus rapidement qu'avec les méthodes traditionnelles. Ces résultats ont mené à la publication d'un article dans

une conférence internationale [37]. TINS propose une interface intuitive pour le développement d'analyses et pour la mise en place de *workflows* d'analyses complexes. Lors de cette thèse, nous avons montré que TINS est une architecture intuitive, robuste et évolutive capable d'exécuter des *workflows* d'analyse complexes dans un environnement de production et avec un surcoût pour la simulation de moins de 10%.

Bien que TINS ait montré sa robustesse dans un environnement de production, son utilisation intuitive pour des développeurs non experts du domaine et ses performances comparées aux méthodes de l'état de l'art, ce travail ouvre aussi des perspectives pour le futur. Ces travaux futurs incluent la généralisation de la méthode de *helper core* dynamique à d'autres modèles de programmation par tâches que TBB, l'ajout de capacités de pilotage d'analyses pour modifier le *workflow* d'analyse pendant l'exécution de TINS et la mise en place d'un système in transit plus avancé. Avec toutes ses fonctionnalités, nous espérons que TINS saura encourager les physiciens à changer leurs habitudes de traitement des données et à migrer vers le traitement des données in situ pour diminuer le temps nécessaire jusqu'à la découverte scientifique.

Bibliography

[1] Boost c++ libraries. https://www.boost.org/. Accessed: 2018-07-02.
➢ Cited on page 37.

[2] Conduit: Simplified data exchange for hpc simulations. https://llnl-conduit.readthedocs.io/en/latest/. Accessed: 2018-06-29.
➢ Cited on pages 42 and 119.

[3] The openmp api specification for parallel programming. https://www.openmp.org. Accessed: 2018-06-29.
➢ Cited on page 28.

[4] Tbb initialization, termination, and resource management details, juicy and gory. https://software.intel.com/en-us/blogs/2011/04/09/tbb-initialization-termination-and-resource-management-details-juicy-and-gory. Accessed: 2018-06-23.
➢ Cited on pages 102 and 108.

[5] Threading building blocks. https://www.threadingbuildingblocks.org/. Accessed: 2018-06-23.
➢ Cited on pages 19, 28, and 152.

[6] Top500 supercomputer sites. https://www.top500.org/. Accessed: 2018-07-06.
➢ Cited on pages 25, 29, and 30.

[7] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Paul Thibault. Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Transactions on Parallel and Distributed Systems*, 2017.
➢ Cited on page 31.

[8] Sean Ahern, Arie Shoshani, Kwan-Liu Ma, Alok Choudhary, Terence Critchlow, Scott Klasky, Valerio Pascucci, Jim Ahrens, E Wes Bethel, Hank Childs, et al. Scientific discovery at the exascale. report from the doe ascr 2011 workshop on exascale data management. *Analysis, and Visualization*, 2(3), 2011.
➢ Cited on pages 18 and 32.

[9] James Ahrens, John Patchett, Andrew Bauer, Sébastien Jourdain, David H Rogers, Mark Petersen, Benjamin Boeckel, Patrick OLeary, Patricia Fasel, and Francesca Samsel. In situ mpas-ocean image-based visualization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Visualization & Data Analytics Showcase*, 2014.
➢ Cited on page 33.

[10] Jérémie Allard, Jean-Denis Lesage, and Bruno Raffin. Modularity for large virtual reality applications. *Presence: Teleoperators and Virtual Environments*, 19(2):142–161, 2010.
➢ Cited on page 41.

[11] MP Allen and DJ Tildesley. Computer simulation of liquids. 1987.
➢ Cited on page 53.

[12] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Steve Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pages 423–424. IEEE, 2004.
➢ Cited on page 42.

[13] Saman Amarasinghe, Dan Campbell, William Carlson, Andrew Chien, William Dally, Elmootazbellah Elnohazy, Mary Hall, Robert Har-

rison, William Harrod, Kerry Hill, et al. Exascale software study: Software challenges in extreme scale systems. *DARPA IPTO, Air Force Research Labs, Tech. Rep*, pages 1–153, 2009.
➣ Cited on page 31.

