N° d'ordre : xxxx

THÈSE

PRÉNTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Salli MOUSTAFA**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Massively Parallel Cartesian Discrete Ordinates Method for Neutron Transport Simulation

Soutenue le : 15 décembre 2015

Après avis des rapporteurs :

Michael A. HEROUX	Scientist, Sandia National Laboratories
Alain HÉBERT	Professor, École Polytechnique de Montréal

Devant la commission d'examen composée de :

Michael A. HEROUX	Scientist, Sandia National Laboratories	Examinateur
Alain HÉBERT	Professor, Ecole Polytechnique de Montréal	Examinateur
Raymond NAMYST	Professor, Université de Bordeaux	Président du jury
Laurent PLAGNE	Research scientist, EDF R&D	Encadrant industriel
Pierre RAMET	Assistant professor, Université de Bordeaux	Directeur de Thèse
Jean ROMAN	Professor, Université de Bordeaux	Directeur de Thèse

– 2015 –

Contents

Introduction

1

An Introduction to Neutron Transport Simulation and Parallel Architectures

1.1 Analysis of nuclear reactor cores . . . 6
  1.1.1 Basic concepts of nuclear reactor physics . . . 6
  1.1.2 Boltzmann transport equation . . . 7
1.2 Multigroup formulation of the transport equation . . . 10
1.3 Discrete ordinates method on Cartesian meshes . . . 12
  1.3.1 Angular discretization . . . 12
  1.3.2 Spatial discretization . . . 13
  1.3.3 Cartesian transport sweep operation . . . 15
1.4 Modern parallel computers and performance evaluation . . . 15
  1.4.1 Architecture and design of parallel computers . . . 17
  1.4.2 Metrics for performance evaluation . . . 18

2

On the Vectorization of the Sweep Operation

2.1 Review of the SIMD model . . . 22
  2.1.1 General presentation of the SIMD paradigm . . . 22
  2.1.2 On the way for vectorization . . . 25
  2.1.3 Generic programming and tools for the vectorization . . . 25
2.2 Arithmetic intensity of the sweep kernel . . . 26
  2.2.1 Memory traffic and flops per cell and per direction . . . 26
  2.2.2 General formula of the arithmetic intensity of the sweep . . . 29
2.3 Vectorization over spatial domain . . . 30
  2.3.1 The algorithm . . . 31
  2.3.2 Maximum theoretical speed-up . . . 32
2.4 Vectorization over the angular variable . . . 32

- 2.4.1 The algorithm . . . 32
- 2.4.2 Study of the arithmetic intensity of the sweep . . . 33
- 2.4.3 Actual performances vs Roofline model . . . 35

3
Performance Modelization of a Parallel Sweep

- 3.1 Preliminary definitions . . . 40
- 3.2 Literature review on performance models of the sweep . . . 42
  - 3.2.1 Flat models . . . 42
  - 3.2.2 Hybrid models . . . 44
- 3.3 New performance model of a parallel sweep . . . 44
  - 3.3.1 Computation steps . . . 45
  - 3.3.2 Communication steps . . . 48
- 3.4 Asynchronous simulator of the sweep . . . 50
  - 3.4.1 General presentation of the simulation algorithm . . . 51
  - 3.4.2 Communication costs . . . 53
- 3.5 Comparative study of the performance models . . . 55
  - 3.5.1 Parameters of the models . . . 55
  - 3.5.2 Evaluation of the performance models . . . 57

4
A Massively Parallel Implementation of the Cartesian Transport Sweep

- 4.1 The emergence of generic task-based runtime systems . . . 66
  - 4.1.1 The traditional MPI+X model . . . 66
  - 4.1.2 Task-based models . . . 66
- 4.2 Implementation of the sweep algorithm with INTEL TBB . . . 67
- 4.3 Implementation of the sweep algorithm with PARSEC . . . 68
  - 4.3.1 Task-graph of the sweep operation . . . 68
  - 4.3.2 Data distribution . . . 70
  - 4.3.3 Optimization of the scheduling through priorities . . . 71
- 4.4 Implementation of the sweep algorithm with STARPU . . . 71
- 4.5 Experiments . . . 75
  - 4.5.1 Task-granularity selection and parameters of the performance models . . . 75
  - 4.5.2 Shared memory performances . . . 76
  - 4.5.3 Distributed memory performances . . . 78

5
Full-core SN Calculations on Massively Parallel Architectures

5.1 Acceleration of source iterations (SI) 86
  5.1.1 Diffusion Synthetic Acceleration (DSA) method 86
  5.1.2 Piecewise DSA method (PDSA) 88
5.2 Validation and performances of Domino 91
  5.2.1 Benchmarks 91
  5.2.2 Validation and performances of the source iterations scheme 94
  5.2.3 Efficiency of the PDSA scheme 97
  5.2.4 Full-core 3D PWR calculations 101

Conclusion and Future Work

Appendixs

A
Presentation of the test platforms

B
Publications

Bibliography

List of Algorithms

1 Inverse Power Algorithm 10
2 Gauss-Seidel 11
3 Scattering iterations 13
4 Discretized algorithm of S_N method as implemented in DOMINO 16

5 Processing of a single spatial cell (in DD0), per angular direction. 27
6 The Sweep Operation 30

7 General simulation algorithm of an asynchronous sweep in 2D 52
8 Asynchronous sweep simulator 54

List of Tables

2.1 SIMD width as a function of the processor architecture. 23

2.2 Critical arithmetic intensities of Intel Xeon E78837 and Intel Xeon E52697 V2 processors. 24

2.3 Memory accesses required to perform a sweep over a `MacroCell` of size $n_x \times n_y \times n_z$ using a quadrature formula comprising D angular directions. 29

2.4 Impact of the padding on the vectorization efficiency in single precision. The sustainable speed-up gives the maximum attainable speed-up taking into account the padding. 33

2.5 Illustration of the speed-up brought by the angular vectorization of the sweep, as a function of the `MacroCell` size ($n_x = n_y = n_z$), and for different angular quadrature orders. The performance measurements were carried-out on an Intel Xeon E78837 processor. 35

2.6 Illustration of the speed-up brought by the angular vectorization of the sweep, as a function of the `MacroCell` size ($n_x = n_y = n_z$), and for different angular quadrature orders. The performance measurements were carried-out on an Intel Xeon E52697 V2 processor. 36

3.1 Notations used in the performance models of the sweep operation. 42

3.2 Characteristics of the test cases used. GFlops is the number of floating point operations required for one complete sweep. We count 25 floating point operations per spatial cell per angular direction (see section 1.3.3). 58

3.3 Optimal domain partitioning and aggregation factors for the Adams *et al.* model using `small` test case on the IVANOE platform. 59

3.4 Optimal domain partitioning and aggregation factors for the Hybrid model using `small` test case on the IVANOE platform. 60

4.1 Characteristics of the test cases used. GFlops is the number of floating point operations required for a single complete sweep. We count 25 floating point operations per spatial cell per angular direction (see section 1.3.3). 76

5.1 One-group cross-sections and source strength S for the KOBAYASHI benchmark. . 92

5.2 Description of benchmarks and calculation parameters. 94

5.3 S_N-only solution time for the Kobayashi problems $1ii$, on a single computing node (dual Intel Xeon E5-2697v2 processors) of the ATHOS platform. Solution times for DENOVO were obtained on 16 processors of the Jaguar XT5 supercomputer, and correspond to the case where weighted diamond difference (WDD) spatial discretization scheme is used [36]. 96

5.4 Discrepancies on the scalar flux, between DOMINO and reference solution extracted from [111], for the Model 1 of the TAKEDA benchmark. 97
5.5 Eigenvalue and computation time for the Model 1 of the TAKEDA benchmark, on a 24-cores computing node of the ATHOS platform. For this experiment, we fixed the number of SP_N iterations to $N_{\text{SPN}} = 4$. 99
5.6 Solution times for a S_{12} 2-group 3D PWR k_{eff} computation on the ATHOS platform. 101
5.7 DOMINO-DENOVO Comparison for a S_{12} 2-group 3D PWR k_{eff} computation. . . . 102
5.8 Solution times for a S_{12} 8-group 3D PWR k_{eff} computation. 102
5.9 Solution times for a S_{16} 26-group 3D PWR k_{eff} computation. 103

A.1 Characteristics of the target machines. 109

List of Figures

1.1 The fission chain reaction (illustration inspired by the CEA web site.) 6
1.2 Illustration of the sweep operation over a 6×6 2D spatial grid for a single direction. 17

2.1 Processing of a single spatial cell (in DD0) in 2D, with a single angular direction. 26
2.2 Sweep over a 5×4 2D spatial domain. It requires 20 scalar steps to go through the whole domain in scalar mode. 31
2.3 Vectorization of the sweep operation according to the spatial variable in 2D. . . . 31
2.4 Single precision arithmetic intensity as a function of the `MacroCell` size ($n_x = n_y = n_z$) and angular quadrature. 34
2.5 Roofline model for Intel Xeon E78837 processor and measured performances of the sweep operation as a function of the arithmetic intensity I_a. The variation of I_a is obtained by varying size of `MacroCells` as in Figure 2.4b. 37
2.6 Roofline model for a Intel Xeon E52697 V2 processor and performances of sweep operation as a function of the arithmetic intensity I_a. The variation of I_a is obtained by varying size of `MacroCells` as in Figure 2.4b. 38

3.1 Simultaneous sweep of all quadrants. This shows that in this example M_2 is equal to 2. 45
3.2 Illustration of the formula giving the number of computation steps, for a single quadrant sweep, over a 2D spatial grid of 6×6 `MacroCell`*s*. This sweep is performed on a dual-core computing node. 47
3.3 Verification of the computation steps formula for a concurrent sweep of all quadrants over a 6×6 `MacroCell` grid, using nine dual-core computing nodes distributed into a 3×3 process grid. Tasks of similar colors are on the same node and follow distribution on Figure 3.1. 48
3.4 Dependencies for the execution of a task in 2D. 53
3.5 Performance of the network interconnect of the IVANOE platform. Measurements are obtained using NetPIPE utility, compiled with OpenMPI 1.6.5. The two vertical lines define the range to which belong messages sent through the network for the benchmarks considered in this study. 55
3.6 Average performance of the sweep operation per `MacroCell`, as a function of the `MacroCell` size. This experiment was carried out using a sequential implementation of the sweep operation on the BIGMEM computing node. 56
3.7 Performance of the Hybrid model using the `small` test case on IVANOE. This figure illustrates how the Hybrid model integrates the overlap of communications by computations. 59

3.8 Comparison of Adams *et al.*, Hybrid and Hybrid-Async models using the `small` test case using parameters of the IVANOE platform. 61

3.9 Performance of the Hybrid model using the `big` test case on IVANOE. 61

3.10 Comparison of Flat, Hybrid and Simulator using the `big` test case on IVANOE . 62

3.11 Impact of the scheduling overhead on the `small` test case, using 64 computing nodes. Process grid partitioning and `MacroCell` sizes are fixed to the optimal values obtained from the run without any overhead (see Table 3.4). The vertical line corresponds to the computation time of a single task. 62

4.1 Animation snapshots showing the behavior of various scheduling strategies on the sweep progress throughout the spatial domain for one single direction. Data is distributed over a $2 \times 2 \times 2$ process grid. Threads of similar colors belong to the same node. Brightest colors are used to show what `MacroCells` are processed during the snapshots. 72

4.2 Single-core performances of the PARSEC, STARPU and INTEL TBB implementations of the sweep operation, averaged by the number of `MacroCells`: 25 (there are five `MacroCells` along each dimension). This experiment was conducted on a computing node of the ATHOS platform (Intel Xeon E5-2600 V2 processors). The angular discretization is a S_{16} Level Symmetric quadrature (288 directions). . . . 76

4.3 Performance of the network interconnect of the IVANOE and ATHOS platforms. Measurements are obtained using NetPIPE utility, compiled with OpenMPI 1.6.5. The two vertical lines define the range to which messages sent through the network belong to, for `small` and `big` test cases, according to task granularity. 77

4.4 Performances of the implementations of the sweep operation using `big` test case (see Table 3.2), on a single 24-cores computing node of the ATHOS supercomputer. All measurements were obtained using the same task-granularity of $20 \times 20 \times 20$ cells per task. 78

4.5 Sensitivity of Hybrid and Hybrid-Async performance models to data distributions using 48 nodes (576 cores) of the IVANOE cluster, and the `big` test case. Predicted performances by the Hybrid model and Hybrid-Async simulator are obtained using a scheduling overhead value of $\delta = 0$ s. 79

4.6 Comparison of *Hybrid* and *Flat* approaches using `small` test case on the IVANOE cluster. We used 64 nodes and 6 computations cores per node. The performance measurements were obtained using optimal partitioning, according to predictions of Hybrid and Adams *et al.* models. 80

4.7 Performance comparison of the PARSEC implementation of the sweep against predicted performances by Hybrid model and Hybrid-Async simulator. This experiment was conducted on the IVANOE cluster, using the `small` test case. Using $\delta = 8.1 \cdot 10^{-6}$ s, $o_c = 4.2 \cdot 10^{-10}$ s/Byte, the prediction of the Hybrid-Async complies with actual measurements. 81

4.8 Performance of the PARSEC implementation of the sweep using `big` test case on the IVANOE cluster. 82

5.1 RT0 finite element in 2D: 5 DoFs (4 for the current and 1 for the scalar flux). . . 88

5.2 Correspondance between flux and current DoFs. 89

5.3 Illustration of the PDSA method on a domain split in two. The first step consists of solving two diffusion problems in parallel on $\mathcal{D}_1$ and $\mathcal{D}_2$, with Neumann boundary conditions. The second step solves also two diffusion problems with non-homogeneous Dirichlet boundary conditions: null flux boundary conditions on the external boundary of the domain and an average value of the flux at the inner interface. 90
5.4 Illustration of the communication pattern in PDSA method on a domain split in two. Two point-to-point communications are needed to exchange flux at the interface between the two subdomains. 91
5.5 Configuration of the KOBAYASHI benchmark [66] (Problem 1). 92
5.6 Core configuration the TAKEDA benchmarks [111]. 92
5.7 Radial view of the PWR 900 MW model [24]. 93
5.8 Comparison of the neutron fluxes for the Kobayashi problem $1ii$ (50% of scattering), obtained from DOMINO, with reference values obtained from GMVP Monte-Carlo code [66]. 95
5.9 Convergence of the inverse power algorithm, without the acceleration, using TAKEDA benchmark. 97
5.10 Convergence of the PDSA scheme using a partitioning of $(1, 1, 1)$. 98
5.11 Convergence of the PDSA scheme as a function of $\rho^{\max}_{\text{PDSA}}$. 100
5.12 Convergence of the PDSA scheme using a partitioning of $(4, 4, 2)$. 100
5.13 Convergence of DOMINO using the 26-group PWR benchmark. We used 64 computing nodes partitioned into $(4, 4, 4)$. 103

Introduction

In today's world, the energy demand is constantly growing. Among the various sources of energy, the nuclear represents 11% of the overall total energy production[1]. Nuclear energy (actually electricity) is produced by the mean of nuclear power plants, using fissile fuel placed in reactor cores. Since the beginning of the nuclear industry in the early 1950s, nuclear reactor core operators, such as Electricité de France (EDF), conduct engineering studies to enhance safety and efficiency of the nuclear power plants. Indeed, before operating the reactor, safety reports require to precisely evaluate the location and the magnitude of the maximal power peak within the reactor core (pin peak power). This is mandatory in order to fulfill the regulatory hurdles. Efficiency concerns will require the evaluation of the reactor longest cycle length as possible, while maximizing the nominal power (see [106] for more details). In order to meet these goals, nuclear engineering studies require to achieve high-fidelity predictive nuclear reactor core simulations. These simulations involve coupled multi-physics calculations that encompass thermal-hydraulics and neutronic studies. In particular, the neutronic studies, or neutron transport calculations, on which we focus in this thesis, consist in describing precisely the neutron flux distribution inside the reactor core. This flux, which corresponds to the neutron phase-space density, depends on seven variables: three in space ($\overrightarrow{r} = (x, y, z)^t$), one in energy ($E$), two in direction ($\overrightarrow{\Omega} \equiv (\theta, \varphi)$) and one in time ($t$). Thus, the precise simulation of the neutron flux distribution in the reactor core would require a tremendous computational effort.

Indeed, let us consider the core of a Pressurized Water Reactor (PWR) 900 MW. From the presentation in [101], we make the following observations.

- There are 157 fuel assemblies in the core, each of which being formed of 289 pin-cells. Each pin-cell is composed of $\mathcal{O}(10)$ radial zones and of $\mathcal{O}(50)$ axial zones.
- The large variations of the energetic spectrum of neutrons requires considering $\mathcal{O}(2 \cdot 10^4)$ spatial mesh points.
- We must consider $\mathcal{O}(10^2)$ angular directions for describing the traveling directions of the neutrons.

It follows that $\mathcal{O}(10^{12})$ values of the flux should be evaluated for each timestep, which represents a very large amount of data. Thereby, to cope with daily industrial calculations, a two-step homogenization approach [106] is generally employed. The first step of this approach leads to the evaluation of homogenized cross-sections using a *lattice code*, while the second step consists in evaluating the neutron flux in the whole reactor core using a *core solver*. The core solver is generally based on a simplified model (*e.g.* diffusion) of the exact neutron transport problem, and thus in fact comprises modelization errors. In order to quantify these modelization errors, existing in core industrial solvers, it necessary to solve the exact Boltzmann Transport Equation (BTE) with a *reference solver*. Two main families of methods are used to perform reference calculations: probabilistic and deterministic methods. The first ones consist in using statistical tools to simulate the history of a large number of neutrons in the core, taking into account all the interactions that may occur between neutrons and the matter in the core. Hence, probabilistic methods enable to avoid the phase-space discretization problems, and can handle complex geometries, which make them very attractive for reactor physics analysis [12, 14, 105]. Unfortunately, the convergence of probabilistic methods allows to estimate the phase-space density with an accuracy that converges rather slowly. The other class of methods, reference deterministic methods, are computationally very demanding because they require a full discretization of the BTE to achieve acceptable levels of accuracy. For this reason, until a few years ago these methods were impracticable for 3D cases because of the limitations on the computing power.

---

[1] `http://www.nei.org/Knowledge-Center/Nuclear-Statistics/World-Statistics`

Computer resources have since grown in capability, and several research works are therefore being conducted in developing reference 3D deterministic solvers [28, 62, 106, 116]. These are based on either discrete ordinates (S_N), or spherical harmonic (P_N), or Method of Characteristics (MOC) which is a special case of S_N. All these methods share the same energetic discretization, and differ on angular and spatial discretizations. The S_N method consists in considering only a finite set of angular flux components $\psi(\vec{r}, E, \vec{\Omega}_i)$ that corresponds to a finite set of carefully selected angles $\vec{\Omega}_i$. In the P_N method, the transport equation is projected onto a set of spherical harmonics, allowing to mitigate the fundamental shortcoming present in the S_N method: the "ray-effects". The MOC uses the same energetic and angular discretizations as the S_N method, but its spatial discretization is based on the long characteristics. This discretization enables the MOC to deal with the heterogeneity and complexity of the reactor core like probabilistic methods. Although the MOC has been proven to be very efficient in 2D, large-scale 3D cases are still very computationally demanding [106].

In this thesis, we consider the problem of solving accurately and efficiently the steady-state BTE using the S_N method on Cartesian mesh. As previously mentioned, this reference method enables the validation of approximate core industrial solvers. In particular, during the reactor core refueling process, it must be guaranteed that the optimization of a new fuel-loading pattern (see [118]) can be done in a short period of time [106]. Consequently, a reference calculation, involving several coupled multi-physics iterations, has to be completed in a limited amount of time. It is therefore necessary to manage to develop highly-optimized algorithms and numerical methods to tackle the large amount of calculations required by the S_N method.

In front of this problem, the landscape of today's parallel computers, on which the simulations have to be executed, has dramatically shifted since the end of frequency scaling of monolithic processors, which has motivated the advent of the multicore processor in the early 2000s [41]. Modern clusters are composed of heterogeneous computing nodes. These computing nodes are equipped with processors having tens of CPU cores, capable of issuing vector instructions on wide registers. A key point to note here is that the off-chip bandwidth, determining the latency of memory accesses between the computer main memory and the CPU registers, is not growing as fast as the computing power of these CPUs [81], leading to a "memory wall". The consequence of this is to put a dramatic emphasis on the sustainable peak-performance per core, since the gap between this metric and the theoretical peak performance trends to increase for *memory bound* applications. Furthermore, computing nodes may comprise accelerators like Graphics Processing Units (GPUs) and manycore devices. This represents a huge amount of computing power that can be used to tackle large numerical simulations in science and engineering such as predictive nuclear reactor core simulations. As a consequence of this hardware evolution, several projects are dedicated in the scientific community to the improvement of the performance of numerical simulation codes. Hence, one solution for maximizing the sustainable peak performance of a computational kernel is to rely on highly optimized external numerical libraries. For instance, the Trilinos project [52] aims at providing optimized packages, developed with advanced object-oriented techniques and state-of-the-art parallel programming paradigms, which can therefore be directly used as building blocks for solving large-scale and complex multi-physics problems. For some specialized applications, as the neutron transport simulation problem, the solution for achieving high-performance on emerging architectures is oriented towards computer code modernization as justified by the large number of recent researches on this subject [11, 53, 60, 83, 84, 97, 100, 113]. This initiative is further promoted by hardware vendors such as the "Intel code modernization enablement program"[1] which provides tools and guidelines for the development of state-of-the-art and cutting-edge numerical simulation tools, capable of scaling on exascale machines. Moreover, the Exa2CT[2] project aims at facilitating the development of highly optimized scientific codes, capable to scale on exascale machines, through the development of new algorithms and programming techniques, validated on proto-applications, and that can be directly re-integrated into parent computational codes.

In the particular case of neutron transport simulations, a lot of efforts have been dedicated for improving the efficiency of the discrete ordinates method. These efforts can be classified in two fields: numerical methods and parallelization strategies that were developed accordingly. The most computationally demanding portion in the S_N method is the space-angle problem called sweep operation, for each energy group. This operation acts like a wavefront propagating throughout the spatial domain, according to the angular direction. Therefore, the successive parallel algorithms developed for the S_N method focus on the parallelization of this sweep operation. The first parallel sweep algorithm KBA [8] is implemented in numerous early S_N solvers [28, 56]. KBA splits the 3D spatial grid on a 2D process grid, enabling each process to perform the sweep on a local subdomain in a classical fork-join mode. The efficiency of this algorithm has been extensively studied in the literature through performance models [39, 55, 64, 108]. Moreover, the advent of modern massively parallel computers has motivated extensions of this algorithm to provide sufficient concurrency for an improved scalability on large number of cores. In the UNIC code system [63], the authors implemented a parallel decomposition over space, energy and angle, enabling to extract more parallelism. A similar approach has been recently introduced in the DENOVO code [34], where a new multilevel parallel decomposition allows concurrency over energy in addition to the space-angle. Furthermore, the PDT code [1] implements an extension of the KBA algorithm that enables a decomposition of the 3D spatial grid over a 3D process grid. These approaches enable to tackle neutron transport problems featuring larger number of energy groups. However, the parallelism in these codes adopts a uniform view of a supercomputer as collection of distributed computing cores (*Flat* approach), without explicitly addressing all the hierarchical parallelism provided by the modern architectures.

Recently, in the neutron transport community, some research initiatives have emerged to cope explicitly with the hierarchical topology (cluster of multiprocessor with multiple cores) of modern architectures. These initiatives are conducted through the development of proto-applications. For instance, the SNAP code, as a proxy application to the PARTISN [9] code focuses on exploring a *Hybrid* programming model for the sweep operation and auto-vectorization capabilities. Similarly, KRIPKE is a proxy application to the ARDRA [68] radiation transport code, that is being developed to study the implications of new programming models and data layouts on its parent code performances.

---

[1] https://software.intel.com/en-us/code-modernization-enablement

[2] http://www.exa2ct.eu/index.html

As we have seen, the neutron transport simulation is a large and complex research area, in an international competitive environment. We record ourselves in this area and, in this thesis, we give some original contributions to the field. Indeed, we considered all the issues associated with the emerging architectures, that encompass the vectorization (SIMD), multithreading and message-passing paradigms. We make a key point on maximizing the sustainable peak performance of our implementation on distributed multicore-based machines using emerging software tools and frameworks (task-based runtime systems) that enable to reach higher performance, on today platforms, and the performance portability on future exascale architectures. More precisely we followed the trends on code modernization, and we developed highly-scalable parallel algorithms for the Cartesian S_N neutron transport simulations over distributed multicore-based architectures. We especially tailored the discrete ordinates method to explicitly use all levels of parallelism available on these architectures. Furthermore, our approach relies at each level on theoretical performance models. In Chapter 1, we present background on reactor physics, the neutron transport equation, and we describe the sweep operation. Then, in Chapter 2, we discuss the vectorization strategies for maximizing the single-core peak performance of the sweep operation, and we justify our approach with a theoretical performance model. In Chapter 3, we discuss the design and theoretical performances of a new efficient parallel *Hybrid* sweep algorithm, targeting multicore-based architectures, using a new performance model which extends existing ones. In Chapter 4, we present an implementation of our *Hybrid* sweep algorithm using emerging task-based models on top of generic runtime systems. The integration of this sweep algorithm into our S_N solver DOMINO is then presented in Chapter 5. A key point in this chapter is the implementation of a new efficient piece-wise acceleration method required to speed-up the convergence of the S_N method in strongly-diffusive media such as PWR cores. The efficiency of this acceleration method is well adapted to optically thick domains and particularly adapted to our *Hybrid* implementation. We conclude giving a general summary of this work and perspectives for future research on the field.

Chapter 1

An Introduction to Neutron Transport Simulation and Parallel Architectures

Contents

1.1 Analysis of nuclear reactor cores 6
    1.1.1 Basic concepts of nuclear reactor physics 6
    1.1.2 Boltzmann transport equation 7
1.2 Multigroup formulation of the transport equation 10
1.3 Discrete ordinates method on Cartesian meshes 12
    1.3.1 Angular discretization 12
    1.3.2 Spatial discretization 13
    1.3.3 Cartesian transport sweep operation 15
1.4 Modern parallel computers and performance evaluation 15
    1.4.1 Architecture and design of parallel computers 17
    1.4.2 Metrics for performance evaluation 18

In this chapter, we introduce the neutron transport simulation as a requirement for enhancing the safety, efficiency and design of nuclear reactor cores. We start by presenting how neutrons inside a nuclear reactor core are modeled and the fundamental equation governing this process: the Boltzmann transport equation. Then, we derive the eigenvalue form of this equation that is used to determine the criticality of a reactor core. After that, we present the multigroup approximation that is used for the discretization of the energy variable and we introduce the discrete ordinates method, and the sweep operation that is used for solving the space-angle problem. Finally, we give an introduction to the modern parallel architectures on which the simulations will have to be executed.

1.1 Analysis of nuclear reactor cores

This section presents a general view on the reactor physics engineering and the mathematical model describing the neutron transport in the reactor core. This presentation is inspired from the works in [30, 76].

1.1.1 Basic concepts of nuclear reactor physics

A nuclear power plant is an industrial facility dedicated to electricity production, from the energy generated by the fissions of heavy nuclei (*e.g.* ^{235}U or ^{239}Pu) taking place in the nuclear reactor core. Each fission is induced by a neutron and releases an average energy of about 200 MeV [30] in the form of heat, and some additional neutrons (2 or 3 on average) which can in turn induce other fissions, hence leading to a chain reaction (Figure 1.1). Therefore, the fission energy increases the temperature of the water[1] circulating in the reactor core, also called a coolant. This high temperature water, by means of a heat exchanger, generates high pressure and high temperature steam from low temperature water. Finally, the steam is used to run turbogenerators producing the electricity. Hence, the more neutron fission, the more energy is released by the reactor. However, the fission probability of a nucleus is higher if the speed of the incident neutron is slow. Thus, in order to increase this fission probability, a moderator is generally used to slow down the neutrons[2]. For a Pressurized Water Reactor (PWR), on which we will focus for the remainder of this dissertation, the coolant also acts as a moderator.

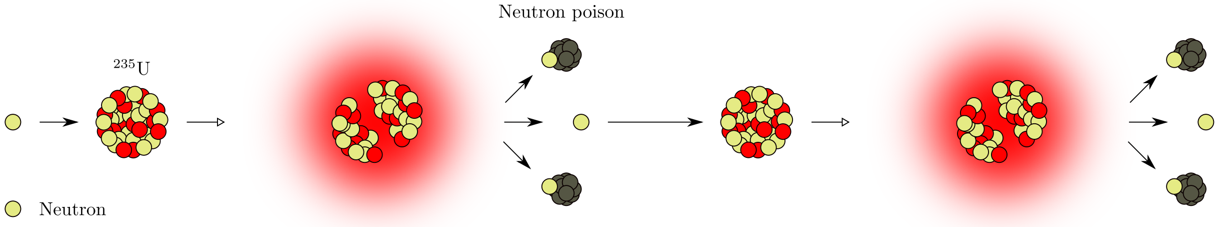


Figure 1.1: The fission chain reaction (illustration inspired by the CEA web site.)

As we have seen, the neutrons play an important role in the analysis of nuclear reactor cores, because they determine the fission process and thus the power generated by a nuclear reactor [76]. In addition, the neutron population of the core is a fundamental information. This information is used to conduct safety studies and optimization of both fuel reload and reactor core design. It is therefore of great interest to evaluate precisely the neutron population (or neutron flux distribution) in each region of the reactor core $\overrightarrow{r}$, at a given time t, as a function of the neutron's energy E, and according their propagation direction $\overrightarrow{\Omega}$. This represents therefore a complicated problem in which the phase-space has 7 dimensions.

---

[1] 330 °C in the case of PWR 900 MW. The coolant is maintained in a liquid state thanks to a high pressure (10 MPa) maintained in the core by the pressurizer.

[2] This process is known as thermalization or moderation of the neutrons.

The distribution of neutrons inside the core is influenced by the different types of nuclear reactions that neutrons can undergo. As already mentioned, a neutron when captured by a heavy nucleus can induce a fission of the nucleus into lighter nuclei. This fission results in more neutrons, energy release and some fission products. In addition, a neutron can simply be captured (pure absorption) by the nucleus without inducing a fission; or it can scatter off (bounce off) the nucleus. Finally the neutron can leak outside the reactor. Hence, evaluating the neutron distribution inside the core requires taking into account all these reactions all together. There are essentially two main families of methods used to simulate the neutron population: the probabilistic methods (Monte-Carlo) and the deterministic ones. The former consist of using statistical tools to characterize the life of a neutron from its "birth" to "death" (absorption by a nucleus or leaking out of the reactor), through a simulation of the history of a large number of particles (neutrons). The main advantage of the Monte-Carlo methods is that they avoid the phase-space mesh problems, because no discretization is required. Unfortunately, probabilistic methods allow phase-space density estimation with an accuracy that converges rather slowly with the number N of particles ($\propto 1/\sqrt{N}$). The latter, deterministic methods, rely on finding a numerical solution of a mathematical equation describing the flow of neutrons inside the core. The fundamental equation governing the flow of neutrons is the Boltzmann Transport Equation (BTE), also called the neutron transport equation.

1.1.2 Boltzmann transport equation

In a nuclear reactor core, there are basically two different situations with regard to the interaction of a neutron with the nuclei present in the core.

- The neutron interacts with no nuclei; it therefore moves spatially at the same speed, without shifting from its initial direction: transport without collision.

- The neutron interacts with a nucleus. In this case, it can either be scattered by the nucleus, that is by changing its direction and its energy, or cause the fission of the target nucleus.

Therefore, the neutron transport equation or linear Boltzmann Transport Equation (BTE) is obtained by establishing a balance between arrivals and migrations of neutrons at the spatial position $\overrightarrow{r}$, traveling with an energy E (or a speed v), toward direction $\overrightarrow{\Omega}$ at a given time t. It is presented in equation (1.1).

$$\begin{aligned}
\frac{1}{v}\frac{\partial \psi}{\partial t}(\vec{r}, E, \vec{\Omega}, t) = &- \overbrace{\vec{\Omega}\cdot\vec{\nabla}\psi(\vec{r}, E, \vec{\Omega}, t)}^{\text{Transport}} \\
&- \overbrace{\Sigma_t(\vec{r}, E, t)\psi(\vec{r}, E, \vec{\Omega}, t)}^{\text{Collision}} \\
&+ \overbrace{\int_0^\infty dE' \int_{S_2} d\vec{\Omega}'\Sigma_s(\vec{r}, E' \to E, \vec{\Omega}'\cdot\vec{\Omega}, t)\psi(\vec{r}, E', \vec{\Omega}', t)}^{\text{Scattering}} \\
&+ \overbrace{\frac{\chi(E)}{4\pi}\int_0^\infty dE' \int_{S_2} d\vec{\Omega}'\nu\Sigma_f(\vec{r}, E', t)\psi(\vec{r}, E', \vec{\Omega}', t)}^{\text{Fission}} \\
&+ \overbrace{S_{\text{ext}}(\vec{r}, E, \vec{\Omega}, t)}^{\text{External source}}, \qquad (1.1)
\end{aligned}$$

where:

$\Sigma_t(\vec{r}, E, t)$ is the total cross-section. It characterizes the probability that a neutron of energy E interacts with a nucleus at the position $\vec{r}$.

$\Sigma_s(\vec{r}, E' \to E, \vec{\Omega}'\cdot\vec{\Omega}, t)$ is the scattering cross-section. It characterizes the probability that a neutron of energy E and direction $\vec{\Omega}$ will be scattered by a nucleus and result in an outgoing neutron of energy E' and direction $\vec{\Omega}'$.

$\Sigma_f(\vec{r}, E, t)$ is the fission cross-section. It characterizes the probability that an interaction between a neutron of energy E with a nucleus at position $\vec{r}$ results in a fission.

ν is the average number of neutrons produced per fission.

$\chi(E)$ is the fission spectrum. It defines the density of neutrons of energy E produced from fission.

$S_{\mathbf{ext}}(\vec{r}, E, \vec{\Omega}, t)$ is an external source of neutrons. It is generally used to start the chain reaction.

S_2 is the unit sphere.

Two different boundary conditions (BC) are generally used in reactor physics applications: vacuum and reflective boundary conditions. They are respectively defined in equation (1.2) and equation (1.3).

$$\text{Vacuum BC:} \quad \psi(\vec{r}, E, \vec{\Omega}, t) = 0 \text{ when } \vec{\Omega}\cdot\vec{n} < 0, \qquad (1.2)$$

$$\text{Reflective BC:} \quad \psi(\vec{r}, E, \vec{\Omega}, t) = \psi(\vec{r}, E, \vec{\Omega}', t) \text{ when } \vec{\Omega}\cdot\vec{n} < 0 \text{ and } \vec{\Omega}\cdot\vec{n} = -\vec{\Omega}'\cdot\vec{n}, \qquad (1.3)$$

where Γ is the boundary of the computational domain (the reactor core in our case), and $\vec{n}$ an outward normal to Γ.

Vacuum BC represents the case where there is no incoming neutron from the outside of the core and is used when the full description of the core is given. Reflective BC is used to take advantage of the symmetries presented by the physical problem. In this dissertation, we will consider only the vacuum BC.

Let $\mathcal{H}$ and $\mathcal{F}$ be the transport and fission operators as defined by the following equations (1.4) and (1.5):

$$\begin{aligned}\mathcal{H}\psi(\vec{r},E,\vec{\Omega},t) = \vec{\Omega}\cdot\vec{\nabla}\psi(\vec{r},E,\vec{\Omega},t) + \Sigma_t(\vec{r},E,t)\psi(\vec{r},E,\vec{\Omega},t)\\ -\int_0^\infty dE' \int_{S_2} d\vec{\Omega}'\Sigma_s(\vec{r},E'\rightarrow E,\vec{\Omega}'\cdot\vec{\Omega},t)\psi(\vec{r},E',\vec{\Omega}',t),\end{aligned} \tag{1.4}$$

$$\mathcal{F}\psi(\vec{r},E,t) = \frac{\chi(E)}{4\pi}\int_0^\infty dE'\int_{S_2} d\vec{\Omega}'\nu\Sigma_f(\vec{r},E',t)\psi(\vec{r},E',\vec{\Omega}',t). \tag{1.5}$$

Then, the equation (1.1) becomes:

$$\frac{1}{v}\frac{\partial\psi}{\partial t}(\vec{r},E,\vec{\Omega},t) = -\mathcal{H}\psi(\vec{r},E,\vec{\Omega},t) + \mathcal{F}\psi(\vec{r},E,t) + S_{\text{ext}}(\vec{r},E,\vec{\Omega},t). \tag{1.6}$$

For a nuclear reactor core under normal operating conditions, we seek to have a self sustained chain reaction in the absence of external sources of neutrons; this corresponds to an equilibrium state between fission neutron production and migrations of neutrons by transport or collision. In order to determine the state of the core in such a situation, it is necessary to find a nonnegative stationary solution of the Boltzmann equation (1.6) without the external source S:

$$\mathcal{H}\psi(\vec{r},E,\vec{\Omega}) = \mathcal{F}\psi(\vec{r},E).$$

However, for any given set of cross-sections, there generally exist no stationary solution to the BTE [76]. One way of transforming the previous equation so that it accepts a nonnegative solution is to adjust the average number of neutron procuced per fission, ν, so that a global time-independant balance can be preserved. Hence, we replace ν by ν/k and we obtain the following generalized eigenvalue problem:

$$\mathcal{H}\psi(\vec{r},E,\vec{\Omega}) = \frac{1}{k}\mathcal{F}\psi(\vec{r},E,\vec{\Omega}), \tag{1.7}$$

for which there will be a largest value of k such that a nonnegative solution exists. The largest such eigenvalue, which is also the spectral radius of operator $\mathcal{F}^{-1}\mathcal{H}$, is called the effective multiplication factor, and denoted k_{eff}. Physically, this coefficient allows to determine the criticality of the reactor core.

- If $k_{\text{eff}} < 1$, $(\mathcal{H}\psi(\vec{r},E,\vec{\Omega}) > \mathcal{F}\psi(\vec{r},E,\vec{\Omega}))$, the neutron production from fissions is less than migrations: the chain reaction turns off, and the reactor is said to be *subcritical*.
- If $k_{\text{eff}} > 1$, $(\mathcal{H}\psi(\vec{r},E,\vec{\Omega}) < \mathcal{F}\psi(\vec{r},E,\vec{\Omega}))$, the neutron production from fissions is larger than migrations: the reactor is said to be *supercritical*.
- If $k_{\text{eff}} = 1$, there is a strict equilibrium between production and migration of neutrons: the reactor is said to be *critical*.

problem (1.7) is solved using an inverse power algorithm, which leads to the computation of the neutron flux ψ and the eigenvalue k, by iterating on the fission term as presented in equation (1.8).

$$\mathcal{H}\psi^{n+1} = \frac{1}{k^n}\mathcal{F}\psi^n, \quad k^{n+1} = k^n\sqrt{\frac{<\mathcal{F}\psi^{n+1},\mathcal{F}\psi^{n+1}>}{<\mathcal{F}\psi^n,F\psi^n>}}. \tag{1.8}$$

Algorithm 1: Inverse Power Algorithm

```
   Input  : $\psi^{\mathbf{0}}$, $\nu\Sigma_f$, $\Sigma_s$, $\Sigma_t$
   Output: $k_{\text{eff}}$, $\psi$
1  $\mathbf{S} = \mathcal{F}\psi$;
2  while $\frac{k-k^{old}}{k^{old}} \geq \epsilon_k$ or $\frac{||\mathbf{S}-\mathbf{S}_{old}||}{||\mathbf{S}_{old}||} \geq \epsilon_\psi$ do
3      $\mathcal{H}\psi = \mathbf{S}$;
4      $\mathbf{S}_{\text{old}} = \mathbf{S}$;
5      $\mathbf{S} = \mathcal{F}\psi$;
6      $k = \sqrt{\frac{<\mathbf{S},\mathbf{S}>}{<\mathbf{S}_{\text{old}},\mathbf{S}_{\text{old}}>}}$;
7      $\mathbf{S} = \frac{1}{k}\mathbf{S}$;
```

Algorithm 1 describes the continuous form of the power iterations, also called external iterations. The power algorithm converges slowly near the criticality. We therefore use the Chebyshev polynomials to accelerate its convergence. This is done by evaluating the neutron flux at iteration $n+1$ as a linear combination of the solutions obtained in the last 3 iterations [104].

Each iteration of the power algorithm involves an inversion of the transport operator (Line 3 of Algorithm 1) and a source computation (Line 5 of Algorithm 1) which requires a discretization of the transport and fission operators. The next section describes the energetic discretization of these operators.

1.2 Multigroup formulation of the transport equation

The discretization of the energy variable is realized according to the multigroup formalism as described in [76]. In this formalism, the energy domain is split into a finite set of energy intervals, called *energy groups* and delimited by a decreasing sequence of carefully selected energy values $E_0, E_1, \ldots, E_G$. The multigroup formulation of the BTE is therefore obtained by integrating equation (1.7) one each of the G energy groups $[E_g, E_{g-1}]$ in turn. The resulting multigroup problem is represented by the linear system (1.9):

$$\begin{pmatrix} H_{11} & H_{12} & \cdots & H_{1G} \\ H_{21} & H_{22} & \cdots & H_{2G} \\ \vdots & \vdots & \ddots & \vdots \\ H_{G1} & H_{G2} & \cdots & H_{GG} \end{pmatrix} \begin{pmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_G \end{pmatrix} = \begin{pmatrix} S_1 \\ S_2 \\ \vdots \\ S_G \end{pmatrix}, \tag{1.9}$$

where:

$$\psi_g(\vec{r}, \vec{\Omega}) = \int_{E_g}^{E_{g-1}} \psi(\vec{r}, E, \vec{\Omega}) dE, \tag{1.10}$$

is the the multigroup angular flux of group g, and

$$\begin{aligned} H_{gg}\psi_g(\vec{r}, \vec{\Omega}) &= \vec{\Omega} \cdot \vec{\nabla}\psi_g + \Sigma_t^g(\vec{r}, \vec{\Omega})\psi_g(\vec{r}, \vec{\Omega}) - \int_{S_2} d\vec{\Omega}' \Sigma_s^{g \to g}(\vec{r}, \vec{\Omega}')\psi_g(\vec{r}, \vec{\Omega}' \cdot \vec{\Omega}), \\ H_{gg'}\psi_{g'}(\vec{r}, \vec{\Omega}) &= -\int_{S_2} d\vec{\Omega}' \Sigma_s^{g' \to g}(\vec{r}, \vec{\Omega}' \cdot \vec{\Omega})\psi_{g'}(\vec{r}, \vec{\Omega}'), \quad g \neq g' \\ S_g(\vec{r}) &= \frac{1}{k_{\text{eff}}^n} \frac{\chi_g}{4\pi} \sum_{g'=1}^{G} \int_{S_2} \nu\Sigma_f^{g'}(\vec{r})\psi_{g'}^n(\vec{r}, \vec{\Omega}')d\vec{\Omega}', \end{aligned} \tag{1.11}$$

define the multigroup equations. The definition of the multigroup flux ψ_g in equation (1.10) is justified by the energy separability hypothesis: the angular flux within the group g is approximated as a product of a function f and ψ_g as defined by equation (1.12).

$$\psi(\vec{r}, E, \vec{\Omega}) \approx f(E)\psi_g(\vec{r}, \vec{\Omega}), \quad \text{such that} \int_{E_g}^{E_{g-1}} f(E)dE = 1. \tag{1.12}$$

In the multigroup formulation (equation (1.11)), the neutron parameters (cross-sections and fission spectrum), are group-wise defined. A comprehensive description of the evaluation of multigroup cross-sections can be found in [17, 86]. We recall that the multigroup scattering cross-sections $\Sigma_s^{g'\to g}$ are expanded on the Legendre polynomials basis (see [49]). In the following, for the sake of clarity, all the numerical algorithms will be presented only for the order 0, corresponding to the case of an *isotropic* collision. In this case, the scattered neutron has a uniform probability to go along all directions in S_2, and the scattering cross-section is given by:

$$\Sigma_s^{g'\to g}(\vec{r}, \vec{\Omega}\cdot\vec{\Omega}') = \frac{1}{4\pi}\Sigma_{s0}(\vec{r}). \tag{1.13}$$

The resolution of the multigroup problem (1.9) is done using a block Gauss-Seidel (GS) algorithm as presented in Algorithm 2. It is worth noting here that the convergence of this GS algorithm depends on the sparsity profile of the transport matrix, which is determined by the scattering of the problem (presence of the coefficient $\Sigma_s^{g'\to g}$ in equation (1.11)). As the up-scattering[1] is only possible for some thermal groups[2] (see [76]), almost all the non-zero elements of the transport matrix are located in its lower triangular part, including the diagonal.

Algorithm 2: Gauss-Seidel

```
Input  : $\psi$, $\mathbf{S}$
Output: $\psi$
while Non convergence do
    for $g \in [\![1, G]\!]$ do
        $\mathbf{Q} = \mathbf{S_g} - \sum_{g'\neq g} H_{gg'}\psi_{g'}(\vec{r}, \vec{\Omega})$;
        $H_{gg}\psi_g(\vec{r}, \vec{\Omega}) = \mathbf{Q}$;
    ▷ Compute stopping criterion
```

Hence, it is not necessary for the GS algorithm to iterate for all energy groups: the resolution is direct for all fast groups, using a forward substitution, and the GS iteration apply only for the thermal groups. Each one of the GS iterations involves resolutions of G one-group space-angle problems:

$$H_{gg}\psi_g^{m+1}(\vec{r}, \vec{\Omega}) = -\sum_{g'\neq g} H_{gg'}\psi_{g'}^{m}(\vec{r}, \vec{\Omega}) + S_g(\vec{r}), \quad g = 1, \dots, G. \tag{1.14}$$

We refer to these one-group problems as monokinetic equations. One should note here that, for each Gauss-Seidel iteration, the resolutions of all the monokinetic equations are necessarily done sequentially. This is an intrinsic property of the Gauss-Seidel algorithm. In order, for example, to parallelize the resolutions of these equations, one could use for example the Block-Jacobi algorithm as presented in [28]. But as our target applications do not feature many energy groups (less than 26), and because the Jacobi iterations converge two times slower than compared to Gauss-Seidel [75], we would rather keep using the Gauss-Seidel algorithm.

---

[1] There is an up-scattering if a neutron of energy g is scattered into a neutron of $g' > g$.

[2] Thermal groups are those having lowest energy. Fast groups are those having highest energy.

There are two major classes of resolution methods of monokinetic equations: P_N and S_N methods. The P_N method consists of expanding the angular flux and the scattering cross-section on a truncated Legendre polynomials basis, whereas the discrete ordinates method (S_N) consists of discretizing the angular variable on a finite number of directions. We will focus on the latter method in the rest of this dissertation.

1.3 Discrete ordinates method on Cartesian meshes

In this section we present the discretizations of the angular and spatial variables of the monokinetic equations.

1.3.1 Angular discretization

Let us consider the monokinetic transport equation (1.14), on which both group indices and iteration indices are removed to simplify the notations. The angular dependency of this equation is resolved by looking for solutions on a discrete set of carefully selected angular directions $\{\vec{\Omega}_i \in S_2,\, i = 1, 2, \cdots, N_{\text{dir}}\}$, called discrete ordinates:

$$\underbrace{\vec{\Omega}_i \cdot \vec{\nabla}\psi(\vec{r}, \vec{\Omega}_i) + \Sigma_t(\vec{r}, \vec{\Omega}_i)\psi(\vec{r}, \vec{\Omega}_i)}_{L\psi(\vec{r},\vec{\Omega}_i)} - \overbrace{\int_{S_2} d\vec{\Omega}'\Sigma_s(\vec{r}, \vec{\Omega}' \cdot \vec{\Omega}_i)\psi(\vec{r}, \vec{\Omega}')}^{R\psi(\vec{r},\vec{\Omega}_i)} = Q(\vec{r}, \vec{\Omega}_i) \quad \forall i, \tag{1.15}$$

where $Q(\vec{r}, \vec{\Omega}_i)$ gathers monogroup fission and inter-group scattering sources. Basically, the choice of the discrete ordinates is the same as to uniformly distribute a finite number of points on the unit sphere; which is not a trivial task because of the curvature of the sphere. In general, these discrete ordinates are determined thanks to a numerical quadrature formula. In DOMINO, we use the *Level Symmetric* quadrature formula, which leads to $N_{\text{dir}} = N(N + 2)$ angular directions, where N stands for the order of the *Level Symmetric* quadrature formula. Each angular direction is associated to a weight w_j for integral calculation on the unit sphere S_2, such that:

$$\int_{S_2} g(\vec{\Omega})d\vec{\Omega} \simeq \sum_{j=1}^{N_{\text{dir}}} w_j g(\vec{\Omega}_j), \tag{1.16}$$

for any function g summable over S_2. Hence, the scattering term $R\psi(\vec{r}, \vec{\Omega}_i)$ in equation (1.15) becomes:

$$R\psi(\vec{r}, \vec{\Omega}_i) \simeq \sum_{j=1}^{N_{\text{dir}}} w_j \Sigma_s(\vec{r}, \vec{\Omega}_j \cdot \vec{\Omega}_i)\psi(\vec{r}, \vec{\Omega}_j).$$

Using equation (1.13), we obtain:

$$R\psi(\vec{r}, \vec{\Omega}_i) = \frac{1}{4\pi}\Sigma_{s0}(\vec{r})\phi_{00}(\vec{r}),$$

where $\phi_{00}(\vec{r})$ is the zeroth angular flux moment defined as:

$$\phi_{00}(\vec{r}) = \int_{S_2} d\vec{\Omega}'\psi(\vec{r}, \vec{\Omega}') \simeq \sum_{j=1}^{N_{\text{dir}}} w_j \psi(\vec{r}, \vec{\Omega}_j).$$

One should note that the major drawback of the discrete ordinates method is the problem of so called "ray effects". Indeed, as shown in [72], for some problems, such as those featuring a localized fixed-source in a pure absorber media, the discrete ordinates method can give a flux only for regions "surrounded" by the directions of the ordinates, taking their origin in the fixed-source. To remedy this issue, one solution is to increase the order of the angular quadrature used. However, in a nuclear reactor core, which is our focus in this work, the neutron sources are uniformly distributed in the whole core, and thus the probability of having non-covered regions is smaller. Therefore, our goals are not oriented towards numerical methods for eliminating these effects.