[14] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. Starpu-mpi: Task programming over clusters of machines enhanced with accelerators. In *European MPI Users' Group Meeting*, pages 298–299. Springer, 2012.
➣ Cited on page 31.

[15] 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.
➣ Cited on pages 28 and 153.

[16] Utkarsh Ayachit. The paraview guide. *Kitware Inc*, 2015.
➣ Cited on page 34.

[17] Utkarsh Ayachit, Brad Whitlock, Matthew Wolf, Burlen Loring, Berk Geveci, David Lonie, and E Wes Bethel. The sensei generic in situ interface. In *In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV), Workshop on*, pages 40–44. IEEE, 2016.
➣ Cited on page 34.

[18] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O'Leary, V. Vishwanath, B. Whitlock, and E. W. Bethel. In situ methods, infrastructures, and applications on high performance computing platforms. *Computer Graphics Forum*, 35(3):577–597, 2016.
➣ Cited on page 33.

[19] Andrew C Bauer, Berk Geveci, and Will Schroeder. The paraview catalyst user's guide v2. 0. kitware, 2015.
➣ Cited on page 34.

[20] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 66. IEEE Computer Society Press, 2012.
➣ Cited on pages 31 and 153.

[21] Janine C. Bennett, Hasan Abbasi, Peer-Timo Bremer, Ray Grout, Attila Gyulassy, Tong Jin, Scott Klasky, Hemanth Kolla, Manish Parashar, Valerio Pascucci, Philippe Pebay, David Thompson, Hongfeng Yu, Fan Zhang, and Jacqueline Chen. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 49:1–49:9, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
➣ Cited on pages 32 and 40.

[22] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM, 1995.
➣ Cited on pages 19 and 28.

[23] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.
➣ Cited on page 28.

[24] Erik Boman, Karen Devine, Lee Ann Fisk, Robert Heaphy, Bruce Hendrickson, Courtenay Vaughan, Umit Catalyurek, Doruk Bozdag, William Mitchell, and James Teresco. Zoltan 3.0: parallel partitioning, load-balancing, and data management services; user's guide. *Sandia National Laboratories, Albuquerque, NM*, 2007.
➣ Cited on page 144.

[25] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.
➣ Cited on page 27.

[26] Thomas E Cheatham III and Daniel R Roe. The impact of heterogeneous computing on workflows for biomolecular simulation and analysis. *Computing in Science & Engineering*, 17(2):30–39, 2015.
➣ Cited on page 67.

[27] Hank Childs, Eric Brugger, Brad Whitlock, Jeremy Meredith, Sean Ahern, David Pugmire, Kathleen Biagas, Mark Miller, Gunther H Weber, Hari Krishnan, et al. Visit: An end-user tool for visualizing and analyzing very large data. Technical report, Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US), 2012.
➣ Cited on page 34.

[28] Younghyun Cho, Surim Oh, and Bernhard Egger. Adaptive space-shared scheduling for shared-memory parallel programs. In *Job Scheduling Strategies for Parallel Processing*, pages 158–177. Springer, 2015.
➣ Cited on page 38.

[29] Emmanuel Cieren, Laurent Colombet, Samuel Pitoiset, and Raymond Namyst. Exastamp: A parallel framework for molecular dynamics on heterogeneous clusters. In *European Conference on Parallel Processing*, pages 121–132. Springer,

2014.
➣ Cited on pages 19, 54, 55, and 151.

[30] K Coulomb, M Faverge, J Jazeix, O Lagrasse, J Marcoueille, P Noisette, A Redondy, and C Vuchener. Visual trace explorer (vite). Technical report, Technical report, 2009.
➣ Cited on page 70.

[31] David E Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999.
➣ Cited on page 26.

[32] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.
➣ Cited on page 28.

[33] J Chassin De Kergommeaux, Benhur Stein, and Pierre-Eric Bernard. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing*, 26(10):1253–1274, 2000.
➣ Cited on page 70.

[34] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
➣ Cited on page 39.

[35] Ewa Deelman, Tom Peterka, Ilkay Altintas, Christopher D Carothers, Kerstin Kleese van Dam, Kenneth Moreland, Manish Parashar, Lavanya Ramakrishnan, Michela Taufer, and Jeffrey Vetter. The future of scientific workflows. *The International Journal of High Performance Computing Applications*, 32(1):159–175, 2018.
➣ Cited on page 42.