The resolution of (1.15) is obtained by iterating on the in-scattering term R as presented in Algorithm 3. In highly diffusive media, the convergence of this algorithm is very slow, and therefore an acceleration scheme must be combined with this algorithm in order to speed-up its convergence. Section 5.1 presents a new acceleration scheme used in Domino to improve the convergence of scattering iterations. As presented in Algorithm 3, each scattering iteration involves the resolution of a fixed-source problem (Line 3), for every angular direction. This is done by discretizing the streaming operator which is presented in following section.

Algorithm 3: Scattering iterations

```
Input  : $\psi^k$
Output: $\psi^{k+\frac{1}{2}}$
1 while Non convergence do
2     $R\psi^k(\vec{r},\vec{\Omega}) = \int_{S_2} d\vec{\Omega}'\Sigma_s(\vec{r},\vec{\Omega}'\cdot\vec{\Omega}_i)\psi^k(\vec{r},\vec{\Omega}')$;
3     $L\psi^{k+\frac{1}{2}}(\vec{r},\vec{\Omega}) = R\psi^k(\vec{r},\vec{\Omega}) + Q(\vec{r},\vec{\Omega})$;
4     ▷ Compute stopping criterion
```

1.3.2 Spatial discretization

Let us consider the fixed-source monokinetic equation along the angular direction $\vec{\Omega} = (\Omega_x, \Omega_y, \Omega_z)$ (Line 3 of the Algorithm 3):

$$\Omega_x \frac{\partial \psi}{\partial x}(x,y,z) + \Omega_y \frac{\partial \psi}{\partial y}(x,y,z) + \Omega_z \frac{\partial \psi}{\partial z}(x,y,z) + \Sigma_t \psi(x,y,z) = B(x,y,z), \quad (x,y,z) \in V_{ijk} \tag{1.17}$$

where B gathers all neutron sources including scattering and fission; the angular variable is omitted to lighten the notations. In this work, we focus on a 3D reactor core model, represented by a 3D Cartesian domain $\mathcal{D}$. The spatial variable of this equation is discretized using a diamond difference scheme (DD), as presented by A. Hébert in [50]. In particular, the DD0 scheme, as implemented in Domino, is derived by combining the moment of order 0 of the transport equation with some closure relations.

Let us first define a map from the cell $V_{ijk} = [x_i, x_{i+1}] \times [y_j, y_{j+1}] \times [z_k, z_{k+1}]$, of size $\Delta x_i \times$

$\Delta y_j \times \Delta z_k$, to the reference mesh $V_{ref} = [-1,1] \times [-1,1] \times [-1,1]$ as follows:

$$\mathcal{M}\colon V_{ijk} \to V_{ref}$$
$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \mapsto \begin{pmatrix} \hat{x} = \frac{2x-(x_i+x_{i+1})}{\Delta x_i} \\ \hat{y} = \frac{2y-(y_j+y_{j+1})}{\Delta y_j} \\ \hat{z} = \frac{2z-(z_k+z_{k+1})}{\Delta z_k} \end{pmatrix}.$$

Using this map, the equation (1.17) is rewritten in:

$$\frac{2\Omega_x}{\Delta x_i}\frac{\partial \psi}{\partial x}(x,y,z) + \frac{2\Omega_y}{\Delta y_j}\frac{\partial \psi}{\partial y}(x,y,z) + \frac{2\Omega_z}{\Delta z_k}\frac{\partial \psi}{\partial z}(x,y,z) + \Sigma_t \psi(x,y,z) = B(x,y,z) \quad (x,y,z) \in V_{ref}. \tag{1.18}$$

Moment of order 0 of the transport equation. The moment of order 0 of the transport equation is obtained by integrating the equation (1.18) on the cell V_{ref}:

$$\begin{aligned} &\Sigma_t \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \psi(x,y,z)dxdydz \\ &+ \frac{2\Omega_x}{\Delta x_i}\int_{-1}^{1}\int_{-1}^{1} \left(\psi(1,y,z) - \psi(-1,y,z)\right) dydz \\ &+ \frac{2\Omega_y}{\Delta y_j}\int_{-1}^{1}\int_{-1}^{1} \left(\psi(x,1,z) - \psi(x,-1,z)\right) dxdz \\ &+ \frac{2\Omega_z}{\Delta z_k}\int_{-1}^{1}\int_{-1}^{1} \left(\psi(x,y,1) - \psi(x,y,-1)\right) dxdy = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} B(x,y,z)dxdydz. \end{aligned} \tag{1.19}$$

To lighten the notations, we define 2 volumetric moments,

$$\psi^{000} = \frac{1}{8}\int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \psi(x,y,z)dxdydz,$$
$$B^{000} = \frac{1}{8}\int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} B(x,y,z)dxdydz,$$

and 6 surface flux moments (3 incoming and 3 outgoing),

$$\psi^{X\pm,00} = \frac{1}{4}\int_{-1}^{1}\int_{-1}^{1} \psi(\pm 1,y,z)dydz,$$
$$\psi^{Y\pm,00} = \frac{1}{4}\int_{-1}^{1}\int_{-1}^{1} \psi(x,\pm 1,z)dxdz,$$
$$\psi^{Z\pm,00} = \frac{1}{4}\int_{-1}^{1}\int_{-1}^{1} \psi(x,y,\pm 1)dxdy.$$

Then equation (1.19) becomes:

$$\frac{\Omega_x}{\Delta x_i}\left(\psi^{X+,00} - \psi^{X-,00}\right) + \frac{\Omega_y}{\Delta y_j}\left(\psi^{Y+,00} - \psi^{Y-,00}\right) + \frac{\Omega_z}{\Delta z_k}\left(\psi^{Z+,00} - \psi^{Z-,00}\right) + \Sigma_t \psi^{000} = B^{000}, \tag{1.20}$$

where $\psi^{X-,00}$, $\psi^{Y-,00}$, $\psi^{Z-,00}$ are known thanks to the boundary conditions. However, 4 unknowns have to be determined according to this equation: the volumic angular flux ψ^{000} and the 3 outgoing surface moments $\psi^{X+,00}$, $\psi^{Y+,00}$, $\psi^{Z+,00}$. Indeed, since it is not possible to determine 4 unknowns from a single equation, we must combine equation (1.20) with 3 other equations. These equations are the closure relations provided by the diamond differencing scheme.

Closing relations These relations are obtained by cancelling the first term appearing in the moment of order 1 of the transport equation. The closing relations for the DD0 scheme are given as follows:

$$\begin{cases} \psi^{X+,00} = 2\psi^{000} - \psi^{X-,00} \\ \psi^{Y+,00} = 2\psi^{000} - \psi^{Y-,00} \\ \psi^{Z+,00} = 2\psi^{000} - \psi^{Z-,00} \end{cases} \tag{1.21}$$

The equations (1.20) and (1.21) are solved by "walking" step by step throughout the whole spatial domain and to progressively compute angular fluxes in the spatial cells. In the literature, this process is known as the **sweep operation**. The whole S_N algorithm as implemented in DOMINO is presented in Algorithm 4. In this algorithm, the sweep operation (Line 15) gathers the vast majority of computations performed. This operation is the focus of this dissertation, and we are going to describe it in detail in the following section.

1.3.3 Cartesian transport sweep operation

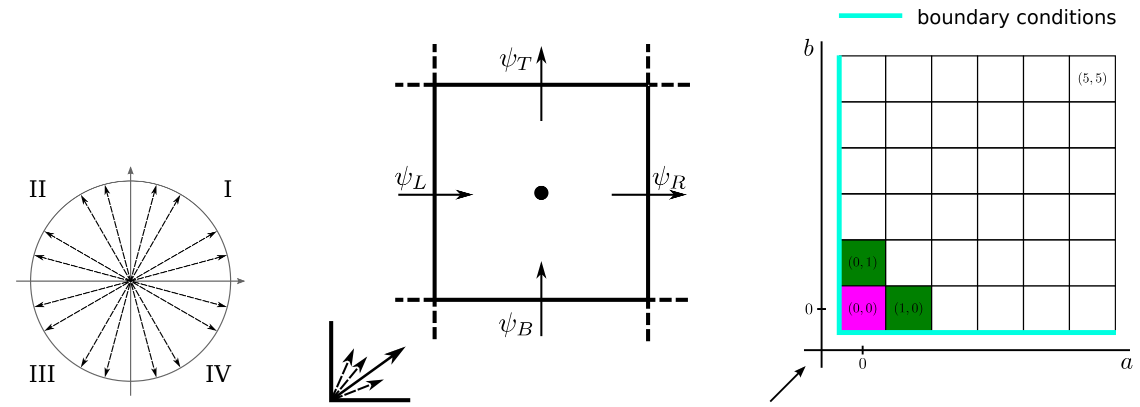
Without loss of generality, we consider the example of a 2D spatial domain discretized into 6 × 6 cells. The DD0 scheme in this case defines 3 DoFs per spatial cell: 1 for the neutron flux and 2 for the neutron current. The sweep operation is used to solve the space-angle problem defined by equations (1.20) and (1.21). It computes the angular neutron flux inside all cells of the spatial domain, for a set of angular directions. These directions are grouped into four quadrants in 2D (or eight octants in 3D). In the following, we focus on the first quadrant (labeled I in Figure 1.2a). As shown in Figure 1.2b, each cell has two incoming dependencies ψ_L and ψ_B for each angular direction. At the beginning, incoming fluxes on all left and bottom faces are known as indicated in Figure 1.2c. Hence, the cell $(0, 0)$ located at the bottom-left corner is the first to be processed. The treatment of this cell allows the updating of outgoing fluxes ψ_R and ψ_T, that satisfy dependencies of the cells $(0, 1)$ and $(1, 0)$. These dependencies on the processing of cells define a sequential nature throughout the progression of the sweep operation: two adjacent cells belonging to successive diagonals cannot be processed simultaneously. Otherwise, treatment of a single cell for all directions of the same quadrant can be done in parallel. Furthermore, all cells belonging to a same diagonal can be processed in parallel. Hence, step by step, fluxes are evaluated in all cells of the spatial domain, for all angular directions belonging to the same quadrant. The same operation is repeated for all the four quadrants. When using vacuum boundary conditions, there is no incoming neutron to the computational domain and therefore processing of the four quadrants can be done concurrently. This sweep operation is subject to numerous studies regarding design and parallelism to reach highest efficiency on parallel architectures. In Chapter 4, we will present our task-based approach that enables us to leverage the full computing power of such architectures.

1.4 Modern parallel computers and performance evaluation

The landscape of today's high-performance computing capability includes a large number of different and powerful computing devices. These are essentially multicore processors and Graphics Processing Units (GPUs). While the former can be considered as an evolution of the classical processors, the latter are of a new kind because of the programming model shift required. Although the focus of this dissertation is on multicore-based architectures, one should keep in mind that this work provides an opening towards GPU-based architectures.

Algorithm 4: Discretized algorithm of S_N method as implemented in DOMINO

```
Input  : $\nu\Sigma_f$, $\Sigma_s$, $\Sigma_t$
Output: $k_{\text{eff}}$, $\psi$
▷ Initialization of external iterations
$\phi = \left(1, 1, \cdots, 1\right)^t$;
$C = \sum_{g=1}^{G} \nu\Sigma_{f,g}.\phi_g$ ; ▷ Fission source computation
▷ External iterations: inverse power algorithm
while Non convergence do
    $S = \begin{pmatrix} \chi_1 \\ \chi_2 \\ \vdots \\ \chi_G \end{pmatrix} \cdot C$
    ▷ Multigroup iterations: Gauss-Seidel
    while Non convergence do
        for $g \in [\![1, G]\!]$ do
            ▷ External sources
            $Q_{ext} = S_g + \sum_{g' \neq g} \Sigma_s^{g' \to g}.\phi_{g'}$ ;
            ▷ Scattering iterations
            while Non convergence do
                $Q = Q_{ext} + \Sigma_s^{g \to g}\phi_g$ ;
                $\vec{\Omega}_k.\vec{\nabla}\psi_k + \Sigma\psi_k = Q \quad \forall\vec{\Omega}_k \in S_N$;
                $\phi_g = \sum_{\Omega_k} \omega_k\psi_k$;
    $\mathbf{S}_{\text{old}} = \mathbf{S}$;
    $C = \sum_{g} \nu\Sigma_{f,g}.\phi_g$ ; ▷ Fission sources update
    $k = \sqrt{\frac{<\mathbf{S},\mathbf{S}>}{<\mathbf{S}_{\text{old}},\mathbf{S}_{\text{old}}>}}$;
    $\mathbf{S} = \frac{1}{k}\mathbf{S}$;
```



(a) Angular quadrature in 2D. Directions are grouped in quadrants.

(b) In each direction, cells have two incoming components of the flux (Here from the left and bottom faces: ψ_L and ψ_B), and generates two outgoing components of the flux (Here on the right and top faces: ψ_R and ψ_T).

(c) Domain decomposition and boundary conditions. The corner cell $(0, 0)$ is the first to be processed for a quadrant, and its processing releases computations of its neighbors (Here $(0, 1)$ and $(1, 0)$).

Figure 1.2: Illustration of the sweep operation over a 6×6 2D spatial grid for a single direction.

1.4.1 Architecture and design of parallel computers

The advent of the multicore processor In the area of computer architecture, Moore's Law [85][1] had allowed for the production of powerful monolithic processors until the beginning of the 2000s. The increase of the computing power of the processors was thus mainly due to the increase of the clock rate. However, as the transistors get smaller, their power density[2] does not scale down below a threshold as it was previously predicted by the Dennard scaling [29, 31]. The consequence of this "power wall" (limits on temperature of the chip and its power consumption) has led processor designers to shift toward multicore processors to keep the growth of the processor performance. Furthermore, to increase the power efficiency (Performance/Watt), processor cores integrate new functional units that are capable of performing a Single Instruction on Multiple Data (SIMD) that will be discussed in Chapter 2. These units, also called vector units, are now common on all modern processors and will continue to be present on processors henceforth.

Modern distributed memory clusters Building ever more powerful computers can be done by aggregating either several multicore processors (sockets) in a single computing node, sharing the main memory of the computer, or several computing nodes interconnected by a high-speed network. In the first case, we obtain either a Symmetric Multi-Processor (SMP) if all the CPU cores on the processor chip have the same access latency to the main memory, or a Non Uniform Memory Access (NUMA) computing node, providing hierarchical access to the memory. Currently, modern distributed computers are built from an interconnection of NUMA nodes, and therefore the parallelism on these computers is decoupled into four parts:

- SIMD units in each CPU core;

---

[1] The number of transistors per chip doubles approximately every two years

[2] Power density is the amount of power (time rate of energy transfer) per unit volume. `https://en.wikipedia.org/wiki/Power_density`

- CPU cores in each socket;
- sockets in each node;
- nodes of a supercomputer.

Consequently, as already noted “the free lunch is over”[1], and taking advantage of the increasing computing power brought by multicore-based architectures, requires a deep shift on the traditional programming models used until now. Indeed, these architectures are generally addressed by means of either *multithreading* or message-passing techniques. While the first solution is limited to shared-memory systems, the second one can be used both on shared and distributed memory systems. A combination of both approaches can also be used. In Chapter 4, we will discuss these solutions in detail.

1.4.2 Metrics for performance evaluation

We primarily use parallel computers to speed-up the computation time required for executing a given workload. To evaluate the performance of the workload on a given parallel computer, the main metric is the elapsed time which allow to determine the parallel efficiency of the workload.

Parallel efficiency This metric is related to the speed-up that can be achieved when running an application on a parallel computer. We consider the following notations:

- T_s the serial computation time;
- T_p the parallel computation;
- P the number of parallel processing units in use.

Then, the speed-up (S) and parallel efficiency (E) are defined by:

$$S = \frac{T_s}{T_p}, \quad E = \frac{S}{P}. \tag{1.22}$$

However, even though the parallel efficiency metric determines the scalability of an application, it does not indicate if the considered application is efficiently using the full computing power of the computer. To achieve this, we rather use another metric.

Flop/s This metric determines the number of floating point operations (Flop) executed per second (s) on a given machine, which can be considered as the “speed of execution” of a computational kernel on a given machine. Each computational kernel is associated with a number of Flop which evaluation can be done either by counting all the floating point operations that exist in the kernel, or using performance monitoring tools that collect hardware performance counter events during the execution of the computational kernel. In general, the Flop/s metric characterizes the performance of an application on a given machine and it should be compared to the theoretical peak performance of the target computer, which is the maximum number of Flop that a computer can perform per second.

▶ Having defined these metrics, we will use them throughout this dissertation to quantitatively evaluate the performances of our implementation on the target architectures. As mentioned, the point we made in this thesis is to tailor the discrete ordinates method for emerging architectures. In order to meet this goal, the first step in our process is the vectorization of the sweep operation as presented in Chapter 2.

---

[1] `http://www.gotw.ca/publications/concurrency-ddj.htm`

Chapter 2

On the Vectorization of the Sweep Operation

Contents

2.1 Review of the SIMD model 22
2.1.1 General presentation of the SIMD paradigm 22
2.1.2 On the way for vectorization 25
2.1.3 Generic programming and tools for the vectorization 25
2.2 Arithmetic intensity of the sweep kernel 26
2.2.1 Memory traffic and flops per cell and per direction 26
2.2.2 General formula of the arithmetic intensity of the sweep 29
2.3 Vectorization over spatial domain 30
2.3.1 The algorithm 31
2.3.2 Maximum theoretical speed-up 32
2.4 Vectorization over the angular variable 32
2.4.1 The algorithm 32
2.4.2 Study of the arithmetic intensity of the sweep 33
2.4.3 Actual performances vs Roofline model 35

As we mentioned in Chapter 1, on the way to designing a highly efficient massively parallel neutron transport solver targeting modern multicore-based supercomputers, we should take care of all the microarchitectural improvements brought by these computers. Since the early 2000s and the advent of the multicore processor, the main processor manufacturers have designed and introduced new vector execution units into the CPU cores, capable to apply a Single Instruction on Multiple Data (SIMD) residing in dedicated registers. The consequence of adding vector units to the CPU cores is a large boost of the corresponding processors peak performance. This SIMD execution model, also called *vectorization*, can therefore be used to reduce the run time of scientific workloads, and thus to maximize the sustainable peak performance [74] and the energetic efficiency of these workloads [38]. In light of this observation, there is more and more research on designing new algorithms and re-factoring legacy scientific codes to enhance their compatibility to vectorization. We can cite the works in [67] where the authors describe some optimizations to help efficiently use vector units for the classic operation of sparse matrix-vector multiply consisting in designing a vector-friendly storage format. Moreover, in [19], authors presented a design of a vectorized implementation of the MergeSort algorithm. In [51] the authors presented an efficient vectorized implementation of a stencil computation. For this reason, we propose in this chapter to study the vectorization capabilities offered by the most computational demanding operation in the discrete ordinates method: the sweep operation, in order to maximize its sustainable single-core performance.

The remainder of this chapter is organized as follows: in section 2.1, we recall the vector (or SIMD) programming model and how it can be used on modern multicore processors. Then, we give in section 2.2 a theoretical analysis of the sweep operation by the means of its computational intensity. In section 2.3 and section 2.4, we present two different strategies that can be used to vectorize the sweep operation.

2.1 Review of the SIMD model

This section recalls the SIMD programming model from early vector processors to modern multicore processors.

2.1.1 General presentation of the SIMD paradigm

Historical trends

In Flynn's taxonomy of computer architectures [40], Single Instruction Multiple Data (SIMD) corresponds to a class of computers that can perform a single instruction on a set of data stream. Such computers are also called vector processors and the process of applying a SIMD instruction is commonly called vectorization. According to the study in [32], vector processors, can be classified into two categories: *memory-to-memory* and *memory-to-register*. In the first category, corresponding to early vector processors, the vector functional units directly retrieves data from the memory, process it and write back the result to memory. *memory-to-register* vector processors have vector registers that are used to store data retrieved from the main memory, and on which vector instructions operate. In practice, early vector processors used to exhibit a high ratio of bandwidth over peak performance and the speed of vectorized algorithms was optimal for basic BLAS1 like operations on long vectors. On modern chips, the situation is totally different and we will see that the performance of most vectorized algorithms is bounded to the *memory-to-register* bandwidth.

SIMD on modern CPUs

Increasing the processor clock rate, or frequency, has been, until the end of the last century, the major factor for improving the floating point performance of processors. However, since the early 2000s this frequency scaling stalled in favor of multicore chips allowing to have powerful processors while minimizing their power consumption. Furthermore, to increase the computing power of a single core, processor designers tend to add larger vector units into the processor cores. These units are capable to apply SIMD instructions, which are extensions of the processor Instruction Set Architecture (ISA), on fixed-size vector registers. Thereby, these vector units can be used to speed-up computations by a factor theoretically equal to the SIMD width of the considered processor. Table 2.1 shows the evolution of the SIMD width for Intel processors. The number of elements that fit into a SIMD register of a given architecture can also be understood as the maximum theoretical speed-up attainable on that architecture. Vector instructions comprises in addition of the classic floating point arithmetics (add/sub/mul/div), other types of instructions such as gather and scatter, or logical operations. Each of these instructions is characterized by a latency and a throughput[1], determined by the considered architecture. On Westmere, the load and store instructions, from CPU L1 cache to SIMD registers, have a throughput of 16 Bytes/cycle. However, as noted in [48], on Sandy Bridge, which has doubled the SIMD width from 128 bits to 256 bits, the load throughput has doubled to 32 Bytes/cycle compared to Westmere, whereas the store throughput remains the same at 16 Bytes/cycle. Furthermore, in [114], authors show that the Intel Sandy Bridge microarchitecture gives no performance gain for the division as compared to the Intel Westmere microarchitecture, on a class of benchmarks. Thereby, for a code bottlenecked by division, the theoretical 2-fold speed-up when moving from Westmere to Sandy Bridge, to which one may expect, can not be observed.

Architecture	Launch date	SIMD width (bits)	Elements per register		ISA
			single precision	double precision	
Westmere	01/2010	128	4	2	SSE
Ivy Bridge	04/2012	256	8	4	AVX
Skylake	08/2015	512	16	8	AVX-512

Table 2.1: SIMD width as a function of the processor architecture.

On the Skylake microarchitecture[2], it is possible to perform up to 16 floating point instructions using a single vector instruction. Consequently, without the usage of SIMD units, we can only take advantage of 1/16 of the CPU peak performance. For this reason, it is essential that scientific codes manage to make usage of these units in order to maximize their sustainable floating point performance. However, a perfectly vectorized algorithm performance will generally depend on its *arithmetic intensity*.