[36] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.
➣ Cited on page 42.

[37] Estelle Dirand, Laurent Colombet, and Bruno Raffin. Tins: A task-based dynamic helper core strategy for in situ analytics. In *Asian Conference on Supercomputing Frontiers*, pages 159–178. Springer, 2018.
➣ Cited on pages 20, 152, and 165.

[38] Ciprian Docan, Manish Parashar, and Scott Klasky. Dart: a substrate for high speed asynchronous data io. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 219–220. ACM, 2008.
➣ Cited on page 39.

[39] Ciprian Docan, Manish Parashar, and Scott Klasky. DataSpaces: an Interaction and Coordination Framework for Coupled Simulation Workflows. *Cluster Computing*, 15(2):163–181, 2012.
➣ Cited on page 39.

[40] Stephanie Donovan, Gerrit Huizenga, Andrew J Hutton, C Craig Ross, Martin K Petersen, and Philip Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the Linux Symposium*, volume 2003, 2003.
➣ Cited on page 32.

[41] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf. Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free i/o. In *2012 IEEE International Conference on Cluster Computing*, pages 155–163, Sept 2012.
➣ Cited on pages 37 and 161.

[42] M. Dorier, R. Sisneros, T. Peterka, G. Antoniu, and D. Semeraro. Damaris/viz: A nonintrusive, adaptable and user-friendly in situ visualization framework. In *Large-Scale Data Analysis and Visualization (LDAV), 2013 IEEE Symposium on*, pages 67–75, Oct 2013.
➣ Cited on pages 37, 76, 77, 81, 130, and 152.

[43] Matthieu Dorier, Robert Sisneros, Leonardo Bautista Gomez, Tom Peterka, Leigh Orf, Lokman Rahmani, Gabriel Antoniu, and Luc Bougé. Adaptive performance-constrained in situ visualization of atmospheric simulations. In *Cluster Computing (CLUSTER), 2016 IEEE International Conference on*, pages 269–278. IEEE, 2016.
➣ Cited on page 37.

[44] M. Dreher and B. Raffin. A flexible framework for asynchronous in situ and in transit analytics for scientific simulations. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 277–286, May 2014.
➣ Cited on pages 41, 42, 130, and 161.

[45] Matthieu Dreher and Tom Peterka. Bredala: Semantic data redistribution for in situ applications. In *Cluster Computing (CLUSTER), 2016 IEEE International Conference on*, pages 279–288. IEEE, 2016.
➣ Cited on pages 42, 131, and 154.

[46] Matthieu Dreher and Tom Peterka. Decaf: Decoupled dataflows for in situ high-performance workflows. Technical report, Argonne National

Lab.(ANL), Argonne, IL (United States), 2017.
➣ Cited on pages 41 and 130.

[47] Matthieu Dreher, Kiran Sasikumar, Subramanian Sankaranarayanan, and Tom Peterka. Manala: a flexible flow control library for asynchronous task communication. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 509–519. IEEE, 2017.
➣ Cited on page 43.

[48] O Durand and L Soulard. Power law and exponential ejecta size distributions from the dynamic fragmentation of shock-loaded cu and sn metals under melt conditions. *Journal of Applied Physics*, 114(19):194902, 2013.
➣ Cited on pages 54 and 135.

[49] Olivier Durand, S Jaouen, L Soulard, Olivier Heuze, and Laurent Colombet. Comparative simulations of microjetting using atomistic and continuous approaches in the presence of viscosity and surface tension. *Journal of Applied Physics*, 122(13):135107, 2017.
➣ Cited on page 54.

[50] Mike Folk, Albert Cheng, and Kim Yates. Hdf5: A file format and i/o library for high performance computing applications. In *Proceedings of supercomputing*, volume 99, pages 5–33, 1999.
➣ Cited on page 32.

[51] Juliana Freire, Cláudio T Silva, Steven P Callahan, Emanuele Santos, Carlos E Scheidegger, and Huy T Vo. Managing rapidly-evolving scientific workflows. In *International Provenance and Annotation Workshop*, pages 10–18. Springer, 2006.
➣ Cited on page 42.

[52] Yuankun Fu, Feng Li, Fengguang Song, and Zizhong Chen. Performance analysis and optimization of in-situ integration of simulation with data analysis: zipping applications up. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 192–205. ACM, 2018.
➣ Cited on page 43.

[53] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 15–23. ACM, 2007.
➣ Cited on page 28.

[54] A. Goswami, Y. Tian, K. Schwan, F. Zheng, J. Young, M. Wolf, G. Eisenhauer, and S. Klasky. Landrush: Rethinking in-situ analysis for gpgpu workflows. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 32–41, May 2016.
➣ Cited on page 153.

[55] Dominik Grewe, Zheng Wang, and Michael FP O'Boyle. A workload-aware mapping approach for data-parallel programs. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 117–126. ACM, 2011.
➣ Cited on page 38.

[56] Salman Habib, Vitali Morozov, Hal Finkel, Adrian Pope, Katrin Heitmann, Kalyan Kumaran, Tom Peterka, Joe Insley, David Daniel, Patricia Fasel, et al. The universe at extreme scale: multi-petaflop sky simulation on the bg/q. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 4. IEEE Computer Society Press, 2012.
➣ Cited on pages 33 and 65.

[57] Tim Harris, Martin Maas, and Virendra J Marathe. Callisto: co-scheduling parallel runtime systems. In *Proceedings of the Ninth European Conference on Computer Systems*, page 24. ACM, 2014.
➣ Cited on pages 38, 77, and 100.

[58] Alan Heirich, Elliott Slaughter, Manolis Papadakis, Wonchan Lee, Tim Biedert, and Alex Aiken. In situ visualization with task-based parallelism. 2017.
➣ Cited on page 42.

[59] Monica Liliana Hernandez Ariza, Matthieu Dreher, Carlos Jaime Barrios-Hernandez, and Bruno Raffin. Asynchronous In Situ Processing with Gromacs: Taking Advantage of GPUs. In *Latin America High Performance Computing Conference*, Petropolis, Brazil, August 2015.
➣ Cited on page 153.

[60] Reazul Hoque, Thomas Herault, George Bosilca, and Jack Dongarra. Dynamic task discovery in parsec: a data-flow task-based runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, page 6. ACM, 2017.
➣ Cited on page 31.

[61] Andra-Ecaterina Hugo, Abdou Guermouche, Raymond Namyst, and Pierre-André Wacrenier. Composing multiple starpu applications over heterogeneous machines: a supervised approach. In *Third International Workshop on Accelerators and Hybrid Exascale Systems*, Boston, United States, May 2013.
➣ Cited on page 38.

[62] William Humphrey, Andrew Dalke, and Klaus Schulten. Vmd: visual molecular dynamics. *Journal of molecular graphics*, 14(1):33–38, 1996.
➣ Cited on pages 57 and 67.

[63] Alexandru C Iordan, Magnus Jahre, and Lasse Natvig. Tuning the victim selection policy of intel tbb. *Journal of Systems Architecture*, 61(10):584–591, 2015.
➣ Cited on page 50.

[64] Sylvie Joussaume, A Bellucci, J Biercamp, Reinhard Budich, A Dawson, Marie-Alice Foujols, B Lawrence, L Linardikis, Sébastien Masson, Yann Meurdesoif, et al. Modelling the earth's climate system: data and computing challenges. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 2325–2356. IEEE, 2012.
➣ Cited on page 32.

[65] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, page 6. ACM, 2014.
➣ Cited on pages 31 and 153.

[66] Hartmut Kaiser, Thomas Heller, Daniel Bourgeois, and Dietmar Fey. Higher-level parallelization for local and distributed asynchronous task-based programming. In *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*, pages 29–37. ACM, 2015.
➣ Cited on page 31.

[67] James Kress, Scott Klasky, Norbert Podhorszki, Jong Choi, Hank Childs, and David Pugmire. Loosely Coupled In Situ Visualization: A Perspective on Why It's Here to Stay. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ISAV2015, pages 1–6, New York, NY, USA, 2015. ACM.
➣ Cited on page 38.