The critical arithmetic intensity issue

As mentioned previously, successive supercomputer generations brought a regular and impressive improvement of their peak performance. Surprisingly enough, the computer bandwidth that measures the maximal data flow between the computer RAM (off-chip bandwidth) and the floating point units (FPUs) did not increase as fast as the peak performance. The consequence of the broadening gap between the off-chip bandwidth and the chip peak performance is to put a dramatic emphasis on the *arithmetic intensity* of computational kernels and defined by:

---

[1] A full list of latency and throughput of instructions is available at `http://www.agner.org/optimize/instruction_tables.pdf`

[2] According to an Intel communication at `https://gcc.gnu.org/wiki/cauldron2014?action=AttachFile&do=view&target=Cauldron14_AVX-512_Vector_ISA_Kirill_Yukhin_20140711.pdf`, processors that will implement the AVX-512 ISA are: Intel Knights Landing co-processor and processors of Xeon series.

$$I_a = \frac{\text{Number of floating point operations}}{\text{Main memory traffic (Read+Write)}} \, .$$

It measures the average number of instructions executed by the CPU per byte read from or written to the main memory, and should be compared to the processor-dependent constant *critical arithmetic intensity*, defined by:

$$I_c = \frac{\text{Peak floating point operations}}{\text{Off-chip Bandwidth}} \, .$$

If the arithmetic intensity I_a of a given kernel is lower than the critical value I_c, then its performance does not depend on the computational power of the target processor, but mainly on the system memory bandwidth: the algorithm is then said to be *memory bound.* Consequently, the impact of the vectorization on such a kernel, with a low arithmetic intensity is negligible. Contrariwise, if I_a is higher than I_c, then the computational kernel is said to be *cpu bound*, and it therefore can benefit from the speed-up of computations enabled by the vectorization.

The critical arithmetic intensity of processors increases with each generation causing an ever larger fraction of algorithms to be *memory bound.* Indeed, on Table 2.2 we give the values of I_c for our test machines: the BIGMEM computing node (Intel Xeon E78837) and a node of the ATHOS platform (Intel Xeon E52697 V2). We distinguish the cases of a single-core, single-socket and the whole node, as the peak sustainable DRAM bandwidth varies with the number of cores in use. We used the Stream benchmark [79, 80] to evaluate the bandwidth in three cases[1]. We found that the single-core I_c of the considered processors are 3.6 Flop/Byte for BIGMEM and 3.4 Flop/Byte for a node of the ATHOS platform. For a single-socket these values are of 8.3 Flop/Byte on the BIGMEM node compared to 15.0 Flop/Byte, on the more recent ATHOS platform highlighting the increase of the critical arithmetic intensity for successive processor generations. As a consequence of this processor evolution, the whole design of our sweep implementation aimed at maximizing the *arithmetic intensity* in order to exploit the full power of modern multicore processors, thanks to the vectorization.

	BIGMEM			ATHOS		
	Stream Bandwidth (GB/s)	Theoretical Peak (GFlop/s)	I_c (F/B)	Stream Bandwidth (GB/s)	Theoretical Peak (GFlop/s)	I_c (F/B)
single-core	5.8	21.2	3.6	12.6	43.2	3.4
single-socket	20.3	170.2	8.3	34.4	518.4	15.0
full node	76.3	680.9	8.9	67.5	1036.8	15.3

Table 2.2: Critical arithmetic intensities of Intel Xeon E78837 and Intel Xeon E52697 V2 processors.

---

[1] It should be noted that Stream does not account for caches. It targets DRAM bandwidth

2.1.2 On the way for vectorization

Exploitation of vector units can be done by relying on the compiler auto-vectorizer. However, while this may work for simple loops, it is generally not the case for complex kernels. In [77], the authors detail give several reasons on why the compiler fails in generating vector code. For instance, compilers generally lack accurate interprocedural analysis, which would help to enable important transformations such as changing the memory layout of data structures needed to vectorize the code. For example, such a memory layout change is required when data are accessed with non-unit stride accesses. Meanwhile, by explicitly using assembly instructions or compiler intrinsics corresponding to the target architecture, one can exploit these vector units with the expense of some hardware constraints specific to vector instructions. In fact, vector instructions operate on packed data loaded inside specialized registers of fixed size. To perform fast load and store operations, data items need to be well aligned on cache boundary: 16 bytes for SSE, 32 bytes for AVX and 64 bytes for AVX-512. Sometimes we have to resort to padding to satisfy this requirement. As an example, when loading 256 bits packet data with Intel AVX in a contiguous memory region of size $256 + 32 \times 3 = 352$ bits, we need to extend this region with $512 - 352 = 160$ bits at the memory allocation stage. Attention should be paid to padding as it increases global memory consumption and useless computations. The drawback of inlining compiler intrinsics into the code is that it degrades the readability of the code and is error prone. To overcome this, one can rely on C++ generic programming concepts by overloading the arithmetic operators to call the corresponding compiler intrinsics.

2.1.3 Generic programming and tools for the vectorization

One solution to get a vectorized code is to use external libraries offering optimized instructions for a given architectures. Examples of these libraries include Intel MKL [115] or ACML [3]. However, these libraries are especially designed and tuned for linear algebra routines and thus can not be easily used in other contexts. It is also possible to rely on the compiler auto-vectorizer, which can enforce the vectorization of loops via a set of hints dictated by the programmer. Thus, the `pragma simd` extension, firstly introduced by Intel Cilk Plus [103] and integrated in GCC and ICC compilers, tells the compiler that a given loop can be vectorized. The performance that can be obtained from such an approach depends on the compiler in use [33]. There are also some SIMD-enabled languages which feature some extensions for writing explicit vectorized code. This is the case with the Intel Cilk array notations [103] where `A[:]` states that an operation on the array `A` is a vector instruction. The Intel SPMD Compiler (ISPC) [91], an LLVM-based language and compiler similar to CUDA/OpenCL, allows to automatically vectorize a C-based code, by mapping several SPMD program instances to SIMD units.

On the other hand, one can use an external library that calls to the compiler intrinsics of the target architecture, and providing a classical API to the application developer that enhances the portability of the code. For instance, Boost.SIMD [33] is a C++ template library which provides a class holding packed data residing in a SIMD register, along with a set operations overloaded to call the corresponding compiler intrinsics. Eigen [47] is a similar library, but unlike Boost.SIMD, it provides a `Map` object that can be used to interface with raw buffers. Thereby, it is possible to easily and elegantly take advantage of the vectorization of operations offered by Eigen, in a code using other containers than those of Eigen. This is the case for our DOMINO code, which uses the `Legolas++` [65], a generic C++ library internally developed at EDF. This library provides basic constructing blocs to build Linear Algebra Solvers. In particular, it allows to write algorithms that apply indifferently on arrays of scalar and on arrays of packs of scalars.

Legolas++ is built on top of Eigen for SIMD issues and INTEL TBB [92] for multi-threading.

Hence, in the following, we rely on the Eigen library to vectorize the sweep kernel, allowing us to benefit from new vector instructions of the next generation of processors without having to modify the code. As an example, the same source code will automatically be compiled for SSE or AVX machine instructions depending on the C++ compiler options. Note that Eigen, internally invoke the SIMD instructions explicitly and that the vectorized binary performance do not depends on the auto-vectorization capability of the C++ compiler.

2.2 Arithmetic intensity of the sweep kernel

In this section, we evaluate the arithmetic intensity of the sweep operation, as a function of the spatial and angular discretizations, in order to characterize the performance of the sweep operation. We first consider the case of a single cell with a single direction.

2.2.1 Memory traffic and flops per cell and per direction

We recall that in the sweep operation, the processing of a single spatial cell c_{ijk} for a single angular direction $\vec{\Omega}$, consists in updating: the scalar flux at the cell-center and the neutron current on outgoing faces. This processing is depicted on Figure 2.1 (in DD0) and described on Algorithm 5, where ψ_u^0 and ψ_u^1 represent incoming and outgoing neutron current along the u dimension ($u = x, y, z$).

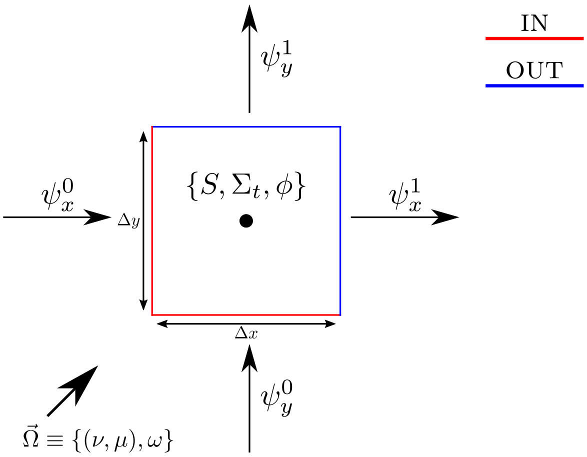


Figure 2.1: Processing of a single spatial cell (in DD0) in 2D, with a single angular direction.

To evaluate the arithmetic intensity of Algorithm 5, we need to determine the flops and the number of memory accesses per cell, per angular direction.

Memory accesses According to the Algorithm 5, on each spatial cell, for each angular direction, the update of the outgoing neutron current requires to load:

- 12 input scalar values according to the quadrature formula used: inverse of the mesh steps δ_u, $u = x, y, z$; source term S; total cross-section Σ_t; outgoing neutron current ψ_u^0, $u = x, y, z$; coordinates and weight of the angular direction $(\nu, \eta, \xi), \omega$;
- 1 input/output value: scalar flux ϕ;

Algorithm 5: Processing of a single spatial cell (in DD0), per angular direction.

```
In  : $\delta x = \frac{2}{\Delta x}$; $\delta y = \frac{2}{\Delta y}$; $\delta z = \frac{2}{\Delta z}$; $S$; $\Sigma_t$; $\phi$; $\{\psi_u^0 | u = x, y, z\}$; $\overrightarrow{\Omega} \equiv \{(\nu, \mu, \xi), \omega\}$
Out: $\phi$; $\{\psi_u^1 | u = x, y, z\}$;
$\epsilon_x = \nu\ \delta x$;    $\epsilon_y = \eta\ \delta y$;    $\epsilon_z = \xi\ \delta z$;
$\psi = \frac{\epsilon_x \psi_x^0 + \epsilon_y \psi_y^0 + \epsilon_z \psi_z^0 + S}{\epsilon_x + \epsilon_y + \epsilon_z + \Sigma_t}$;
$\psi_x^1 = 2\psi - \psi_x^0$;
$\psi_y^1 = 2\psi - \psi_y^0$;
$\psi_z^1 = 2\psi - \psi_z^0$;
$\phi = \phi + \psi \cdot \omega$;
```

- and 3 output values: outgoing neutron current $\psi_u^1,\ u = x, y, z$.

This amounts to a total of 16 read/write memory accesses. As a matter of fact, the storage of the neutron current is not necessary for a stationary computation. This reduces the drop of the number of memory accesses to 13. In this case, the neutron currents will simply be denoted by $\psi_u,\ u = x, y, z$.

Flops The floating point operations performed on a cell consist of 7 additions, 3 subtractions, 10 multiplications and 1 division. Note that the quantities $\frac{2}{\Delta_u}$, $u = x, y, z$ are pre-evaluated, allowing to save 3 divisions and 3 multiplications per cell. On modern processors, each of add/-sub/mul operation account for 1 flop. The question of how many flops to count for one floating point division is a debatable one as the answer depends on how this operation is implemented on the target architecture. There exist however some tools that can help in evaluating the exact number of flops of a given kernel for a given architecture. Some of these tools, as LIKWID[1] or PAPI[2], monitor events occurring in hardware performance counters of the target processor, in order to determine actual flops executed by the processor in a laps of time. Some other tools evaluate the flops with extended benchmarks. For instance, LIGHTSPEED [3] is a library featuring a set of routines for accurate flops counting of operations other than add/sub/mul. With this library, the number of flops of such an operation is equal to average value of the ratio between the execution time of the considered operation and the execution time of a multiplication (which counts as 1 flop), on the same architecture. According to our experiments with Lightbox, we found that the division operation costs 4.8 flops (resp. 5.8 flops) on BIGMEM (resp. ATHOS). However, for our studies, we want an architecture-independent measure of the flops of our sweep kernel, in order to compare its performance on the different architectures. Consequently, we have conventionally set the number of flops of a division to be 5. Therefore, the processing of a single spatial cell for a single angular direction cost 25 flops.

Thus, in single precision, the arithmetic intensity of the sweep of a single cell with one angular direction is $\frac{25}{4\times16} \approx 0.39$ if the neutron currents are stored, and $\frac{25}{4\times13} \approx 0.48$ otherwise. Therefore, it is found that these values are below I_c of our target machines (see Table 2.2), indicating that it is not efficient to sweep a single cell with a single direction. We are going to increase it by grouping the cells into a `MacroCell` object.

---

[1] https://code.google.com/p/likwid/

[2] http://icl.cs.utk.edu/papi/

[3] http://research.microsoft.com/en-us/um/people/minka/software/lightspeed/

Definition 1 (`MacroCell`). *A `MacroCell` is a structure that represents a 3D block of contiguous spatial cells, and some physical data associated to these cells as described on Listing 2.1. The number of cells aggregated into the x, y and z dimensions is respectively denoted by n_x, n_y, and n_z.*

Listing 2.1: The MacroCell structure.

```
template <class RealType>
class MacroCell {

  private:

  // total section
  Legolas::MultiVector<RealType,3> sigma_;

  // mesh steps
  Legolas::MultiVector<RealType,2> steps_;

  // inverses of the mesh steps
  Legolas::MultiVector<RealType,2> invSteps_;

  // total source (fission + scattering)
  Legolas::MultiVector<RealType,3> source_;

  // scalar flux
  Legolas::MultiVector<RealType,4> phi_;

  // angular quadrature
  VectorizedQuadrature<RealType> quadrature_;

  // number of angular directions per octant
  int directionPerOctantNumber_;

  // number of cells in the MacroCell
  int nx_;
  int ny_;
  int nz_;

  public:

  void computePhi ( int forwardX, int forwardY, int forwardZ,
                    Legolas::MultiVector<RealType,3> & psiX,
                    Legolas::MultiVector<RealType,3> & psiY,
                    Legolas::MultiVector<RealType,3> & psiZ );

};
```

2.2.2 General formula of the arithmetic intensity of the sweep

We consider the sweep of a `MacroCell` of size $n_x \times n_y \times n_z$, using a quadrature formula having D angular directions. In the previous section, we have shown that there are 25 Flop/cell/direction. Therefore, in a general case, the number of flops is $25Dn_xn_yn_z$, and the associated number of memory accesses is given on Table 2.3. The sizes of variables are detailed as follows.

Storage of the currents	Variable	Size ($\times 4$ Bytes)
	$\{\delta_u \mid u = x, y, z\}$	$n_x + n_y + n_z$
	S	$n_xn_yn_z$
	Σ_t	$n_xn_yn_z$
	ϕ	$n_xn_yn_z$
	(ν, μ, ξ)	$3D$
	ω	D
No	$\{\psi_u \mid u = x, y, z\}$	$D(n_xn_y + n_xn_z + n_yn_z)$
Yes	$\{\psi_u^l \mid u = x, y, z \text{ and } l = 0, 1\}$	$D(3n_xn_yn_z + n_xn_y + n_xn_z + n_yn_z)$

Table 2.3: Memory accesses required to perform a sweep over a `MacroCell` of size $n_x \times n_y \times n_z$ using a quadrature formula comprising D angular directions.

- δ_u: There are n_u cells along the dimension $u = x, y, z$; each cell being associated to a mesh step. Thus, the total number of mesh steps are the sum of the number of cells for the three dimensions (x, y, z): $n_x + n_y + n_z$.
- S, Σ_t, ϕ: For each spatial cell, there is one of each; thus, the size of each of these variables is equal to the total number of cells in the `MacroCell`: $n_xn_yn_z$.
- For each angular direction, there are:
  - $3 + 1$ values which represent the coordinates and the weight associated with the direction.
  - 1 value per incoming face of the `MacroCell`, for the neutron currents, if they are not stored. The size of each `MacroCell` face is either n_xn_y, or n_yn_z or n_xn_z. If the neutron currents are stored, we must use 3 additional values per cell.

Hence, the general formula evaluating the arithmetic intensity for a sweep of a `MacroCell`, if all the neutron currents are stored, I_a^s is given by equation (2.1).

$$I_a^s = \frac{25Dn_xn_yn_z}{4((n_x + n_y + n_z) + 3n_xn_yn_z + D(3n_xn_yn_z + n_xn_y + n_xn_z + n_yn_z) + 4D)} \tag{2.1}$$

However, as mentioned in section 2.1.1, flops are cheaper than memory accesses on modern processors because of the gap between the processor peak performance and the off-chip memory bandwidth. Thus, to take advantage of all the power of modern processors, it is essential to save the memory bandwidth by reducing the unnecessary number of memory accesses. We consider this requirement and we do not store the neutron currents. In this case, the arithmetic intensity is given by equation (2.2).

$$I_a = \frac{25Dn_xn_yn_z}{4((n_x + n_y + n_z) + 3n_xn_yn_z + D(n_xn_y + n_xn_z + n_yn_z) + 4D)} \tag{2.2}$$

A numerical evaluation of these formulas is presented in Figure 2.4).

Algorithm 5 shows that the computations performed on a single cell is similar for any two angular directions and for any two cells. The full sweep algorithm in 3D is presented on Algorithm 6, where the angular directions are grouped in octants (Line 1). Each octant has M angular directions such that $D = 8M$. While the processing of a cell for all angular directions belonging to a single octant can be done once, the processing of two cells in a single step is only possible when the cells belong to a same diagonal plane. Therefore, two different strategies can be used to parallelize the sweep operation. The first strategy leads to parallelize the computations over angular directions (Line 5), while the second one leads to parallelize the computations over the cells on the same diagonal (Line 2). In the following, we are going to illustrate how to efficiently exploit both strategies in order to maximize the single-core performance of the sweep kernel, by vectorizing the computations.

Algorithm 6: The Sweep Operation

```
1  forall the $o \in \{1, \dots, 8\}$ do
2      forall the $c \in Cells$ do
3          ▷ $c = (i, j, k)$
4          $\delta x = \frac{2}{\Delta x}$;   $\delta y = \frac{2}{\Delta y}$;   $\delta z = \frac{2}{\Delta z}$;
5          forall the $\vec{\Omega} \in \{\vec{\Omega}_d^o,\ d = 1, \dots, M\}$ do
6              ▷ $\vec{\Omega} \equiv \{(\nu, \mu, \xi), \omega\}$
7              $\epsilon_x = \nu\ \delta x$;   $\epsilon_y = \eta\ \delta y$;   $\epsilon_z = \xi\ \delta z$;
8              $\psi = \frac{\epsilon_x \psi_x + \epsilon_y \psi_y + \epsilon_z \psi_z + S}{\epsilon_x + \epsilon_y + \epsilon_z + \Sigma_t}$;
9              $\psi_x = 2\psi - \psi_x$;
10             $\psi_y = 2\psi - \psi_y$;
11             $\psi_z = 2\psi - \psi_z$;
12             $\phi = \phi + \psi \cdot \omega$;
```

2.3 Vectorization over spatial domain

In this section, we discuss the design and implementation issues of the spatial vectorization through performance modeling. We will use the following definitions.

Definition 2 (Front)**.** *For a fixed octant, we define a front of order f as the list of all cells (a, b, c) such that $a + b + c = f$.*

Definition 3 (Step)**.** *A step processes a set of cells with a set of angular directions in a single instruction. Each of these sets may be reduced to a singleton.*

In the remaining of this section, without loss of generality, we focus on SSE the SIMD width of which is `packSize` $= 4$. For the sake of clarity, we first consider a 2D spatial domain, which size is defined by $n_x = 5$ and $n_y =$ `packSize`, as depicted in Figure 2.2.

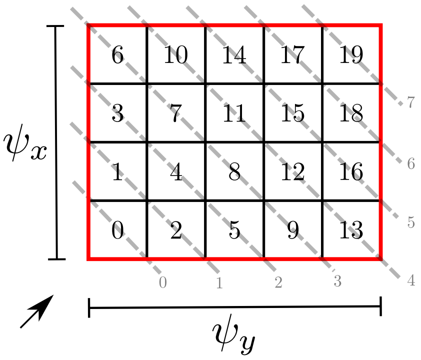


Figure 2.2: Sweep over a 5×4 2D spatial domain. It requires 20 scalar steps to go through the whole domain in scalar mode.

2.3.1 The algorithm

Figure 2.2 shows a sweep moving from the bottom-left corner of the 2D grid to the upper-right, where all the cells on the same front can be processed at once. Our aim here is to use vector instructions to parallelize the processing of all the cells belonging to the same front. However, the number of cells on the fronts is not constant. Hence, we cannot directly use vector instructions as they operate on fixed-size registers. For this reason, we must add padding cells to the fronts $0, 1, 2$ and $5, 6, 7$ in order to have fixed-size vectors that fit the SIMD width of the target processor. This strategy is illustrated on Figure 2.3. It shows that as the sweep moves from the bottom-left to the upper-right corner, each step consists in the process of a block of `packSize` cells in vector mode. The neutron currents along the x dimension ψ_x have the same size as in the scalar algorithm. Thus, they can be read once and kept in the SIMD registers until the sweep is finished. One should note that, as we do not store the neutron currents, ψ_x is overwritten after each step, and thus it must be reset to the correct value before the processing of fronts of order $1, 2$ and 3:

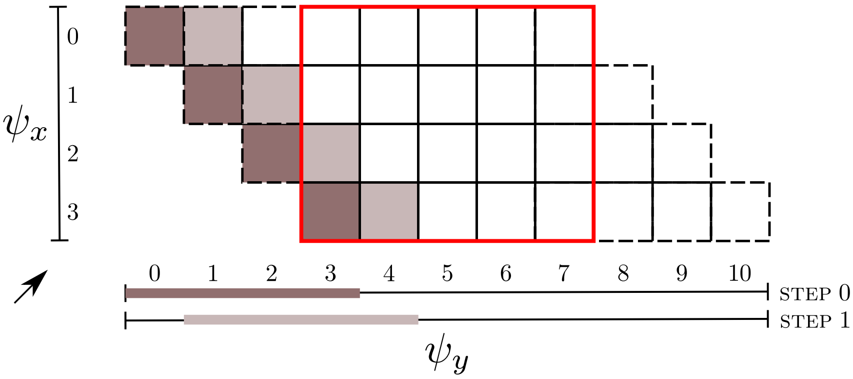


Figure 2.3: Vectorization of the sweep operation according to the spatial variable in 2D.

$$\psi_x[\texttt{packSize} - 1 - f] = 0, \quad f = 0, \dots, \texttt{packSize} - 1,$$

where f is the front order. The situation is a little different along the y dimension. Indeed, the processing of the block of cells at step 0 requires to load data indexed by $0, 1, 2$ and 3 from the ψ_y buffer. This can be done using an aligned load provided that the buffer is correctly aligned to cache boundary. However, at the step 1, we have to load data indexed by $1, 2, 3$ and 4: we cannot use anymore an aligned load to retrieve that data. This is a first limitation of this algorithm as an unaligned load operation has an overhead relative to an aligned load operation[1].

2.3.2 Maximum theoretical speed-up

The considered example requires $n_x n_y = 20$ scalar steps to process the sweep for one quadrant, in scalar mode. In vector mode, the number of steps is equal to the number of front, that is $n_x + n_y - 1 = 8$. Thus, the maximum theoretical speed-up brought by this spatial SIMD is $s_{\max} = 20/8 = 2.5$. In general, the formula giving the speed-up is given by equation (2.3).

$$s_{\max} = \frac{\texttt{packSize}\ n_x}{n_x + \texttt{packSize} - 1}, \tag{2.3}$$

If n_x is large, then the theoretical speed-up reaches its optimal value of `packSize`. In general, if $n_y >$ `packSize`, then the domain is split in slices of sizes of width equal to `packSize` along the y dimension. In this case, the efficiency formula becomes:

$$s_{\max} = \frac{n_x n_y}{(n_x + \texttt{packSize} - 1)\left\lceil \frac{n_y}{\texttt{packSize}} \right\rceil}, \tag{2.4}$$

where the ceiling takes into account the padding that should be used along the y dimension.

The generalization of the spatial SIMD algorithm in 3D is relatively straightforward. Indeed, we just need to loop over all the planes along the z dimension, and to vectorize the computations on each of these planes, which is exactly a 2D sweep as presented in section 2.3.1. Thus the theoretical efficiency in 3D remains also the same as in 2D (equation (2.4)).

To summarize, the spatial sweep algorithm as presented in this section, has only two limitations that are limiting its efficiency: unaligned load/store and additional padding cells. The performance results from a preliminary implementation were disappointing and have therefore confirmed the identified limitations. In the following section, we will present the angular vectorization strategy that allows us avoiding these limitations.