[68] T Kuhlen, R Pajarola, and K Zhou. Parallel in situ coupling of simulation with a fully featured visualization system. In *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization (EGPGV)*, 2011.
➣ Cited on page 34.

[69] Matthew Larsen, James Ahrens, Utkarsh Ayachit, Eric Brugger, Hank Childs, Berk Geveci, and Cyrus Harrison. The alpine in situ infrastructure: Ascending from the ashes of strawman. In *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, pages 42–46. ACM, 2017.
➣ Cited on page 42.

[70] Matthew Larsen, Eric Brugger, Hank Childs, Jim Eliot, Kevin Griffin, and Cyrus Harrison. Strawman: A Batch In Situ Visualization and Analysis Infrastructure for Multi-Physics Simulation Codes. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ISAV2015, pages 30–35, New York, NY, USA, 2015. ACM.
➣ Cited on page 42.

[71] Wolfgang Lechner and Christoph Dellago. Accurate determination of crystal structures based on averaged local bond order parameters. *The Journal of chemical physics*, 129(11):114707, 2008.
➣ Cited on page 136.

[72] Tau Leng, Rizwan Ali, Jenwei Hsieh, Victor Mashayekhi, and Reza Rooholamini. An empirical study of hyper-threading in high performance computing clusters. *Linux HPC Revolution*, 45, 2002.
➣ Cited on page 29.

[73] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Robert Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *Supercomputing, 2003 Acm/Ieee Conference*, pages 39–39. IEEE, 2003.
➣ Cited on page 32.

[74] Min Li, Sudharshan S Vazhkudai, Ali R Butt, Fei Meng, Xiaosong Ma, Youngjae Kim, Christian Engelmann, and Galen Shipman. Functional partitioning to optimize end-to-end performance on many-core architectures. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–12. IEEE, 2010.
➣ Cited on page 37.

[75] Esther Liu, Jiand Pacitti, Patrick Valduriez, and Marta Mattoso. A survey of data-intensive scientific workflow management. *Journal of Grid Computing*, 13(4):457–493, 2015.
➣ Cited on page 42.

[76] Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, Manish Parashar, Nagiza Samatova, Karsten Schwan, Arie Shoshani, Matthew Wolf, Kesheng Wu, and Weikuan Yu. Hello adios: The challenges and lessons of developing leadership class i/o frameworks. *Concurr. Comput. : Pract. Exper.*, 26(7):1453–1473, May 2014.
➣ Cited on pages 32 and 42.

[77] Jay F. Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible IO and Integration for Scientific Codes Through The Adaptable IO System (ADIOS). In *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*, CLADE '08, pages 15–24, New York, NY, USA, 2008. ACM.
➣ Cited on page 78.

[78] Xiaosong Ma, Jonghyun Lee, and Marianne Winslett. High-level buffering for hiding periodic output cost in scientific simulations. *IEEE Transactions on Parallel and Distributed Systems*, 17(3):193–204, 2006.
➣ Cited on page 37.

[79] Preeti Malakar, Venkatram Vishwanath, Christopher Knight, Todd Munson, and Michael E Papka. Optimal execution of co-analysis for large-scale molecular dynamics simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 60. IEEE Press, 2016.
➣ Cited on page 43.

[80] Preeti Malakar, Venkatram Vishwanath, Todd Munson, Christopher Knight, Mark Hereld, Sven Leyffer, and Michael E. Papka. Optimal scheduling of in-situ analysis for large-scale scientific simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 52:1–52:11, New York, NY, USA, 2015. ACM.
➣ Cited on page 43.

[81] Timothy G Mattson, Romain Cledat, Vincent Cavé, Vivek Sarkar, Zoran Budimlic, Sanjay Chatterjee, Joshua B Fryman, Ivan Ganev, Robin Knauerhase, Min Lee, et al. The open community runtime: A runtime system for extreme scale computing. In *HPEC*, pages 1–7, 2016.
➣ Cited on page 28.

[82] Michael D McCool, Arch D Robison, and James Reinders. *Structured parallel programming: patterns for efficient computation.* Elsevier, 2012.
➣ Cited on page 28.