2.4 Vectorization over the angular variable

2.4.1 The algorithm

We consider that the neutron currents are not stored for maximizing the arithmetic intensity of the sweep. The vectorization of a cell processing during the sweep, along the angular directions, requires to move the loop over directions into the innermost level as presented Algorithm 6. Inside each spatial cell, we compute simultaneously several angular directions belonging to the octant currently being swept (the forall loop on Line 5 of Algorithm 6), using SIMD instructions which operate on packs of directions. Handling any angular quadrature order requires us to set up a padding system: for example, S_{16} Level Symmetric angular quadrature formula gives $16(16 + 2) = 288$ angular directions or $288/8 = 36$ directions per octant. When using single precision with Intel AVX, as 36 is not a multiple of 8, we perform 40 angular directions processing per spatial cell corresponding to an efficiency of $36/40 = 0.9$ (see Table 2.4). Except in the case

---

[1] It is expected that the penalty cost for an unaligned load/store operation is going to decrease for next generations of processors.

	N_{dir}	Directions per octant (M)			Sustainable speed-up	
		w/o padding	w/ padding			
			SSE	AVX	SSE	AVX
S_2	8	1	4	8	1	1
S_4	24	3	4	8	3	3
S_8	80	10	12	16	3.3	5
S_{12}	168	21	24	24	3.5	7
S_{16}	288	36	36	40	4	7.2

Table 2.4: Impact of the padding on the vectorization efficiency in single precision. The sustainable speed-up gives the maximum attainable speed-up taking into account the padding.

of S_2, where the vectorization does not give any improvement, the padding performance penalty is acceptable even for a quadrature formula featuring a small number of directions. For instance, using a S_4 quadrature, it is theoretically possible to speed-up the computations by a factor of 3 with SSE. In addition, using product quadrature formulas such as Gauss-Legendre, we have more flexibility to reduce the efficiency loss due to the padding. Indeed, one can choose a combination of the number of azimuthal and polar directions that gives a total number of directions divisible by `packSize`.

As mentioned in section 2.1.2, we use vectorized instructions provided by the Eigen library [47] to enforce the vectorization of the sweep kernel while maintaining the readability of the code. Listing 2.2 shows a snapshot of this vectorization, which features some constructs of Eigen. `Eigen::Array` is a class representing a matrix. The type of the matrix elements is defined by the first template parameter. In our case, we use this matrix class for representing a block of contiguous data, which size is equal to the SIMD width `packSize` of the target architecture. As mentioned previously, `Eigen::Map` is a class mapping an existing array of data which can be declared as well-aligned.

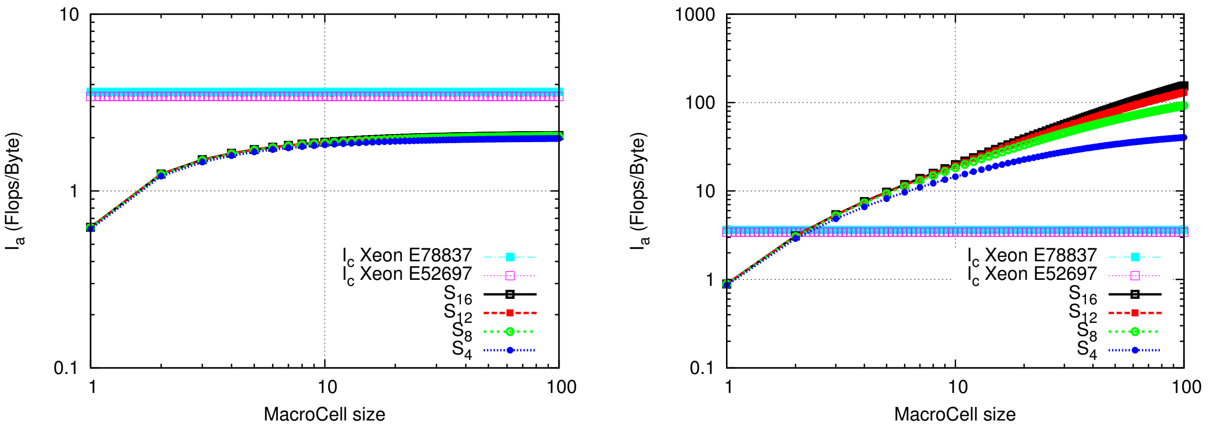
In the following, we discuss the performance the sweep operation vectorized over angular directions on Intel Westmere and Ivy Bridge microarchitectures. The performance measurements were carried out on the BIGMEM and ATHOS platforms.

2.4.2 Study of the arithmetic intensity of the sweep

To assess the advantage of not storing the neutron currents, we have numerically evaluated the arithmetic intensity of the sweep operation, defined by equations (2.1) and (2.2), as a function of the `MacroCell` size and for a different angular quadratures. The results are presented on Figure 2.4. It shows the variation of the arithmetic intensity of the sweep as a function of the `MacroCell` size and for different angular quadratures. The first observation that should be made here is that when the neutron currents are stored (Figure 2.4a), then independently of the angular quadrature used, the arithmetic intensity of the sweep operation is below the critical threshold I_c of both platforms. Thus, is this case, the sweep kernel is *memory bound*: its performance depends only on the sustainable bandwith of the target platform, and the vectorization will provide no performance gain. Contrariwise, when the neutron currents are not stored (Figure 2.4b), the arithmetic intensity of the sweep dramatically improves. Starting from a `MacroCell` of size $3\times3\times3$, I_a is over the critical treshold I_c for both processors: the sweep

Listing 2.2: The SIMD implementation of the sweep algorithm: `BlockArray` is a type representing an array of packed data.

```
...
typedef Eigen::Array<RealType, packSize, 1> BlockArray;
typedef Eigen::Map<BlockArray, Eigen::Aligned> BlockArrayView;
...

const BlockArray sijk(BlockArray::Constant(source[k][j][i]));

const int nblocks=directionPerOctant/packSize;
for (int b=0; b<nblocks; b++){
  ...

  // Load the neutron current according to direction
  BlockArrayView psiX(&psiXd[dir]);
  BlockArrayView psiY(&psiYd[dir]);
  BlockArrayView psiZ(&psiZd[dir]);

  BlockArray psiOut00(sijk);
  psiOut+= epsX*psiX+epsY*psiY+epsZ*psiZ;

  BlockArray denom(sigmaIJK);
  denom+=epsX;
  denom+=epsY;
  denom+=epsZ;

  psiOut/=denom;

  // Update of the scalar flux
  phi+=psiOut*ConstBlockArrayView(&weight[dir]);
}
```



(a) The neutron currents are stored

(b) The neutron currents are not stored

Figure 2.4: Single precision arithmetic intensity as a function of the `MacroCell` size ($n_x = n_y = n_z$) and angular quadrature.

kernel is then *cpu bound*, and its performance depends on the theoretical peak performance of the considered CPU. In this case, it makes more sense to vectorize the computations over angular directions. In the following, we are going to study the actual performance obtained on the test machines, showing the impact of vectorization.

2.4.3 Actual performances vs Roofline model

In this section, we first discuss the speed-up brought by the vectorized implementation of the sweep operation on Intel Westmere and Intel Ivy Bridge microarchitectures. Then, we compare the performances of the vectorized implementation against a theoretical performance model determined by the processor architecture: the Roofline model, as presented in [117].

Performances of the angular vectorization

SSE Table 2.5 shows the performances of both scalar and angular-vectorized single-core implementations of a sweep, using the SSE-enabled computing node BIGMEM. The performance measurements were obtained by averaging the computation time of ten successive runs. We observe that: first, when the spatial computational domain has a single cell, then the speed-up brought by the vectorization is only 1.14 using a S_4 quadrature, which is below the sustainable speed-up of 3 that is expected in single precision (see Table 2.4) because of the padding we used. This slow speed-up is justified by the fact that for a `MacroCell` of size $1 \times 1 \times 1$, as mentioned in section 2.2.2, the sweep kernel is *memory bound* and thus the computational gain brought by the vectorization is overshadowed by a slower DRAM bandwidth. The same observation applies to the cases of a sweep on a single cell with S_8, S_{12} and S_{16} quadratures. As expected, the speed-up brought by the angular vectorization increases with the `MacroCell` size, or equivalently with the arithmetic intensity, and reaches 2.4, 3.78, 4.23, 4.44 respectively for the four angular quadratures, when the `MacroCell` size is $64 \times 64 \times 64$ cells. One should note that the speed-up of 3.78 in the case of S_8 with $n_x = 64$ is larger than 3.3 which was expected (see Table 2.4). Indeed, by vectorizing the computations, the number of integer calculations required for loop indexing, for instance, are reduced in the same time, which can then lead to the observed superlinear speed-up.

	S_4			S_8			S_{12}			S_{16}		
n_x	T_{scalar} (ms)	T_{SSE} (ms)	(×)	T_{scalar} (ms)	T_{SSE} (ms)	(×)	T_{scalar} (ms)	T_{SSE} (ms)	(×)	T_{scalar} (ms)	T_{SSE} (ms)	(×)
1	0.048	0.042	1.14	0.033	0.032	1.03	0.036	0.032	1.12	0.031	0.032	0.96
2	0.036	0.035	1.02	0.041	0.035	1.17	0.048	0.036	1.33	0.053	0.036	1.47
4	0.056	0.043	1.30	0.092	0.051	1.80	0.149	0.060	2.48	0.204	0.069	2.95
8	0.210	0.103	2.03	0.499	0.154	3.24	0.938	0.242	3.87	1.374	0.326	4.21
16	1.555	0.730	2.13	3.857	1.111	3.47	7.377	1.808	4.08	10.88	2.472	4.40
32	11.04	4.501	2.45	29.43	7.567	3.88	57.42	14.30	4.01	86.05	18.94	4.54
64	82.21	34.22	2.40	222.3	58.70	3.78	436.8	103.2	4.23	649.9	146.2	4.44

Table 2.5: Illustration of the speed-up brought by the angular vectorization of the sweep, as a function of the `MacroCell` size ($n_x = n_y = n_z$), and for different angular quadrature orders. The performance measurements were carried-out on an Intel Xeon E78837 processor.

AVX We performed the same study as in the previous paragraph, changing only the target architecture to use an Intel E52697 V2 processor and the results are reported on Table 2.6.

n_x	S_4			S_8			S_{12}			S_{16}		
	T_{scalar} (ms)	T_{AVX} (ms)	(×)	T_{scalar} (ms)	T_{AVX} (ms)	(×)	T_{scalar} (ms)	T_{AVX} (ms)	(×)	T_{scalar} (ms)	T_{AVX} (ms)	(×)
1	0.038	0.043	0.8	0.027	0.032	0.84	0.027	0.032	0.84	0.028	0.032	0.87
2	0.031	0.035	0.8	0.032	0.034	0.94	0.037	0.035	1.05	0.041	0.035	1.17
4	0.043	0.043	1.0	0.066	0.045	1.46	0.105	0.049	2.14	0.140	0.059	2.37
8	0.144	0.108	1.33	0.342	0.140	2.44	0.638	0.171	3.73	0.934	0.239	3.90
16	1.062	0.761	1.39	2.635	0.985	2.67	5.004	1.243	4.02	7.362	1.779	4.13
32	7.335	4.806	1.52	19.90	6.671	2.98	38.76	8.684	4.46	57.67	13.13	4.39
64	59.28	37.00	1.62	157.3	51.85	3.03	308.3	68.09	4.52	458.5	105.9	4.32

Table 2.6: Illustration of the speed-up brought by the angular vectorization of the sweep, as a function of the `MacroCell` size ($n_x = n_y = n_z$), and for different angular quadrature orders. The performance measurements were carried-out on an Intel Xeon E52697 V2 processor.

We observe that the computation time in scalar mode on the Intel Xeon E52697 V2 is slower than that was obtained on the Intel Xeon E78837. For instance, the sweep in scalar mode an a `MacroCell` of size 64 × 64 × 64 with a S_4 angular quadrature takes 82.21 ms on the Intel Xeon E78837 processor compared to 59.28 ms on Intel Xeon E52697 V2 processor, which represents an improvement of a factor of 27.8%. To justify this factor, one should consider the improvements brought by Ivy Bridge microarchitecture, including: higher DRAM bandwidth, larger cache sizes and the new μop cache for storing the decoded instructions[1]. These new capabilities of the Ivy Bridge could therefore boost the throughput of instructions and thus the performances of a computational kernel. Although we have not quantified the individual effects of each of these features, the improvement we observed complies with the study in [59], where the authors observed an average speed-up of 33% over a series of benchmarks, between Westmere and Sandy Bridge architectures.

The second observation is that, as on the Westmere node, the speed-up brought by the vectorization increases with the `MacroCell` size and the angular quadrature. However, this speed-up is lower than expected. For instance, let us consider the performance of a sweep of a `MacroCell` of size 64 × 64 × 64 with a S_{16} quadrature. Using AVX we got a speed-up of 4.32, relative to a scalar execution, which is less than the expected speed-up of 7.2 (see Table 2.4). One should keep in mind that, as explained in [114], a floating point division performs slightly worse on Sandy Bridge than on a Westmere. Thus, if a AVX-vectorized code uses floating point divisions, then its performance may not correspond to the optimal speed-up that could be expected.

Roofline model

Here, we compare the actual single-core performance of the vectorized implementation of the sweep operation against the Roofline model [117] which defines the maximum performance attainable on a given multicore architecture. This Roofline model uses two parameters (the sustainable DRAM bandwidth $\mathcal{B}$, and the theoretical peak performance $\mathcal{P}$ of the target processor) to evaluate the maximum attainable performance by a kernel, as a function of its arithmetic intensity I_a. The evaluation of this Roofline model is defined by equation (2.5).

$$\text{Performance} = \min\left(\mathcal{P}, I_a * \mathcal{B}\right). \tag{2.5}$$

---

[1] A comprehensive study on processor microarchitecture is available from http://agner.org/optimize/microarchitecture.pdf

In the following, as we are focusing on the single-core performances, we consider a particular case of the Roofline model by evaluating its parameters for a single core.

Figure 2.6 shows the performance variation of the vectorized implementation of the sweep on the Intel Xeon E78837 processor, as a function of the arithmetic intensity I_a, and for different quadrature order. As mentioned previously, we observe that the performance of the sweep increases with I_a and the order of angular quadrature used. Furthermore, the actual performances follow the same trend as the predicted performance from the Roofline model, and reach a plateau the value of which depends on the quadrature: for a S_{16}, the plateau is 13.6 Gflop/s, corresponding to 62.9% of the theoretical peak performance per core of the BIGMEM node. One should note that the saturation of actual performances occurs a little later after the ridge point indicated by the Roofline model. In addition, the discrepancies between actual performances and the upper bound given by the Roofline model, are larger for lower values of I_a ($<$ 20 Flop/Byte). For instance, using a S_{16} quadrature, the ratio between the prediction of the Roofline model and the actual performance of the sweep is 20.9 for $I_a = 0.89$ and of 1.5 when $I_a = 154.6$. Indeed, our evaluation of the flops for the sweep operation (25 flops per cell per angular direction) takes into account only floating point operations performed in the computational kernel, and thus ignores all the remaining operations performed in the code. This hypothesis is valid, provided that the `MacroCell` in use is large enough to provide a large amount of computations in the kernel to counterbalance the remaining operations in the code. Consequently, for low values of I_a the flops are underestimated which explains the large discrepancy between the Roofline model and actual measurements.

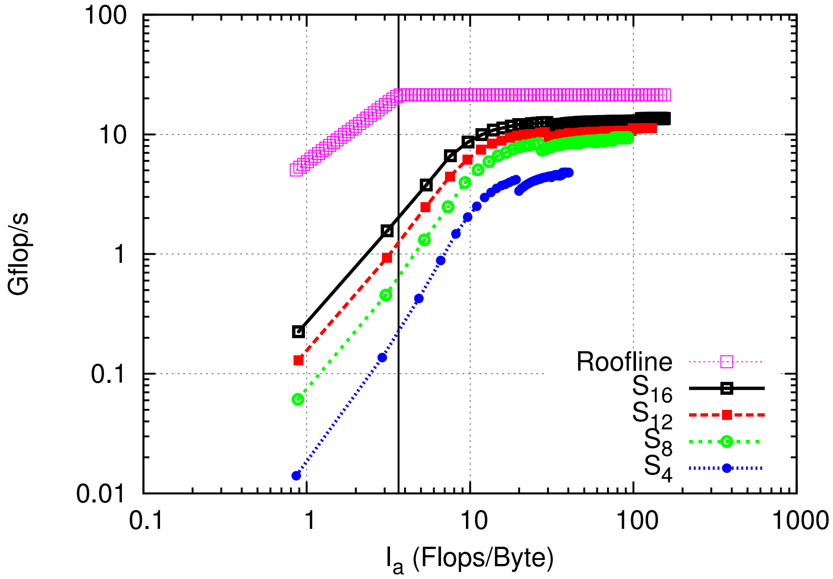


Figure 2.5: Roofline model for Intel Xeon E78837 processor and measured performances of the sweep operation as a function of the arithmetic intensity I_a. The variation of I_a is obtained by varying size of `MacroCells` as in Figure 2.4b.

The same study was performed on the Intel Xeon E52697 V2 processor (Figure 2.6), and yields similar conclusions. One should note that the saturation of the performances on the Intel Xeon E52697 V2 processor occurs nearly for the same value of I_a as on the Intel Xeon E78837 processor. Indeed, the critical arithmetic intensities of both processors are approximately the

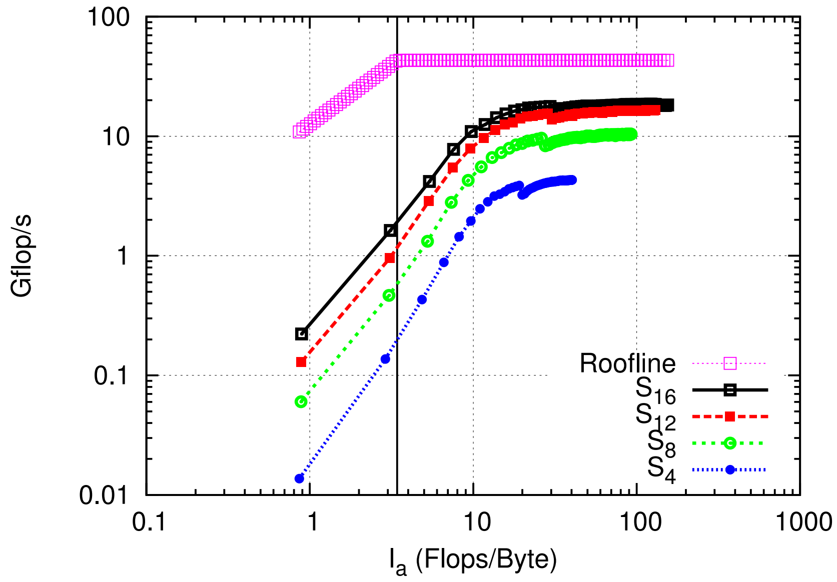


Figure 2.6: Roofline model for a Intel Xeon E52697 V2 processor and performances of sweep operation as a function of the arithmetic intensity I_a. The variation of I_a is obtained by varying size of `MacroCells` as in Figure 2.4b.

same (respectively 3.4 and 3.6 Flop/Byte), which justify the similarity of the observed trends.

▶ In this chapter, we presented a first parallel implementation of the sweep operation targeting modern SIMD-enabled multicore processors. We first studied the arithmetic intensity of the sweep operation depending on whether the neutron currents are stored or not, for different order of the angular quadrature and for different spatial mesh resolution. We found out that when the neutron currents are not store, then the arithmetic intensity is dramatically improved which allowed us to efficiently vectorize the sweep operation. While the spatial vectorization is a promising algorithmic strategy, its efficiency is theoretically limited by the required padding to enforce data alignment. Meanwhile, the angular vectorization gives good performances on our test machines. For instance, on the SSE-enabled Intel Xeon Westmere processor, the angular-vectorized implementation of the sweep achieves 13.6 Gflop/s when using a S_{16} angular quadrature and `MacroCell` of size $100 \times 100 \times 100$, corresponding to 62.9% of the theoretical peak performance per core of that processor. In practice, we intend to use vectorized kernels in parallel. We will see in following sections that this additional level of parallelism is applied to other `MacroCells` that can be processed in parallel. Indeed, the smallest the `MacroCells` are, the biggest the parallel potential. Hence, the optimal macrocell size is a trade-off between the kernel performance and the parallel efficiency.

Comforted with this high performance per core, we are going to present in Chapter 3, a theoretical study of the sustainable performance of the sweep operation on clusters of multicore processors.

Chapter 3

Performance Modelization of a Parallel Sweep

Contents

3.1 Preliminary definitions 40
3.2 Literature review on performance models of the sweep 42
  3.2.1 Flat models 42
  3.2.2 Hybrid models 44
3.3 New performance model of a parallel sweep 44
  3.3.1 Computation steps 45
  3.3.2 Communication steps 48
3.4 Asynchronous simulator of the sweep 50
  3.4.1 General presentation of the simulation algorithm 51
  3.4.2 Communication costs 53
3.5 Comparative study of the performance models 55
  3.5.1 Parameters of the models 55
  3.5.2 Evaluation of the performance models 57
    3.5.2.1 Optimizing process grid and task-granularity 57
    3.5.2.2 Performance comparisons of the models 58
    3.5.2.3 Impact of the scheduling overhead in the Hybrid-Async model 60

Our goal is to develop a highly efficient parallel neutron transport solver on massively parallel architectures. To achieve this goal, we rely on the discrete ordinates method (S_N) for solving the neutron transport equation. The vast majority of computations, within the S_N method, is spent in the sweep operation. Therefore, it is essential to first design a highly efficient parallel sweep algorithm. Considering this requirement, we adopted a bottom-up process by first answering the question on how to maximize the floating performance of the sweep operation on a single computing core. In Chapter 2, we found that for today and future architectures, to maximize the performance per core, it is essential to make use of the current SIMD vector units. In this chapter, we move to the upper-level to study the parallel performance of the sweep operation on distributed multicore architectures.

A substantial amount of work on the neutron transport theory, for the last decade, was focused on the development of sweep algorithms that are capable of maintaining a high efficiency on large number of cores. In section 3.1, we will introduce basic notations that are going to be used to develop the theoretical performance models. Then, we present some previous work on the modelization of the sweep operation (section 3.2) before presenting the two major contributions we made on this performance modelization (section 3.3 and section 3.4):

1. a new performance model of the sweep, targeting distributed multicore systems;
2. a simulator of an asynchronous implementation of the sweep.

A comparative study of the theoretical models is given in section 3.5, where all performance measurements were carried-out using vector instructions corresponding to target architectures.

3.1 Preliminary definitions

We consider a 3D spatial domain $\mathcal{D}$, discretized into N_x, N_y, N_z cells along the x, y and z axis:

$$\mathcal{D} = [\![1, N_x]\!] \times [\![1, N_y]\!] \times [\![1, N_z]\!],$$

and a 3D angular discretization comprising $N_{\text{dir}} = 8 \times M$ directions, where M is the number of directions per octant. The sweep problem is defined as following:

Problem 1 (Parallel Sweep)**.** *Find the minimal execution time of a parallel implementation of the sweep, over a spatial domain $\mathcal{D}$, along N_{dir} angular directions, on distributed multicore supercomputers.*

The vast majority of computations performed during the resolution of the neutron transport problem, according to the S_N method, is concentrated in the sweep operation. Thereby, for the last decades, a substantial part of the researches on neutron transport simulations was dedicated to the development of scalable and efficient sweep algorithms. Successive works proposed different approaches to overcome the sweep problem, and focused mainly on the development of new strategies aiming to maintain high parallel efficiency on large number of cores. However, as presented in section 1.4, the architecture of today and future parallel computers has shifted from traditional machines. Thus, algorithms and traditional parallel programming models should be adapted accordingly. From a performance point of view, the efficiency metric must be used in addition with a complementary metric: the "speed of execution" of a given kernel, or Flop/s. Indeed, this latter metric characterizes how much an application uses the power of a given computer (see section 1.4.2).

Let us first give some definitions and present notations that are going to be used to describe the theoretical models. For convenience, all the notations are reported in Table 3.1, where the following variables are used:

- A_x, A_y and A_z are the number of cells along the x, y, z dimensions of a single `MacroCell` (see Definition 1);
- $S_x = N_x/A_x$, $S_y = N_y/A_y$, and $S_z = N_z/A_z$, the global number of `MacroCells` along each axis of the spatial domain;
- P, Q, and R, the number of processes along x, y and z axis
- $L_x = S_x/P$, $L_y = S_y/Q$, and $L_z = S_z/R$ the local number of `MacroCells` on each process, along each axis of the spatial domain.

Definition 4 (Task)**.** *A task represents the computation of all neutron fluxes on a single* `MacroCell`*, of Cartesian coordinates* (a, b, c)*, for a subset of angular directions,* A_m *(belonging to a single octant* o*), and for a subset of energy groups* A_g*. We denote this task as* $\mathcal{T}^{o,a,b,c,g}$*.*

In the remaining of this chapter, we consider only a sweep for one group. Thus, for the sake of clarity a task will be simply denoted by $\mathcal{T}^{o,a,b,c}$.

Definition 5 (Front)**.** *A front of order* f*, for a given octant* o *denoted* $\mathcal{U}_f^{\mathrm{o}}$*, is the set defined as following:*

$$\mathcal{U}_f^{\mathrm{o}} = \left\{ \mathcal{T}^{o,a,b,c} \quad | \quad a + b + c = f \right\}.$$

$\forall o,\ F = S_x + S_y + S_z - 2$ is the number of diagonal planes (front) of `MacroCells` per octant, on the whole domain. This allows us to define the list of tasks belonging to the front f, for all the octants,

$$\mathcal{U}_f = \bigcup_{o=1}^{8} \mathcal{U}_f^{o},$$

and the total number of tasks available in the sweep problem,

$$\mathcal{U} = \bigcup_{f=1}^{F} \mathcal{U}_f.$$

Proposition 1. *Assume that* $S_x \geq S_y \geq S_z$*. Then:*

$$\forall \mathrm{o}, \forall f,\ \left| \mathcal{U}_f^{\mathrm{o}} \right| \leq S_y \times S_z.$$

Proof. The map defined as:

$$\begin{aligned} \mathcal{F} \colon \mathcal{U}_f^{\mathrm{o}} &\to [\![1, N_x]\!] \times [\![1, N_y]\!] \\ \mathcal{T}^{o,a,b,c} &\mapsto \begin{pmatrix} b \\ c \end{pmatrix}, \end{aligned}$$

is an injection. Therefore, it follows that the number of elements in the set $\mathcal{U}_f^{\mathrm{o}}$ is less than that of $[\![1, N_x]\!] \times [\![1, N_y]\!]$ ■

In the remaining of this chapter we will assume that $S_x \geq S_y \geq S_z$.

		Adams *et al.*	Hybrid	Hybrid-Async
Problem definition	Size of spatial domain	$N_x \times N_y \times N_z$		
	Number of angular directions	N_{dir}		
	Number of nodes	N_{nodes}		
	Number of cores per node	N_{cores}		
Task granularity	`MacroCell` sizes	$A_x \times A_y \times A_z$		
	Directions	A_m	M	
	Energy groups	G	1	
	Sizes of faces	D_x, D_y, D_z		
Partitioning	Process grid	$P \times Q \times R$		
	Number of `MacroCell`s per process	$L_x \times L_y \times L_z$		
Useful variables	Number of stages	N_{stages}		
	Global communication time	T_{comm}	T^*_{comm}	
	Computation time per task	T_{task}		
	Grind time	T_{grind}		
	Makespan	T_{Adams}	T_{Hybrid}	$T_{\text{Hybrid-Async}}$
	Number of fronts	F		

Table 3.1: Notations used in the performance models of the sweep operation.

3.2 Literature review on performance models of the sweep

The BTE resolution represents a significant portion in the main applications targeted by the DOE's Advanced Simulation Computing[1], formerly known as ASCI [71]. This initiative led to numerous research activities on the resolution of BTE. An important part of these researches concerns the development of efficient parallel sweep algorithms, and can be categorized in two different classes: *Flat* and *Hybrid* models. The first class corresponds to models implemented by assuming a uniform view of a parallel computer as a collection of independent computing cores. *Flat* models are usually implemented by using the standardized message-passing interface MPI [46]. The second class integrates the hierarchy of modern distributed memory clusters as described in section 1.4.1, and are implemented using several parallel programming models together (message-passing and threading for instance). This section reviews some previous works on performance modelization of the sweep operation using *Flat* and *Hybrid* models.

3.2.1 Flat models

In [8], Koch, Baker, and Alcouffe have proposed the KBA algorithm which decomposes the 3D Cartesian grid onto a 2D process grid. This reference algorithm, in the field of S_N Cartesian neutron transport simulation, is also used in the neutron transport code DENOVO [28], developed at ORNL[2]. Moreover, the codes PENTRAN [26] and UNIC [63] partition the global problem onto a 3D virtual grid of $S \times A \times G$ processes, where S, A, and G represent respectively the number of processes allocated for the spatial, angular and energy decompositions. The KBA algorithm served as a basis for a large majority of parallel sweep algorithms on structured meshes.

---

[1] `http://www.lanl.gov/asc/`

[2] Oak Ridge National Laboratory

Structured meshes

For structured meshes, Hoisie *et al.* [55] extended the KBA algorithm parallel performance model, by taking into account both communications and computations that fit the SWEEP3D[1] MPI based application. Chaussumier in [108] studied the impact of software pipelining in the overlap of communications by computations. In [39], Azmy *et al.* give a method for finding the best way to decompose a given fixed-size problem, between angular and spatial decompositions, through performance models.

Recently in [1], Adams *et al.* proposed a generalization of the KBA algorithm on structured meshes. As opposite to the KBA algorithm which parallelizes the sweep operation over planes, Adams *et al.* model defines a *volumetric* decomposition of the spatial domain according to the x, y and z axis. Indeed, they introduced a 3D spatial domain decomposition onto a 3D process grid. They have also presented scheduling algorithms, for the sweep operation, proved optimal. In the following, we will refer to this model as Adams *et al.* model. Its general description is given as following. Let G be the number of energy groups; A_m and A_g the numbers of angular directions and energy groups per task; A_z the number of z-planes each process needs to compute before a communication step occurs. $N_k = L_z/A_z$ gives the number of communication steps per process. Hence, each task carries the computation of angular flux for A_m directions, A_g energy groups and $L_x \times L_y \times A_z$ cells. Note that the number of cells per task, according to the x and y dimensions, can be modified through aggregation factors A_x and A_y. According to Adams *et al.* model, the makespan of the the whole sweep is therefore:

$$T_{\text{Adams}} = N_{\text{stages}}\ (T_{\text{task}} + T_{\text{comm}}), \tag{3.1}$$

where T_{task} is the cost of a computation step; T_{comm} the time needed by a process to communicate its outgoing angular flux to its neighboring processes after each computation step; and N_{stages} the total number of computation steps required to perform the whole sweep:

$$N_{\text{stages}} = 2\underbrace{\left(\left\lceil \frac{P}{2} \right\rceil - 1 + \left\lceil \frac{Q}{2} \right\rceil - 1 + N_k \left(\left\lceil \frac{R}{2} \right\rceil - 1\right)\right)}_{N_{\text{fill}}} + \overbrace{8MGN_k/(A_m A_g)}^{N^p_{\text{tasks}}}. \tag{3.2}$$

N_{fill} is the minimum number of stages before a sweep front can reach the center-most processors, and N^p_{tasks} is the number of tasks per process.

A key point to note here is that this Adams *et al.* model is able to predict the *multigroup* sweep computation time rather than just for a single-group.

Unstructured meshes

Flat performance models for the sweep operation, on unstructured meshes, have been largely studied. Thus, Kerbyson *et al.* in [64] presented the first performance model for transport sweeps on unstructured meshes. A similar work has been carried out by Plimpton *et al.* as described in [95]. In [4], Kumar *et al.* explored scheduling strategies for a generalized sweep algorithm. Their algorithm is applicable to more general cases than radiation transport problems as it uses no geometric information about the mesh. In [21], the authors presented new algorithms for the sweep operation on unstructured meshes by designing algorithms that achieve overlap of communications by computations, as well as message buffering to reduce cost associated with the latency of parallel machines used.

---

[1] http://wwwc3.lanl.gov/pal/software/sweep3d/sweep3d_readme.html

Recent advances on parallel computer architectures motivated the research community to develop new strategies to leverage the full power of these modern architectures. This evolution trend, in the S_N neutron transport community, has led to the development of *Hybrid* sweep algorithms.

3.2.2 Hybrid models

In [119], Yan *et al.* proposed a performance model of the sweep operation for unstructured meshes, targeting multicore-based clusters. Their model favors the overlap of communications by computations by explicitly dedicating a single thread (master) per process to handle all communications performed locally by the parent process, while the remaining threads (workers) keep doing computations. To develop this model, they first represent the sweep algorithm as a Direct Acyclic Graph (DAG), before partitioning it into sets of vertices. Finally, these sets are evenly mapped on processes. According to this model, the sweep execution time is decided by the maximum running time of the master and workers. Their communication model is proportional to the number of processes, and does not integrates a dynamic overlap of communications by computations according to the problem size.

Except for the recent release of PARTISN[1], as presented in [9], which features a parallel implementation of the sweep using both MPI and OpenMP; and the Yan *et al.* sweep model, all the previously listed codes and theoretical models (see section 3.2.1) follow the classical *Flat* parallel programming model. Even if modern MPI implementations can directly take advantage of shared memory to synchronize two processes on the same node [45], it remains that some extra memory is used by MPI for managing communications [15] due to the double-buffering strategy. Moreover, as mentioned in section 1.4, each node of modern distributed multicore machines comprises an increasingly large number of cores. In addition, the main memory capacity per core is dropping by a factor of 30% every two years (see [88]), and on-chip memory bandwith is not increasing as fast as the available computational power. Thus, it is essential to employ a programming model capable of taking into account these constraints. From this perspective, with a *Hybrid* model, that combines the use of MPI to manage inter-node communications, and threads within a multicore node to synchronize local data, we can benefit of lower memory latency and data movement on each node, as synchronizations can be done via the shared memory without extra copies. Bearing this idea in mind, we introduce in the next section a new performance model targeting distributed multicore machines. We will refer to this model as Hybrid model.

3.3 New performance model of a parallel sweep

Considering the need to shift the parallel programming models from *Flat* to *Hybrid*, we aim at designing and implementing a new hybrid sweep algorithm for modern architectures. The algorithm we propose is given as following: loop over all fronts, and perform the following actions on each front f.

1. On each node, a single multithreaded process executes all local tasks that belong to the front f; each thread being bounded to a unique core. Thus, data-transfers between the tasks are achieved through shared-memory accesses.

---

[1] A Cartesian S_N neutron transport code developed at LANL

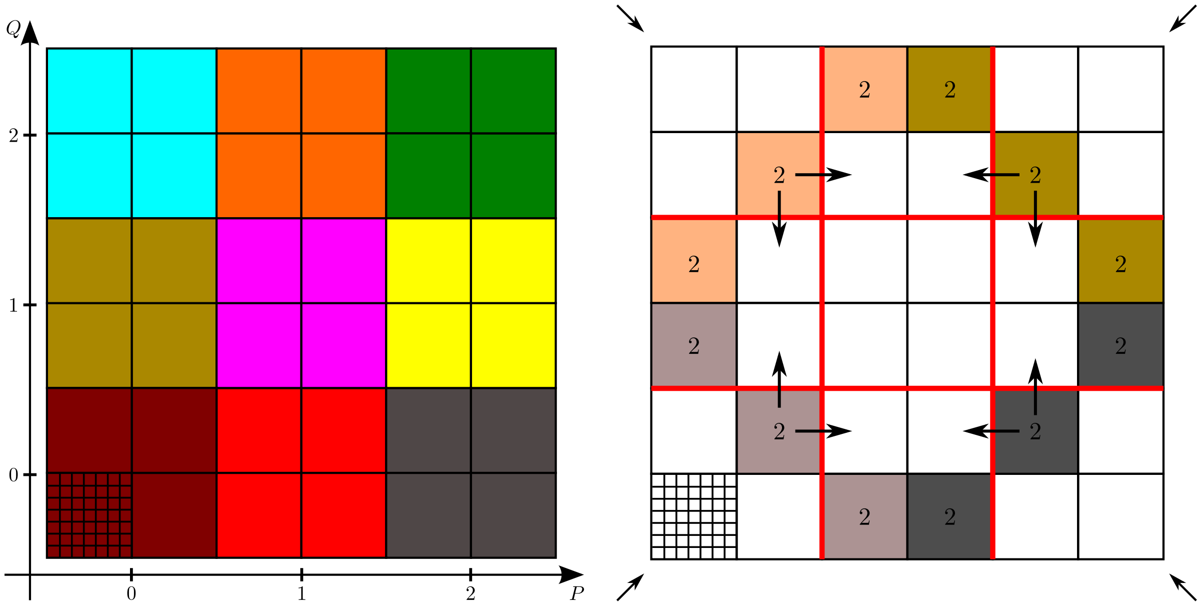
2. On each node, a single communication thread is dedicated to process all communications involving that node. Each communication step, occurring on the front f, can be processed while that node executes the tasks of the front $f+1$. Such a strategy enhances overlap of communications by computations as will be shown in section 3.3.2.

The Adams *et al.* model does not apply for such implementations. Thereby, we aim at extending this model to take into consideration the two levels of parallelism available on distributed multicore systems. The goal is twofold:

- validate the need for such an approach;
- provide us insight on the ideal process grid for a given problem and architecture.

To derive this model, we need to evaluate the new number of computation steps N_{stages}, and a new global communication time T^{*}_{comm}. Indeed, in this model not all steps induce a communication step anymore thanks to the shared memory accesses. Furthermore, in this model, the constraint on the coupling between computation and communication steps is released, allowing to minimize explicit global synchronizations. Thus, the makespan of the sweep operation in this Hybrid model is given by:

$$T_{\text{Hybrid}} = N_{\text{stages}} T_{\text{task}} + T^{*}_{\text{comm}}. \tag{3.3}$$



(a) Illustration of the blocked data distribution. `MacroCell`s of similar colors belong to the same process.

(b) Data distribution and communication pattern of the sweep for a 6 × 6 `MacroCell` grid, over a 3 × 3 process grid ($A_x = A_y = 7$; $S_x = S_y = 6$; $L_x = L_y = 2$; $F = 11$).

Figure 3.1: Simultaneous sweep of all quadrants. This shows that in this example M_2 is equal to 2.

3.3.1 Computation steps

A computation step is defined as the execution of a set of tasks in parallel. This set may be reduced to a singleton. To derive the formula giving the total number of computation steps, let us consider the following notations:

- For a given process grid (or domain partitioning) defined by (P, Q, R), $\mathcal{N}(p, q, r)$ is the node to which belongs the process of rank $r \times P \times Q + q \times P + p$. For the Hybrid model, there is a one-to-one mapping from the process list to the node list. Therefore, unless explicitly stated otherwise, we will use interchangeably process and node namings in the description of the Hybrid model

- $$\mathcal{N} = \bigcup_{(p,q,r)\in[\![1,P]\!]\times[\![1,Q]\!]\times[\![1,R]\!]} \mathcal{N}(p, q, r),$$

  the list of all nodes

- $\mathcal{T}(p, q, r)$ is the set of all tasks belonging to the node $\mathcal{N}(p, q, r)$:

  $$\mathcal{T}(p, q, r) = \left\{ \mathcal{T}^{o,a,b,c} \in \mathcal{U} \quad | \quad \frac{a_o}{L_x} = p,\ \frac{b_o}{L_y} = r,\ \frac{c_o}{L_z} = r \right\},$$

  where (a_o, b_o, c_o) represents the coordinates of the `MacroCell` (a, b, c) relative to the octant o

- $\mathcal{W}_f$ is the set of all nodes having at least one task of the front f, for all octants:

  $$\mathcal{W}_f = \left\{ \mathcal{N}(p, q, r) \quad | \quad \exists o,\ \mathcal{U}_f^o \cap \mathcal{T}(p, q, r) \neq \emptyset \right\};$$

  we will call each of those nodes a *working* node

- N_f^w is the number of tasks of the front f, for all octants, on the node $w = \mathcal{N}(p, q, r) \in \mathcal{W}_f$:

  $$N_f^w = |\{\mathcal{U}_f \cap \mathcal{T}(p, q, r)\}|$$

- N_f is the number of computation steps to process all tasks of the front f

- N_{cores} is the number of cores per node

These notations are defined in a general 3D case, but in the following, for the sake of clarity, we will rather use 2D figures.

Figure 3.1 shows the evolution of the sweep for all the four quadrants of a 2D spatial grid of 6×6 `MacroCell`. The computer used in this example is a cluster comprising nine nodes; each node having two computing cores dedicated to execute the tasks. For this example, there are eight nodes involved to process the front $f = 2$:

$$\mathcal{W}_2 = \{\mathcal{N}(0,0),\ \mathcal{N}(0,1),\ \mathcal{N}(0,2),\ \mathcal{N}(1,0),\ \mathcal{N}(1,2),\ \mathcal{N}(2,0),\ \mathcal{N}(2,1),\ \mathcal{N}(2,2)\}.$$

The largest number of ready tasks on a single node, for the front $f = 2$, is $\max_{w\in\mathcal{W}_2}(N_2^w) = 2$. Therefore, the number of computation steps required to process all tasks belonging to this front is $N_2 = \lceil 2/2 \rceil = 1$. In general, for a given working node w, if N_f^w is a multiple of $N_{\text{cores}} - 1$[1], then the number of computation steps required to process all tasks of the front f on the node w

---

[1] Note here that, as previously said, we do not use the full number of cores per node for task computations; rather a single core is entirely dedicated for communications.

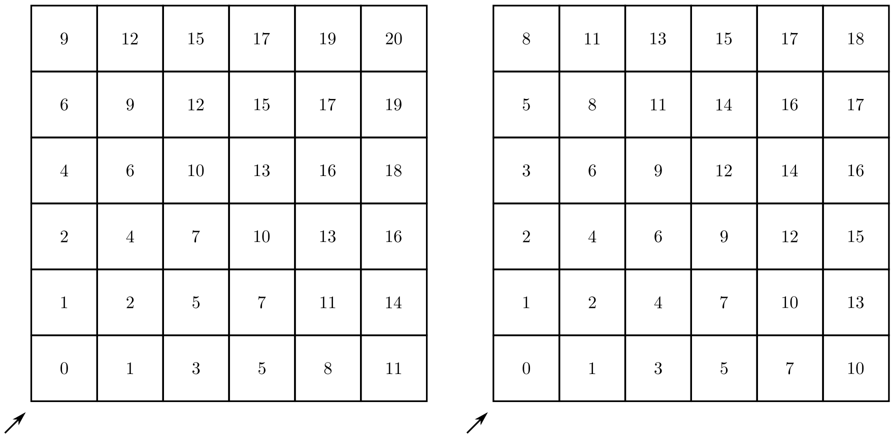
is exactly equal to the quotient of N_f^w by $N_{\text{cores}} - 1$. Otherwise, we could at least give an upper bound on the number of computation steps according to the equation (3.4).

$$N_f = \left\lceil \frac{\max_{w \in \mathcal{W}_f}(N_f^w)}{N_{\text{cores}} - 1} \right\rceil. \tag{3.4}$$

According to the Proposition 1, we have

$$N_f \leq \frac{N_y \times N_z}{N_{\text{cores}} - 1}.$$

Actually, considering the nearest superior integer when evaluating the number of computation steps is somewhat restrictive, as it overestimates this number, by implicitly requiring to finish all tasks of a given front before moving to the next. As an example, consider the sweep of the quadrant 0, over a 6 × 6 regular grid, using 2 parallel threads, as depicted in Figure 3.2. In this case, according to the equation (3.4), the complete sweep requires 21 steps. However, if we allow processing of tasks on $\mathcal{U}_{f+1}^0$, even if all tasks on $\mathcal{U}_f^0$ have not been yet finished, the same sweep takes 19 steps. This issue is solved by a simulator that will be presented in section 3.4.



9	12	15	17	19	20
6	9	12	15	17	19
4	6	10	13	16	18
2	4	7	10	13	16
1	2	5	7	11	14
0	1	3	5	8	11

(a) Upper bound: 21 steps

8	11	13	15	17	18
5	8	11	14	16	17
3	6	9	12	14	16
2	4	6	9	12	15
1	2	4	7	10	13
0	1	3	5	7	10

(b) Lower bound: 19 steps

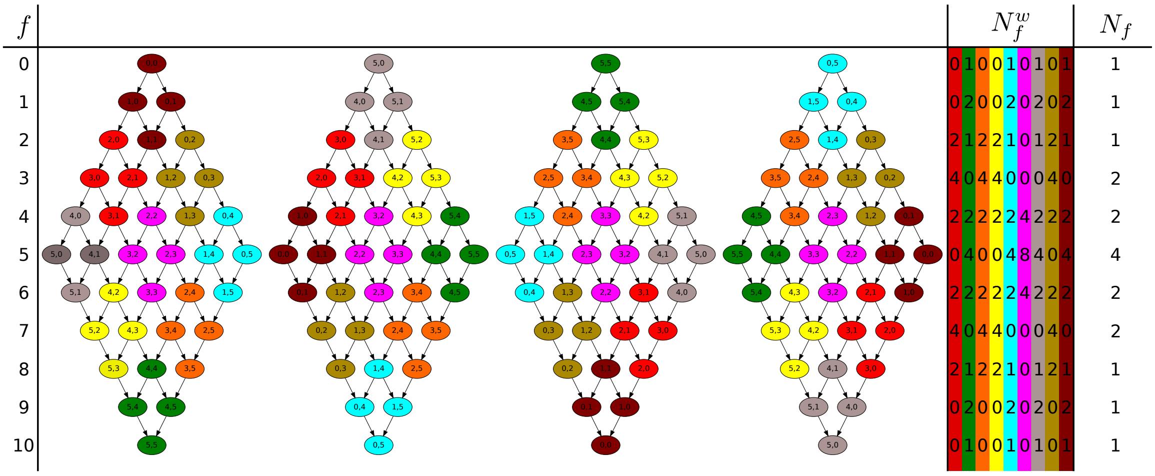
Figure 3.2: Illustration of the formula giving the number of computation steps, for a single quadrant sweep, over a 2D spatial grid of 6 × 6 `MacroCell`*s*. This sweep is performed on a dual-core computing node.

By summing up the equation (3.4) over all fronts, we obtain an upper bound of the number of computation steps required to process the whole sweep:

$$N_{\text{stages}} = \sum_{f=0}^{F-1} N_f. \tag{3.5}$$

Figure 3.3 shows a verification procedure of this formula, by counting the number of computation steps in a 2D case.

Evaluating the total number of the computation steps required to process the whole sweep algorithm is the first step toward a full modelization of the sweep execution time. Indeed, communications costs should be evaluated according to the process grid partitioning and the network characteristics of the target computer. The section 3.3.2 details strategies we developed to handle communications cost in the Hybrid model of the sweep.



f	N_f^w									N_f
0	0	1	0	0	1	0	1	0	1	1
1	0	2	0	0	2	0	2	0	2	1
2	2	1	2	2	1	0	1	2	1	1
3	4	0	4	4	0	0	0	4	0	2
4	2	2	2	2	2	4	2	2	2	2
5	0	4	0	0	4	8	4	0	4	4
6	2	2	2	2	2	4	2	2	2	2
7	4	0	4	4	0	0	0	4	0	2
8	2	1	2	2	1	0	1	2	1	1
9	0	2	0	0	2	0	2	0	2	1
10	0	1	0	0	1	0	1	0	1	1

Figure 3.3: Verification of the computation steps formula for a concurrent sweep of all quadrants over a 6×6 `MacroCell` grid, using nine dual-core computing nodes distributed into a 3×3 process grid. Tasks of similar colors are on the same node and follow distribution on Figure 3.1.

3.3.2 Communication steps

We define a communication step as a data transfer, from a process sender A to a remote process receiver B. The amount of data sent is equal to the size of a `MacroCell` face: $D_u,\ u = x, y, z$, times the number M of angular directions per octant, as presented by equation (3.6).

$$\begin{cases} D_x = A_y \times A_z \times M \\ D_y = A_x \times A_z \times M \\ D_z = A_x \times A_y \times M \end{cases} \tag{3.6}$$

Communication time on a single front

Let us first assume that every working node, $w = \mathcal{N}(p, q, r) \in \mathcal{W}_f$, has finished processing its set of ready tasks, $\mathcal{T}(p, q, r) \cap \mathcal{U}_f$. These nodes can therefore start processing tasks in $\mathcal{T}(p, q, r) \cap \mathcal{U}_{f+1}$, while local communication threads perform data transfer toward neighboring nodes.