[83] Naveen Michaud-Agrawal, Elizabeth J Denning, Thomas B Woolf, and Oliver Beckstein. Mdanalysis: a toolkit for the analysis of molecular dynamics simulations. *Journal of computational chemistry*, 32(10):2319–2327, 2011.
➣ Cited on page 67.

[84] Paolo Missier, Stian Soiland-Reyes, Stuart Owen, Wei Tan, Alexandra Nenadic, Ian Dunlop, Alan Williams, Tom Oinn, and Carole Goble. Taverna, reloaded. In *International conference on scientific and statistical database management*, pages 471–481. Springer, 2010.
➣ Cited on page 42.

[85] Clément Mommessin, Matthieu Dreher, Bruno Raffin, and Tom Peterka. Automatic data filtering for in situ workflows. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*, pages 370–378. IEEE, 2017.
➣ Cited on pages 43 and 126.

[86] Oscar H Mondragon, Patrick G Bridges, and Terry Jones. Quantifying scheduling challenges for exascale system software. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, page 8. ACM, 2015.
➣ Cited on page 35.

[87] Oscar H Mondragon, Patrick G Bridges, Scott Levy, Kurt B Ferreira, and Patrick Widener. Scheduling in-situ analytics in next-generation applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pages 102–105. IEEE, 2016.
➣ Cited on pages 36 and 77.

[88] Gordon E Moore. Cramming more components onto integrated circuits. electronics 38 (8): 114–117, 1965.
➣ Cited on page 25.

[89] Ryan W Moore and Bruce R Childers. Using utility prediction models to dynamically choose program thread counts. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 135–144. IEEE, 2012.
➣ Cited on page 38.

[90] Kenneth Moreland, Christopher Sewell, William Usher, Li-ta Lo, Jeremy Meredith, David Pugmire, James Kress, Hendrik Schroots, Kwan-Liu Ma, Hank Childs, et al. Vtk-m: Accelerating the visualization toolkit for massively threaded architectures. *IEEE computer graphics and applications*, 36(3):48–58, 2016.
➣ Cited on page 42.

[91] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.
➣ Cited on page 78.

[92] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.
➣ Cited on page 27.

[93] Philippe Pébaÿ and Janine Bennett. An asynchronous many-task implementation of in-situ statistical analysis using legion. *Sandia National Laboratories, Sandia Report SAND2015-10345*, 2015.
➢ Cited on page 161.

[94] Philippe Pebay, Janine C Bennett, David Hollman, Sean Treichler, Patrick S McCormick, Christine M Sweeney, Hemanth Kolla, and Alex Aiken. Towards asynchronous many-task in situ data analysis using legion. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1033–1037. IEEE, 2016.
➢ Cited on page 42.

[95] Tom Peterka, Robert Ross, Attila Gyulassy, Valerio Pascucci, Wesley Kendall, Han-Wei Shen, Teng-Yok Lee, and Abon Chaudhuri. Scalable parallel building blocks for custom data analysis. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV 2011)*, pages 105–112. IEEE, 2011.
➢ Cited on page 42.

[96] N Pineau, L Soulard, L Colombet, T Carrard, A Pellé, Ph Gillet, and J Clérouin. Molecular dynamics simulations of shock compressed heterogeneous materials. ii. the graphite/diamond transition case for astrophysics applications. *Journal of Applied Physics*, 117(11):115902, 2015.
➢ Cited on page 54.

[97] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics*, 117(1):1–19, 1995.
➢ Cited on pages 33, 65, and 74.

[98] Arun Raman, Ayal Zaks, Jae W Lee, and David I August. Parcae: a system for flexible parallel execution. In *ACM SIGPLAN Notices*, volume 47, pages 133–144. ACM, 2012.
➢ Cited on page 38.

[99] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.
➢ Cited on page 28.

[100] E Schikuta. Message-passing-interface-forum: Mpi: A message-passing interface standard. *Techn. Ber., University of Tennessee, Knoxville, Tennesee*, 1994.
➢ Cited on page 30.

[101] H Bernhard Schlegel and Michael J Frisch. Transformation between cartesian and pure spherical harmonic gaussians. *International Journal of Quantum Chemistry*, 54(2):83–87, 1995.
➢ Cited on page 136.

[102] Will J Schroeder, Bill Lorensen, and Ken Martin. *The visualization toolkit: an object-oriented approach to 3D graphics*. Kitware, 2004.
➢ Cited on page 34.

[103] Ajeet Singh, Pavan Balaji, and Wu-chun Feng. Gepsea: a general-purpose software acceleration framework for lightweight task offloading. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 261–268. IEEE, 2009.
➢ Cited on page 37.

[104] L Soulard. Molecular dynamics study of the micro-spallation. *The European Physical Journal D*, 50(3):241–251, 2008.
➢ Cited on pages 33, 54, 57, and 135.

[105] L Soulard, N Pineau, J Clérouin, and L Colombet. Molecular dynamics simulations of shock compressed heterogeneous materials. i. the porous case. *Journal of Applied Physics*, 117(11):115901, 2015.
➢ Cited on page 34.

[106] Alexander Stukowski. Visualization and analysis of atomistic simulation data with ovito–the open visualization tool. *Modelling and Simulation in Materials Science and Engineering*, 18(1):015012, 2009.
➢ Cited on page 57.

[107] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.
➢ Cited on page 25.

[108] Rajeev Thakur and William Gropp. Test suite for evaluating performance of mpi implementations that support mpi_thread_multiple. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 46–55. Springer, 2007.
➢ Cited on page 31.

[109] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing mpi-io portably and with high performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32. ACM, 1999.
➢ Cited on page 32.

[110] David Van Der Spoel, Erik Lindahl, Berk Hess, Gerrit Groenhof, Alan E Mark, and Herman JC Berendsen. Gromacs: fast, flexible, and free. *Journal of computational chemistry*, 26(16):1701–1718, 2005.
➢ Cited on page 74.

[111] Jeffrey S Vetter. *Contemporary high performance computing: from Petascale toward exascale.* CRC Press, 2013.
➢ Cited on pages 32 and 120.

[112] Philippe Virouleau, François Broquedis, Thierry Gautier, and Fabrice Rastello. Using data dependencies to improve task-based scheduling strategies on numa architectures. In *European Conference on Parallel Processing*, pages 531–544. Springer, 2016.
➣ Cited on page 28.

[113] V. Vishwanath, M. Hereld, and M. E. Papka. Toward simulation-time data analysis and i/o acceleration on leadership-class systems. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 9–14, Oct 2011.
➣ Cited on pages 33 and 40.

[114] Edward E Zajac. Computer-made perspective movies as a scientific and communication tool. *Communications of the ACM*, 7(3):169–170, 1964.
➣ Cited on page 33.

[115] Yong Zhao, Mihael Hategan, Ben Clifford, Ian Foster, Gregor Von Laszewski, Veronika Nefedova, Ioan Raicu, Tiberiu Stef-Praun, and Michael Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Services, 2007 IEEE Congress on*, pages 199–206. IEEE, 2007.
➣ Cited on page 42.

[116] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. Predata - preparatory data analytics on peta-scale machines. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.
➣ Cited on page 39.

[117] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T. A. Nguyen, J. Cao, H. Abbasi, S. Klasky, N. Podhorszki, and H. Yu. Flexio: I/o middleware for location-flexible scientific data analytics. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 320–331, May 2013.
➣ Cited on pages 37, 40, and 78.

[118] Fang Zheng, Hasan Abbasi, Jianting Cao, Jai Dayal, Karsten Schwan, Matthew Wolf, Scott Klasky, and Norbert Podhorszki. In-situ i/o processing: a case for location flexibility. In *Proceedings of the sixth workshop on Parallel Data Storage*, pages 37–42. ACM, 2011.
➣ Cited on page 41.

[119] Fang Zheng, Hongfeng Yu, Can Hantas, Matthew Wolf, Greg Eisenhauer, Karsten Schwan, Hasan Abbasi, and Scott Klasky. Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 78:1–78:12, New York, NY, USA, 2013. ACM.
➣ Cited on pages 35, 76, 77, 80, 152, and 161.

[120] Bruno H Zimm. The scattering of light and the radial distribution function of high polymer solutions. *The Journal of Chemical Physics*, 16(12):1093–1099, 1948.
➣ Cited on page 68.