Patterns of communications We assume that all nodes communicate in parallel, as depicted in Figure 3.1b. We denote by $T^*_{\text{comm}}(w, f)$ the communication time performed by the node w after processing all tasks of the front f. Then the global communication time for the front f is given by equation (3.7).

$$T^*_{\text{comm}}(f) = \max_{w \in \mathcal{W}_f} T^*_{\text{comm}}(w, f). \tag{3.7}$$

Due to our hypothesis, requiring to finish processing all tasks of the front f before a communication step can occur, $T^*_{\text{comm}}(f)$ should be evaluated on a node having executed the highest number of tasks. In the following, this node is denoted by $w_{\max}$:

$$w_{\max} \in \left\{ w \in \mathcal{W}_f \quad \middle| \quad T^*_{\text{comm}}(f) = \max_{w \in \mathcal{W}_f} T^*_{\text{comm}}(w, f) \text{ and } T^*_{\text{comp}}(f) = \max_{w \in \mathcal{W}_f} T^*_{\text{comp}}(w, f) \right\}.$$

Proposition 2. *For a given front, the set of nodes performing the highest number of communications and computations can be considered as a singleton.*

Proof. Let w_1, w_2 be two nodes belonging to this set. Recall that in this Hybrid model, it is required to finish all tasks in $\mathcal{U}_f$ before moving to the next front. Also we assume that any two different nodes can communicate in parallel, with their respective neighbors; and that there is no network contention on the message delivery. According to these requirements, communication times for the nodes w_1 and w_2 shall not change. Consequently, no matter on which node, from this set, we evaluate the communication time on the front f. ■

Cost of a single data transfer For a given front f, let $M_f^w(u)|u \in (x, y, z)$ be the number of `MacroCell` faces sent by the node w towards the u direction. Hence:

$$T^*_{\text{comm}}(w, f) = \sum_{u \in (x,y,z)} M_f^w(u)\tau_u, \tag{3.8}$$

where τ_u $u \in (x, y, z)$ is the time to required to complete the communication of a single face of a `MacroCell`, along the u direction. Estimating the communication time of a message over a distributed memory computer represents a tremendous research area. The linear communication model (see [54, 107]) is widely used. It is defined by equation (3.9).

$$\tau_u = \alpha + \frac{D_u}{\beta}, \tag{3.9}$$

where α and β are respectively the latency and bandwidth of the network interconnect, measured between any two nodes. This model is a good approximation of the communication time for two any computing nodes directly interconnected. However, on a typical cluster, the topology of the network interconnect may introduce a significant variation on this model. Indeed, in such an environment, the communication cost may be different depending on the considered pair of nodes as explained in [20], especially due to the message routing issues and the network contentions. A more elaborated communication model that takes into account concurrency over the network resources on nodes, useful to cope with large messages, is presented in [78, 82].

For our study, we altered the linear communication model as following:

$$\tau_u = \begin{cases} o_c \cdot D_u + T_{\text{cr}}(D_u), & D_u < d_0 \\ o_c \cdot D_u + \alpha + \dfrac{D_u}{\beta}, & D_u \geq d_0 \end{cases} \tag{3.10}$$

where $T_{\text{cr}}(d)$ returns the measured communication time between two nodes for a message of size d; and o_c is a fixed communication overhead per transferred byte. We use the network benchmarking utility NetPIPE [109], for evaluating $T_{\text{cr}}(u)$. d_0 is experimentally set to the message size from which the messages are transferred with the bandwith of the communication network, so that the linear communication model can be used to predict the communication cost.

Global communication time

The global communication time for the sweep operation is finally given by equation (3.11).

$$T^*_{\text{comm}} = \sum_{f=0}^{F-1} (1 - k_f) \max_{w \in \mathcal{W}_f} T^*_{\text{comm}}(w, f) \tag{3.11}$$

A part of them can be overlapped by computation threads. To take this effect into account, it is necessary to introduce an overlap rate, depending on the front being processed k_f.

Evaluating the overlap rate on a given front f, and on a given node w, requires to determine the amount of tasks on the front $f+1$. This overlap rate is evaluated on the node $w_{\max}$ as defined in section 3.3.2. If $T^*_{\text{comp}}(w_{\max}, f+1) > T^*_{\text{comm}}(w_{\max}, f)$ then all communications on the node $w_{\max}$ could be hidden. Otherwise, the amount of communications that could not be overlapped by computations is:

$$T^*_{\text{comm}}(w_{\max}, f) - T^*_{\text{comp}}(w_{\max}, f+1).$$

Consequently, the overlap rate is:

$$k_f = \frac{\max\left(T^*_{\text{comm}}(w_{\max}, f) - T^*_{\text{comp}}(w_{\max}, f+1), 0\right)}{T^*_{\text{comm}}(w_{\max}, f)}, \quad f = 0, \cdots, F-2. \tag{3.12}$$

Indeed, we do not evaluate k_f on the last front because there is no communication at this step.

3.4 Asynchronous simulator of the sweep

The Hybrid theoretical performance model presented in the section 3.3, under-exploits the available parallelism in the sweep operation. Indeed, this model requires to execute all tasks on a given front before moving to the next one. Such a procedure imposes an explicit synchronization, which can therefore leads to an under-utilization of the computational resources. To overcome this issue, we propose a new theoretical model having the following properties:

- On each node, a multithreaded process executes local tasks asynchronously
- On each node, a dedicated thread handle all local communications, as for the Hybrid model, hence enhancing the overlap of communications by computations
- A task can be processed as soon as it has received all its dependencies

Such a model allows to leverage more parallelism from an algorithm, and thus maximizes the occupation of computational resources. To study the efficiency and the performance of this parallelization strategy, we need to evaluate the number of computation steps, as well as the communication costs, by integrating the ability for a node to execute several tasks belonging to successive fronts. As one may observe, this model can not be easily described using a closed-form expression as we did for the Hybrid model.

There are some existing tools aiming to simulate the execution of an application on distributed multicore systems. For instance, SimGrid, presented in [18], allows to have an abstract model of a large-scale distributed system, so that the execution of a parallel application can be simulated on that system. However, such tools are more generic and versatile, and thus do not explicitly take into account specific particularities of a given algorithm, which can afford to simplify the simulation. Thereby, to simulate the execution of Hybrid sweep on distributed multicore systems, we rather designed a simulator that mimics the behavior of an asynchronous scheduler, that keeps executing tasks at earliest time, and tailored for the sweep algorithm. Such a simulator is exploited in modern task-based runtime systems (see Chapter 4). This section presents the design of this simulator, referred as Hybrid-Async in the following. Without loss of generality, the simulation algorithm is presented in a 2D case.

3.4.1 General presentation of the simulation algorithm

The theoretical performance model presented here predicts the computation time of the sweep operation on a distributed memory computer. This is achieved by simulating the execution of an asynchronous implementation of the sweep operation on such a computer, according to the problem size and the characteristics of computational resources used (number of nodes; number of cores per node; latency and bandwidth of the network interconnect). Such a strategy has the potentiality of better estimating the computation time, compared to a formula-based model as presented in section 3.3.

Preliminary considerations

Let us consider:

- a discretization of the time variable into intervals of length equal to the computation time of a single task, T_{task}:

$$\begin{cases} I_l = [l \cdot T_{\text{task}},\ (l+1) \cdot T_{\text{task}}[\,,\ l \geq 0 \\ [0, +\infty[= \bigcup\limits_{l \geq 0} I_l \end{cases} \tag{3.13}$$

- a global time, `clock`, common to all nodes, and set to 0 at the beginning of the simulation;
- a local time for the communication thread, $t_c^{(p,q)}$, giving the date at which the communication thread on the node $\mathcal{N}(p,q)$ is available for performing next communications.

A task $\mathcal{T}^{q,a,b}$ is represented by equation (3.14)

$$\begin{cases} \mathcal{T}^{q,a,b} := \{start,\ end,\ dep,\ T_{\text{task}}\} \\ \mathcal{T}^{q,0,0}.start = 0.0 \end{cases}, \tag{3.14}$$

where *start* and *end* represent the beginning and finishing execution dates of this task; *dep* is dependency number for this task. One should note here that this modelization corresponds to a simultaneous sweep of all quadrants. In fact, the beginning execution dates for the the four corner tasks ($\mathcal{T}^{q,0,0},\ q = 0, 1, 2, 3$), are set to 0.0. Consequently these tasks are executed at the beginning of the simulation.

Definition 6 (Task scheduling and execution)**.** *A task $\mathcal{T}^{q,a,b}$ is scheduled if all its dependencies have been removed:*

$$\mathcal{T}^{q,a,b}.dep = 0.$$

If $\texttt{clock} \geq \mathcal{T}^{q,a,b}.start$, then we say that the task $\mathcal{T}^{q,a,b}$ is ready to be executed. The execution of a ready task $\mathcal{T}^{q,a,b}$ allows to set its finishing execution date according to the equation (3.15):

$$\mathcal{T}^{q,a,b}.end = \texttt{clock} + T_{task}. \tag{3.15}$$

The simulation algorithm

The simulation engine encapsulates the behavior of an asynchronous implementation of the sweep operation, over a multicore distributed machine: local tasks are executed by a set of working threads (we assume one thread per computing core), and communications are carried out sequentially by a separate thread per node. Proposition 3 defines the main idea behind the simulation algorithm.

Proposition 3. *On each time spacing I_l, a given node could execute a maximum of $N_{cores} - 1$ tasks.*

Proof. Let w be a node with a set of n tasks ready to be executed at time `clock`, and assume that $n > N_{\text{cores}} - 1$. On this node, there are N_{cores} parallel threads, including a single one dedicated to communications. By assuming that the computation time of tasks is uniform, and equal to T_{task}, it follows therefore that $N_{\text{cores}} - 1$ tasks, among the n ready tasks, are going to be executed by $\texttt{clock} + T_{\text{task}} \notin I_l$. The remaining tasks could not be executed since the global time is out of I_l. ■

The main idea of the simulation algorithm (see Algorithm 7) is then to keep moving forward in time, and to perform the following action on each node: execute ready tasks on each node (Line 5) and update dependencies of adjacent ones; realize data transfers and schedule ready tasks for execution (Line 7 and Line 8). This process continues until all tasks are executed. The value of `clock` at the end of the simulation gives the sweep time.

Algorithm 7: General simulation algorithm of an asynchronous sweep in 2D

```
In  : Latency ($\alpha$), Bandwidth ($\beta$), Scheduling overhead ($\delta$), Communication overhead ($o_c$)
Out: Sweep execution time (clock)
1  clock = 0.0; ▷ Global clock
2  RemainingTasks = $4 \times S_x \times S_y$;
3  while RemainingTasks > 0 do
4      for $w = \mathcal{N}(p, q) \in \mathcal{N}$ do
5          ▷ Execute $n$ ready tasks on $\mathcal{N}(p, q)$: $\mathcal{T}_1^{q,a,b}, \cdots, \mathcal{T}_n^{q,a,b}$ ($n \leq N_{\text{cores}} - 1$)
6          RemainingTasks − = $n$;
7          ▷ Update the clock of the communication thread $t_c^{(p,q)}$
8          ▷ Perform communications and schedule ready tasks
9      clock+ = $T_{\text{task}}$;
10 return clock;
```

Scheduling and execution of tasks We recall that, as the sweep goes through the grid from a given corner to the far opposite corner, each processed task removes one dependency to one of its adjacent tasks. When all dependencies of a given task have been removed, then this task is scheduled for execution. It is worth to note here that for a sequential execution, only a single task can be released for execution per time step. In this case, the makespan of the problem is simply the total number of tasks times the computational cost of a single task. Opposingly, for a parallel execution, we can have more ready tasks to be executed than available computational resources. It therefore raises the following problem:

Problem 2 (Scheduling)**.** *Let an algorithm described by a task graph and a set of computational resources. Given a set of ready tasks at some execution time, find the most efficient way to schedule those tasks in order to minimize the makespan.*

This scheduling problem is a more general concept and it is subject to numerous studies in the parallel computing area. For the particular case of the Hybrid-Async model, the available parallelism during the sweep operation is dynamically discovered as the sweep moves throughout the spatial grid. We state the following requirements to achieve an optimal scheduling:

1. The execution of a set of ready tasks should contribute to release the highest number of tasks that can be released at this step;
2. Locally executed tasks should contribute to minimize idle time of neighboring nodes.

In the Hybrid-Async model, tasks are sorted in respect to these requirements. However, for a real implementation implementation on top of a generic runtime system (see section 4.3) for instance, such scheduling policies may introduce an overhead per task, depending on the heuristics used internally in the runtime in order to decide which tasks should be scheduled for next execution. We integrate this property in the simulator as following: once a given task $\mathcal{T}^{q,a,b}$ has received all its incoming dependencies from tasks $\mathcal{T}^{q,a,b-1}$ and $\mathcal{T}^{q,a-1,b}$ (see Figure 3.4), at a given time t_r, then it can be scheduled at the time $t_s = t_r + \delta$, where δ is the average scheduling overhead per task. This scheduling overhead can be interpreted as the delay needed by the runtime system to select the task for execution according to a given scheduling policy. Hence, the beginning execution date of the task $\mathcal{T}^{q,a,b}$ is updated using the equation (3.16):

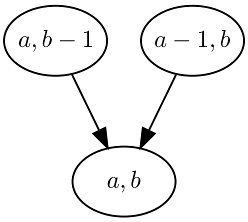


Figure 3.4: Dependencies for the execution of a task in 2D.

$$\mathcal{T}^{q,a,b}.start = \max\left(\mathcal{T}^{q,a-1,b}.end, \mathcal{T}^{q,a,b-1}.end\right) + \delta. \tag{3.16}$$

It is important to note here that this scheduling overhead per task induces some requirements on the task granularity: a task should be coarse enough such that the value of the scheduling overhead can be negligible compared to the task computation time. Otherwise, the performance of the implementation can be very poor. By the same time, the granularity of the task should allow to extract a large number of tasks, from the considered problem, to maximize occupation of computational resources. We will discuss the impact of this scheduling overhead on the performance in section 3.5.

3.4.2 Communication costs

When a given node $\mathcal{N}(p, q)$ has processed some of its ready tasks, the global time will be $\texttt{clock} + T_{\text{task}}$. At this time, a set of communications to be processed on this node is notified to the local communication thread. Because there may be some pending communications when this notification is sent, and due to the fact that local communications are handled sequentially, the communication thread can only start processing those communications at time:

$$t_c^{(p,q)} = \max\left(\texttt{clock} + T_{\text{task}}, t_c^{(p,q)}\right).$$

To show how each communication is handled inside the Hybrid-Async model, let us consider a task $\mathcal{T}^{0,a,b} \in \mathcal{T}(p, q)$ which releases one dependency to its neighboring task $\mathcal{T}^{0,a+1,b}$, along the x dimension. A communication must occurs therefore from the node $\mathcal{N}(p, q)$ to the node $\mathcal{N}(p+1, q)$. As said previously, a dedicated thread is going to handle sequentially all communications initiated on each node. For this reason, the considered communication on the node $\mathcal{N}(p,q)$ will be issued at time:

$$\max\left(t_c^{(p,q)}, \mathcal{T}^{0,a,b}.end\right),$$

and $t_c^{(p,q)}$ is updated to this time, to handle the sequentiality of communications. Finally, the cost of a message transmission between any two nodes follows the same communication model as we used for the Hybrid model (see section 3.3.2). Again, the communication thread is updated to take into account transmission of the message:

$$t_c^{(lp,lq)} \mathrel{+}= \tau_x.$$

Algorithm 8 presents the full Hybrid-Async model.

Algorithm 8: Asynchronous sweep simulator

```
In  : Latency ($\alpha$), Bandwidth ($\beta$), Scheduling overhead ($\delta$), Communication overhead ($o_c$)
Out: Sweep execution time (clock)
clock $= 0.0$; ▷ Global clock
RemainingTasks $= 4 \times S_x \times S_y$;
while RemainingTasks $> 0$ do
    for $w = \mathcal{N}(p,q) \in \mathcal{N}$ do
        ▷ Execute $n$ ready tasks on $w$: $\mathcal{T}_1^{q,a,b}, \cdots, \mathcal{T}_n^{q,a,b}$ $(n \leq N)$
        RemainingTasks $-=$ $n$;
        ▷ Update the clock of the communication thread $T_c$
        $t_c^{(lp,lq)} = \max\left(\texttt{clock} + T_{\text{task}}, t_c^{(lp,lq)}\right)$;
        ▷ Perform communications and schedule ready tasks
        for $\mathcal{T}^{q,a,b} \in \left\{\mathcal{T}_1^{q,a,b}, \cdots, \mathcal{T}_n^{q,a,b}\right\}$ do
            ▷ Communications
            if $\mathcal{T}^{q,a,b}.comm_x$ then
                $t_c^{(lp,lq)} \mathrel{+}= \tau_x$;
                $\mathcal{T}^{q,a+1,b}.start = \max\left(\mathcal{T}^{q,a+1,b}.start, t_c^{(lp,lq)}\right)$;
            if $\mathcal{T}^{q,a,b}.comm_y$ then
                $t_c^{(lp,lq)} \mathrel{+}= \tau_y$;
                $\mathcal{T}^{q,a,b+1}.start = \max\left(\mathcal{T}^{q,a,b+1}.start, t_c^{(lp,lq)}\right)$;
            ▷ Scheduling
            if $\mathcal{T}^{q,a+1,b}.ready$ then
                $\mathcal{T}^{q,a+1,b}.start \mathrel{+}= \delta$
            if $\mathcal{T}^{q,a,b+1}.ready$ then
                $\mathcal{T}^{q,a,b+1}.start \mathrel{+}= \delta$
    clock$+= T_{\text{task}}$;
return clock;
```

In this section, we presented two different but related algorithms of parallelizing a sweep, on distributed multicore systems: the Hybrid model and the Hybrid-Async simulator. In the next section, we present some performance studies to highlight the advantage of these two algorithms over classical *Flat* approaches.

3.5 Comparative study of the performance models

The goal of this section is to compare all performance models described in previous sections. We first explain how parameters used in the performance models are obtained, then we discuss the efficiency and performance of each model. These studies are realized using two different 3D test cases, and model parameters are set for the IVANOE platform (see Table A.1).

3.5.1 Parameters of the models

This section specifies measurements of the parameters used to benchmark the theoretical performance models.

Computer network performances For a given machine, we use the NetPIPE [109] utility to evaluate communication time for message of increasing sizes. This allowed us to evaluate the network performance of the IVANOE platform, and the result is presented in Figure 3.5.

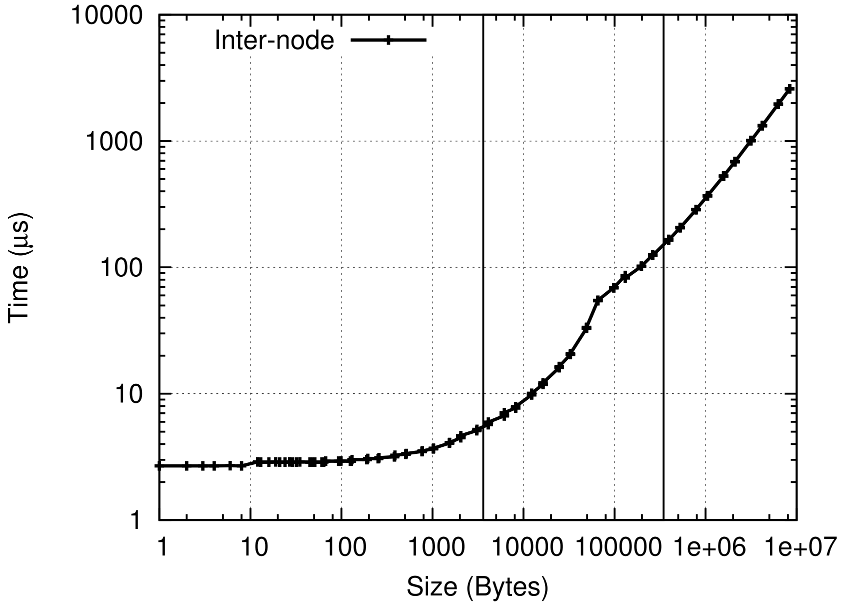


Figure 3.5: Performance of the network interconnect of the IVANOE platform. Measurements are obtained using NetPIPE utility, compiled with OpenMPI 1.6.5. The two vertical lines define the range to which belong messages sent through the network for the benchmarks considered in this study.

Task computing time In the neutron transport community, the computation time per spatial cell, per angular direction and per energy group is usually called *grind* time, denoted as T_{grind} in the following. Thus, the computation time of a single task can therefore be evaluated according to the number of spatial cells, angle and energy groups aggregated into it. This linear model of computation time per task ignores cache effects and bus memory contentions due to concurrent accesses; but it gives at least a lower limit on the task computation time.

In this study, the *grind* time is evaluated through experimental measurements. For each test case, we set this value to the average computation time per spatial cell and per angular direction, using a sequential implementation of the sweep operation. It is worth to note here that in practice, the *grind* time is not a linear function of the number of cells. To assess this hypothesis, we conduct an experiment consisting to evaluate the average performance per `MacroCell`, from the performance of a sequential implementation of the sweep operation, on the BIGMEM computing node (see Table A.1). To achieve this, we considered a test case featuring $5 \times 5 \times 5$ `MacroCells` of increasing sizes. This study is presented on Figure 3.6. We observe that the performance per `MacroCell` (or indirectly, per spatial cell) increases, non-linearly, with the number of cells and increases slightly starting from `MacroCells` of size $60 \times 60 \times 60$ cells, and corresponding to an average performance per `MacroCell` of 10.6 GFlop/s. As the `MacroCell` sizes (A_x, A_y and A_z) define the task granularity in all the performance models, one should be aware on the choice of the *grind* time. In the following, we use a single *grind* time per test case, obtained by averaging the computation time for a sequential run of sweep operation (see Table 3.2).

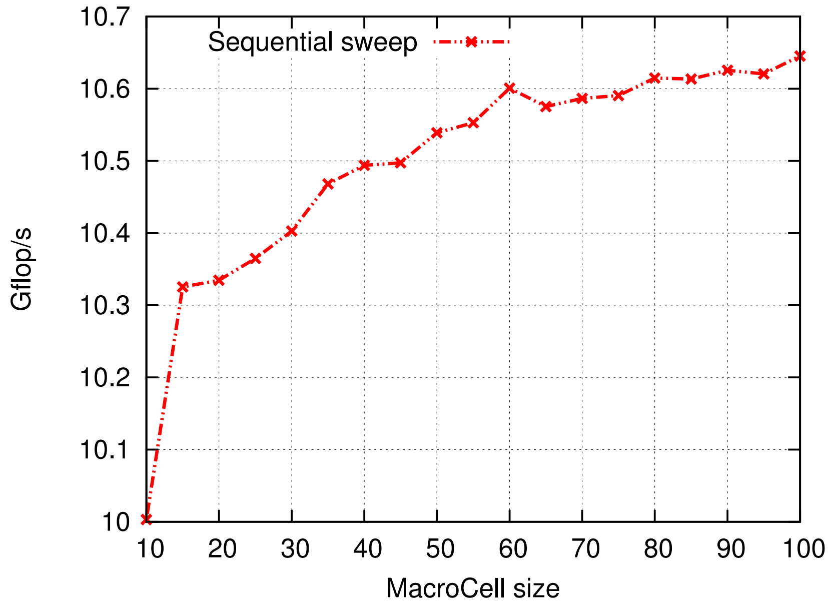


Figure 3.6: Average performance of the sweep operation per `MacroCell`, as a function of the `MacroCell` size. This experiment was carried out using a sequential implementation of the sweep operation on the BIGMEM computing node.

Communication and scheduling overheads

For a given computer, we choose the best values of these parameters that fit the sweep models to real execution times. It is worth to note that the scheduling overhead (δ) is strongly dependent on the runtime being used. We will use different values of δ to illustrate its impact on the performance when the task granularity is not quite large enough to amortize it.

We will first evaluate the performance models by considering that scheduling overhead per task, and communication overhead, are null.

3.5.2 Evaluation of the performance models

We consider two different test cases: `small` and `big` as described in Table 3.2. These test cases differ on their spatial mesh sizes; the total number of angular directions remain the same, and corresponds to a S16 Level Symmetric quadrature (288 angular directions). The `small` test case is intended to stress the behavior of theoretical performance models when there is not enough parallelism; while the `big` test case allows to perform a strong scalability study on large number of cores.

3.5.2.1 Optimizing process grid and task-granularity

We recall that the main goal of the developed performance models is to optimize the process grid partitioning for a set of computational resources and for a given test case. Thereby, for all the theoretical models, for a given test case and a given number of nodes (or processes), we select the optimal process grid partitioning that minimizes the global sweep time by exploring all the possible configurations. Furthermore, for the Adams *et al.* model, we select the best value of A_z that minimizes the total number of stages (equation (3.2)), by solving the Problem 3 (notations are defined in Table 3.1).