Integration of High-Performance Task-Based In Situ for Molecular Dynamics on Exascale Computer

The exascale era will widen the gap between data generation rate and the time to manage their output and analysis in a post-processing way, dramatically increasing the end-to-end time to scientific discovery and calling for a shift toward new data processing methods. The in situ paradigm proposes to analyze data while still resident in the supercomputer memory to reduce the need for data storage. Several techniques already exist, by executing simulation and analytics on the same nodes (in situ), by using dedicated nodes (in transit) or by combining the two approaches (hybrid). Most of the in situ techniques target simulations that are not able to fully benefit from the ever growing number of cores per processor but they are not designed for the emerging manycore processors. Task-based programming models on the other side are expected to become a standard for these architectures but few task-based in situ techniques have been developed so far.

This thesis proposes to study the design and integration of a novel task-based in situ framework inside a task-based molecular dynamics code designed for exascale supercomputers. We take benefit from the composability properties of the task-based programming model to implement the TINS hybrid framework. Analytics workflows are expressed as graphs of tasks that can in turn generate children tasks to be executed in transit or interleaved with simulation tasks in situ. The in situ execution is performed thanks to an innovative dynamic helper core strategy that uses the work stealing concept to finely interleave simulation and analytics tasks inside a compute node with a low overhead on the simulation execution time.

TINS uses the Intel® TBB work stealing scheduler and is integrated into ExaStamp, a task-based molecular dynamics code. Various experiments have shown that TINS is up to 40% faster than state-of-the-art in situ libraries. Molecular dynamics simulations of up to 2 billions particles on up to 14,336 cores have shown that TINS is able to execute complex analytics workflows at a high frequency with an overhead smaller than 10%.

Développement d'un Système In Situ à Base de Tâches pour un Code de Dynamique Moléculaire Classique Adapté aux Machines Exaflopiques

L'ère de l'exascale creusera encore plus l'écart entre la vitesse de génération des données de simulations et la vitesse d'écriture et de lecture pour analyser ces données en post-traitement. Le temps jusqu'à la découverte scientifique sera donc grandement impacté et de nouvelles techniques de traitement des données doivent être mises en place. Les méthodes *in situ* réduisent le besoin d'écrire des données en les analysant directement là où elles sont produites. Il existe plusieurs techniques, en exécutant les analyses sur les mêmes nœuds de calcul que la simulation (in situ), en utilisant des nœuds dédiés (in transit) ou en combinant les deux approches (hybride). La plupart des méthodes in situ traditionnelles ciblent les simulations qui ne sont pas capables de tirer profit du nombre croissant de cœurs par processeur mais elles n'ont pas été conçues pour les architectures many-cœurs qui émergent actuellement. La programmation à base de tâches est quant à elle en train de devenir un standard pour ces architectures mais peu de techniques in situ à base de tâches ont été développées.

Cette thèse propose d'étudier l'intégration d'un système in situ à base de tâches pour un code de dynamique moléculaire conçu pour les supercalculateurs exaflopiques. Nous tirons profit des propriétés de composabilité de la programmation à base de tâches pour implanter l'architecture hybride TINS. Les workflows d'analyses sont représentés par des graphes de tâches qui peuvent à leur tour générer des tâches pour une exécution in situ ou in transit. L'exécution in situ est rendue possible grâce à une méthode innovante de *helper core* dynamique qui s'appuie sur le concept de vol de tâches pour entrelacer efficacement tâches de simulation et d'analyse avec un faible impact sur le temps de la simulation.

TINS utilise l'ordonnanceur de vol de tâches d'Intel® TBB et est intégré dans ExaStamp, un code de dynamique moléculaire. De nombreuses expériences ont montrées que TINS est jusqu'à 40% plus rapide que des méthodes existantes de l'état de l'art. Des simulations de dynamique moléculaire sur des système de 2 milliards de particles sur 14,336 cœurs ont montré que TINS est capable d'exécuter des analyses complexes à haute fréquence avec un surcoût inférieur à 10%.

Communauté
UNIVERSITÉ Grenoble Alpes

FROM RESEARCH TO INDUSTRY
cea