Problem 3 (Optimal sweep)**.** *The minimal computational time for the sweep operation for a given test case and a set of computational resources is reached for a given domain partitioning* $(P^{opt}, Q^{opt}, R^{opt})$*, and a given aggregation factors* $(A_x^{opt}, A_y^{opt}, A_z^{opt})$ *such that for all domain partitioning* (P, Q, R) *and for all* `MacroCell` *size* (A_x, A_y, A_z)*:*

$$T_{sweep}\left\{(P^{opt}, Q^{opt}, R^{opt}, A_x^{opt}, A_y^{opt}, A_z^{opt})\right\} \leq T_{sweep}\left\{(P, Q, R, A_x, A_y, A_z)\right\}.$$

Adams *et al.* model It is worth to note here that, for the Adams *et al.* model, for a given process grid partitioning, the `MacroCell` sizes along the x and y dimensions are set respectively to N_x/P and N_y/Q as the authors did. Indeed, optimizing A_x and A_y makes more sense when the local subdomain has to be processed by a multithreaded process, as for Hybrid and Hybrid-Async models, in order to adjust the amount of local tasks, for efficiency purposes.

Hybrid model and Hybrid-Async simulator To determine the optimal `MacroCell` size for the Hybrid model and the Hybrid-Async simulator, we must have a measure of the scheduling overhead per task. Indeed, assuming that the task scheduling is negligible, it is clear that the best decomposition is therefore the one that will extract a large amount of parallelism from the considered problem. Therefore, the optimal `MacroCell` will be a single cell, which does not correspond to our experimental studies as we are going to see in Chapter 4. This suggests that there is an optimal lower bound for the `MacroCells`, which can be determined for instance by an estimatation of the scheduling overhead per task. Without having to define this overhead, we can meanwhile compare the performance predictions from our models (Hybrid model and Hybrid-Async simulator) with that of Adams *et al.* model, by searching for the optimal `MacroCell` size to be not less than the optimal task size returned by the Adams *et al.* model, and defined by optimal aggregation factors $(A_x^{\text{opt}}, A_y^{\text{opt}}, A_z^{\text{opt}})$.

3.5.2.2 Performance comparisons of the models

In this section we propose a comparative study of theoretical models presented in previous sections, regarding performance and scalability using both `small` and `big` test cases (see Table 3.2). Experiments are carried out using settings for IVANOE platform.

		`small`	`big`
N_{dir}		288	288
Discretizations	$N_u \| u \in (x, y, z)$	120	480
T_{grind} (ns)		2.12	1.92
GFlops		12.44	796.26

Table 3.2: Characteristics of the test cases used. GFlops is the number of floating point operations required for one complete sweep. We count 25 floating point operations per spatial cell per angular direction (see section 1.3.3).

Hybrid model Figure 3.7 shows the performance of the sweep operation as predicted by the Hybrid model, described in section 3.3, using the `small` test case. This experiment was carried out using parameters of the IVANOE platform, and $N_{\text{cores}} - 1$ computing cores per node; 1 core is set free for the communication thread. This figure presents the performances of the sweep operation for three different settings, regarding the overlap rate of communications by computations. The first case corresponds to an implementation where there is no overlap of communications by computations, which underestimates the performances. In this case, the predicted performance when using 768 cores is 1.96 Tflop/s, corresponding to a parallel efficiency of 23.8%, relative to the performance on a single node (12 cores). In the second case, where we assume a full overlap of communications by computations, the predicted performances are overestimated. More precisely, the performance and the efficiency become respectively 3.5 Tflop/s and 42.3% representing an improvement of a factor of 1.7. However, it is not always possible in practice to overlap all communications by computations. Indeed, as explained in section 3.3.2, this is only possible when there are enough computational tasks per node to hide the progress of data transfers through the network interconnect. To illustrate this, let us consider the Hybrid model prediction with an adaptive overlap rate for 64 nodes (768 cores), which slightly underestimates the performances because the computation steps is overestimated (see 3.3.1). For that number of nodes, the Hybrid model found that $(15, 10, 5)$ is the optimal size of the `MacroCell` (see Table 3.4), which gives a total of 18432 tasks, and thus only 288 tasks per node to be shared by 11 computation cores (≈ 26 tasks per core). Bearing in mind that these tasks are not ready at the same time, it is therefore not possible to achieve a high overlap rate when the number of cores is large. This is what is shown by the predicted performances from Hybrid model with an adaptive overlap rate. Up to 96 cores (960 tasks per core), all communications are hidden; and beyond that point, the overlap rate decreases until 37.5% at 768 cores. Indeed, for this point, there are only 18 tasks per core.

Note 1. *Considering the results of Hybrid using the* `small` *test case on* 768 *cores, we observe that when using* `MacroCells` *of size* $(5, 5, 5)$ *there are* 216 *tasks per process, that is* ≈ 20 *tasks per core, which is largely enough to keep busy all the computing cores. However, even without taking into account scheduling overhead, the Hybrid model returned* $(15, 10, 5)$ *as the best* `MacroCell` *size, which corresponds to* 36 `MacroCells` *per process. The reason is that communications are*

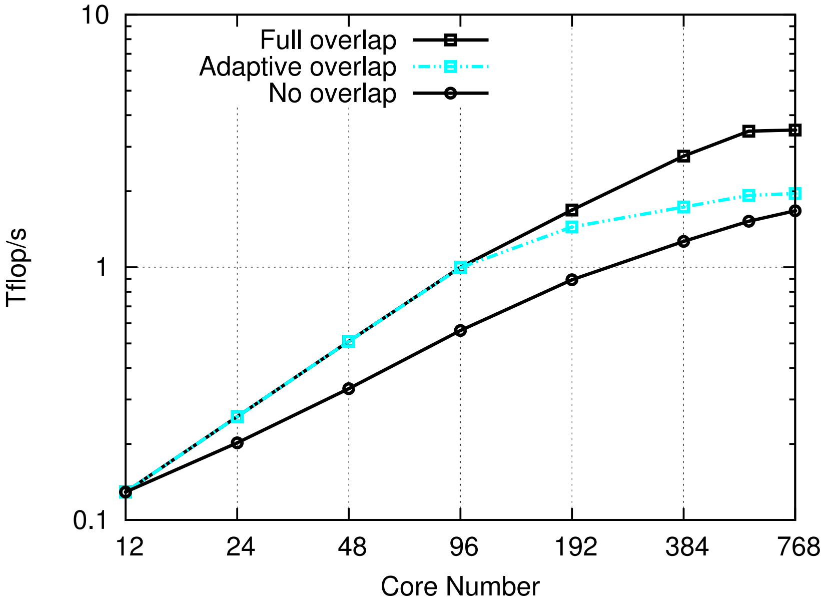


Figure 3.7: Performance of the Hybrid model using the `small` test case on IVANOE. This figure illustrates how the Hybrid model integrates the overlap of communications by computations.

much more dominant at this point, and therefore by decreasing task granularity, the amount of communications is larger.

In the following, all results with the Hybrid model are evaluated using an adaptive overlap rate.

	Partitioning			Aggregation			N_{tasks}	Message size (KB)		
N_{cores}	P	Q	R	A_x	A_y	A_z		s_x	s_y	s_z
12	3	2	2	40	60	15	384	126.5	84.3	337.5
24	6	2	2	20	60	15	768	126.5	42.2	168.7
48	6	4	2	20	30	10	2304	42.2	28.1	84.3
96	8	6	2	15	20	10	4608	28.1	21.1	42.2
192	12	8	2	10	15	10	9216	21.1	14.0	21.1
384	24	8	2	5	15	5	36864	10.5	3.5	10.5
576	24	12	2	5	10	5	55296	7.0	3.5	7.0
768	24	8	4	5	15	5	36864	10.5	3.5	10.5

Table 3.3: Optimal domain partitioning and aggregation factors for the Adams *et al.* model using `small` test case on the IVANOE platform.

Adams *et al.*, Hybrid and Hybrid-Async models Figure 3.8 shows a comparison of the performance as predicted by the Adams *et al.*, Hybrid and Hybrid-Async models on the `small` test case. On a single node, the performances of all the three models are similar. Up to 96 cores, Hybrid and Hybrid-Async achieve nearly the same performances. Indeed, as explained previously, when the number of tasks per core is relatively high, the Hybrid model can perfectly

	Partitioning			Aggregation			N_{tasks}	Message size (KB)		
N_{cores}	P	Q	R	A_x	A_y	A_z		s_x	s_y	s_z
12	1	1	1	5	5	5	110592	3.5	3.5	3.5
24	2	1	1	5	5	5	110592	3.5	3.5	3.5
48	2	2	1	5	5	5	110592	3.5	3.5	3.5
96	2	2	2	6	5	5	92160	3.5	4.2	4.2
192	4	2	2	10	10	5	27648	7.0	7.0	14.0
384	4	4	2	10	10	5	27648	7.0	7.0	14.0
576	6	4	2	10	10	5	27648	7.0	7.0	14.0
768	8	4	2	15	10	5	18432	7.0	10.5	21.1

Table 3.4: Optimal domain partitioning and aggregation factors for the Hybrid model using `small` test case on the IVANOE platform.

overlap almost all communications by computations. For the same number of cores, we observe that the performance of Hybrid and Hybrid-Async is 1.4 times faster than the performance of Adams *et al.* model. This is due to the fact that the Adams *et al.* model involves much more communications than our models. Beyond 384 cores, the Adams *et al.* model is faster than the Hybrid because the latter overestimates the number of computation steps, as explained in section 3.3.1. However, in the Hybrid-Async model, those constraints are removed and no explicit synchronization exists anymore. For this reason, the predicted performances from the Hybrid-Async simulator is higher that those of Hybrid and Adams *et al.* models. At 768 cores, the predicted performances from Hybrid-Async simulator is 1.07 times higher than Hybrid model, and 1.1 times than Adams *et al.* model. This is due to the fact that the asynchronous approach is more suitable to leverage more parallelism compared to Adams *et al.* and Hybrid models.

It is worth to note that the Hybrid model is simpler than Hybrid-Async simulator, mainly because it does not take into account asynchronous execution of tasks and evaluates the computation steps using a closed-form expression. Consequently, it is able to quickly predict the sweep computation according to parameters given as input. Opposingly, the Hybrid-Async simulator is more complex and gives better performances, at a cost of the run time of the simulation which is higher than that of Hybrid model (by a factor of ten in average).

We performed the same study on the `big` test case, which is based on a larger spatial mesh. We observe that the Hybrid model is able to hide almost all communications (Figure 3.9), and the predicted performances is slightly higher than that of the Adams *et al.* model (Figure 3.10), by a factor of 1.02 at 768 cores, even with the overestimation of the computation steps. The Hybrid-Async exploits better the available parallelism, and is able to achieve a very good scalability. The predicted performances from this model is higher than Hybrid and Adams *et al.* models, respectively by ×1.13 and ×1.16 at 768 cores.

3.5.2.3 Impact of the scheduling overhead in the Hybrid-Async model

Figure 3.11 presents a sensitivity study, of the Hybrid-Async model, to the scheduling overhead. This study is carried out with the `small` test case, using 64 nodes of the IVANOE platform. We fixed the task granularity and the process grid partitioning to the optimal values obtained from a run without any scheduling overhead (see Table 3.4). We made the following observations: the performance of the sweep operation decreases as the scheduling overhead increases, with a large drop of performance (from 2.1 Tflop/s to 1.6 Tflop/s) when using $\delta = 57.43\ \mu s$. This

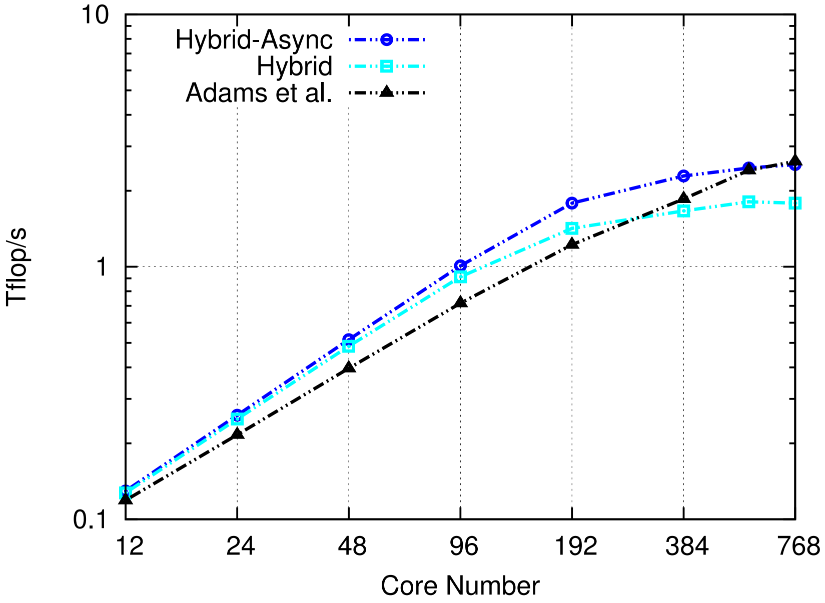


Figure 3.8: Comparison of Adams *et al.*, Hybrid and Hybrid-Async models using the `small` test case using parameters of the IVANOE platform.

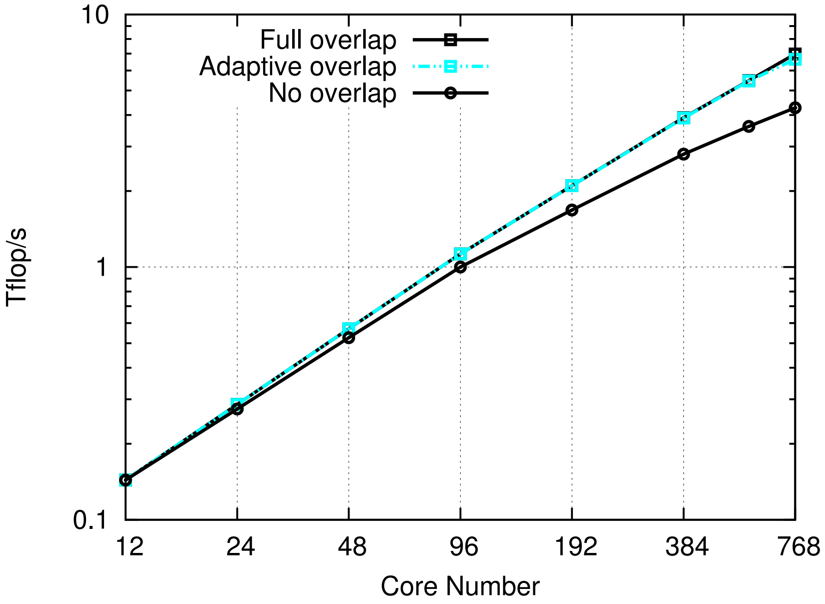


Figure 3.9: Performance of the Hybrid model using the `big` test case on IVANOE.

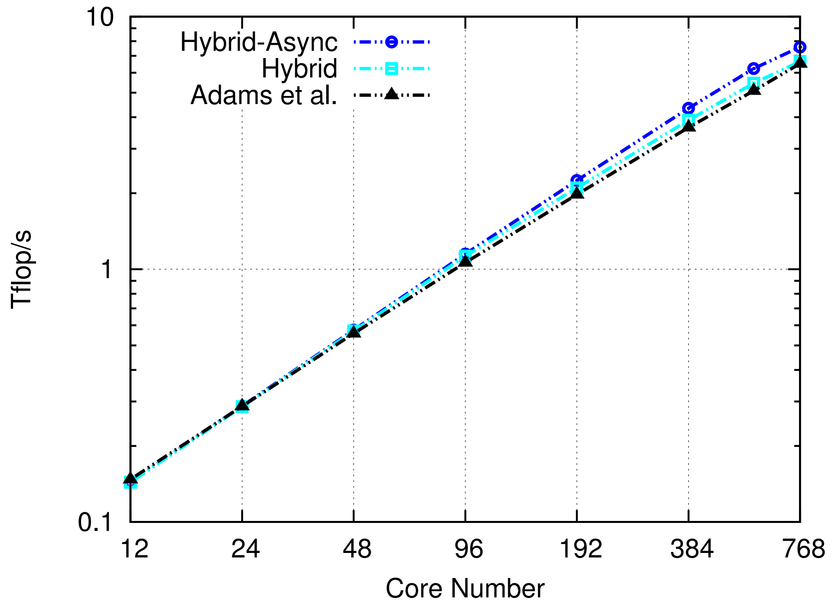


Figure 3.10: Comparison of Flat, Hybrid and Simulator using the `big` test case on IVANOE

value coincides with the computation time of a single task for the considered task granularity. This behavior is related to the fact that our simulation algorithm executes tasks by time spacing equal to T_{task}.

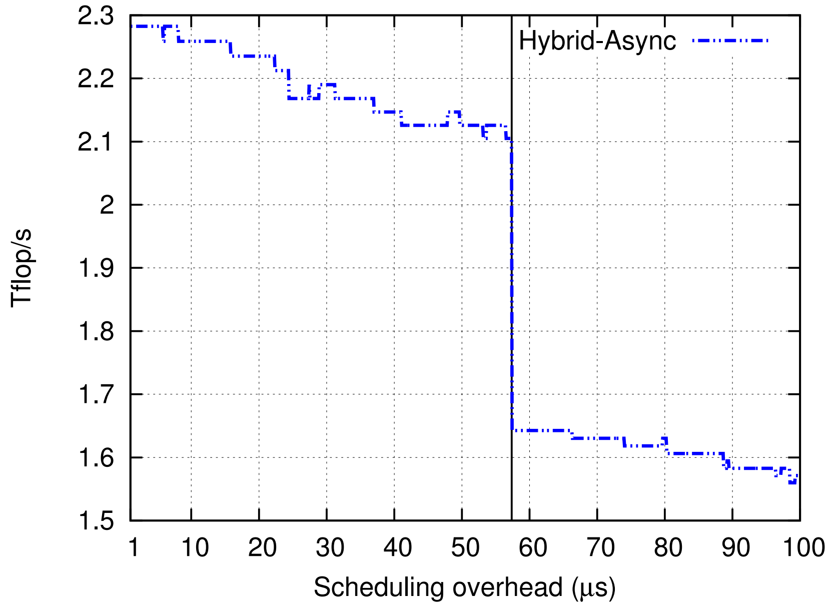


Figure 3.11: Impact of the scheduling overhead on the `small` test case, using 64 computing nodes. Process grid partitioning and `MacroCell` sizes are fixed to the optimal values obtained from the run without any overhead (see Table 3.4). The vertical line corresponds to the computation time of a single task.

▶ In this chapter, we have theoretically studied the performance of the sweep operation on multicore distributed systems, predicted by different classes of models. The classical *Flat* model, which assumes a parallel computer as a collection of independent computing cores, is widely used for designing parallel algorithms. We reviewed one such model, namely Adams *et al.* model, providing optimal results for parallel sweeps. Due to the shift on parallel computer architectures, moving from uniprocessor-based clusters to multi/many-core based clusters, traditional approaches are not anymore suited to leverage the full power of today computers. Given this situation, we developed two new performance models that take better into account architectures of hybrid machines: these are Hybrid and Hybrid-Async models. We found the predicted performances from the Hybrid model, acting as an implementation using explicitly a combination of message-passing and threading, is higher than that of the Adams *et al.* model when the considered test case provides enough parallelism to overlap communications by computations. The Hybrid-Async simulator, which simulates the execution of an asynchronous implementation of the sweep algorithm, removes all explicit synchronizations and thus is able to better exploit the available parallelism. This latter strategy increases the occupation of computational resources, and we found that the predicted performances from this simulator remains higher than predicted performances from Adams *et al.* and Hybrid models.

These results justify our strategy to design a *Hybrid* implementation of the sweep operation, targeting distributed multicore machines. In Chapter 4, we will present an implementation of the sweep operation, following the Hybrid-Async simulator, and using a generic task-based runtime system. We will see that the predicted performances with Hybrid-Async simulator is closer to the real performances as compared to the Hybrid model, but more costly. However, the latter model is nevertheless capable to quickly predict the optimal process grid partitioning, and thus it can be used to parametrize a sweep run in production contexts.

Chapter 4

A Massively Parallel Implementation of the Cartesian Transport Sweep

Contents

4.1 The emergence of generic task-based runtime systems 66
  4.1.1 The traditional MPI+X model 66
  4.1.2 Task-based models 66
4.2 Implementation of the sweep algorithm with INTEL TBB 67
4.3 Implementation of the sweep algorithm with PARSEC 68
  4.3.1 Task-graph of the sweep operation 68
  4.3.2 Data distribution 70
  4.3.3 Optimization of the scheduling through priorities 71
4.4 Implementation of the sweep algorithm with STARPU 71
4.5 Experiments 75
  4.5.1 Task-granularity selection and parameters of the performance models . 75
  4.5.2 Shared memory performances 76
  4.5.3 Distributed memory performances 78
    4.5.3.1 Optimal partitionning of the spatial domain 78
    4.5.3.2 Comparison of *Hybrid* and *Flat* approaches 79
    4.5.3.3 Hybrid-Async simulator using `small` test case 80
    4.5.3.4 Hybrid-Async simulator using `big` test case 81

To design an efficient massively parallel neutron transport solver, we considered the discrete ordinates method for solving the neutron transport equation (see Chapter 1). The vast majority of computations performed with this method are spent in the sweep operation. In Chapter 3, we have theoretically studied the performances of that operation on distributed multicore architectures, through the Hybrid model and the Hybrid-Async simulator. We established that the predicted performances of the Hybrid-Async simulator are higher than that by the classical *Flat* approaches such as the Adams *et al.* model. In this chapter, we will present a practical implementation of the Hybrid-Async model, using three different task-based runtime systems: PaRSEC [13], StarPU [5] and Intel TBB [99] frameworks.

4.1 The emergence of generic task-based runtime systems

As presented in section 1.4, modern parallel computers are built as an interconnection of several heterogeneous computing nodes, each of which comprises a number of computing devices such as multicore processors, accelerators such as Graphics Processing Units (GPUs) or manycore devices. Each of these devices must be addressed with a specific programming paradigm, or using extensions of computer programming languages. Consequently, such an approach implies mixing several programming paradigms together in the same application, through the well known MPI+X programming model.

4.1.1 The traditional MPI+X model

MPI+X is the most popular programming model for modern parallel computers. It consists in using the message passing standard MPI[1] to handle inter-nodes communications; while computing devices on each node are explicitly addressed via programming models specifically tuned for considered nodearchitectures. For instance, to write a computational kernel targeting a computing node equipped with traditional multicore processors, X can be either OpenMP [27], Intel TBB [99] or PThreads. On the other hand, if the computing node comprises GPUs, X can be NVIDIA CUDA [90] or OpenCL [110].

This MPI+X programming model generally follows a fork-join model, which is no longer relevant for today's parallel computers. Indeed, with this model, computations and communications phases are usually serialized, and thus global synchronizations are implicitly introduced in the execution flow. Thereby, the available parallelism inside an algorithm is underutilized, and computational resources are not efficiently exploited. Indeed, some workers go to idle state while they could be making progress on some other work (e.g. for MPI+OpenMP: master thread performs communications while other threads are waiting). These issues motivated the emergence of task-based models on top of generic runtime systems.

4.1.2 Task-based models

Generic concepts of task-based runtime systems

Regarding modern heterogeneous clusters, many initiatives have emerged in previous years to develop efficient runtime systems. Most of these runtime systems use a task-based paradigm to express concurrency and dependencies by employing a task dependency graph to fully represent the application to be executed. This graph is a directed acyclic graph (DAG) where nodes are computational tasks and edges represent data flows and dependencies. The major interest of this approach is the separation of major concerns arising when designing parallel programs:

---

[1] http://www.mpi-forum.org/

- Description of the algorithm
- Writing optimized computational kernels
- Efficient scheduling of the tasks over the underlying hardware

These task-based models offer an elegant and efficient way to express parallelism inside an algorithm, while avoiding the cumbersome hand-coding of all the communication primitives needed to realize data transfers. Such models allow the removal of all unnecessary and global synchronizations between tasks; hence exposing a larger number of parallel tasks.

State-of-the-art on task-based models

The past decade has witnessed the development of several approaches of task-based models on top of generic runtime systems. The main differences between these approaches are related to their representation of the graph of tasks, whether they manage data movements between computational resources, the extent to which they focus on task scheduling, and their capabilities of handling distributed and heterogeneous systems. Runtime systems such as QUARK [120], STARPU [5], or STARSS [7] propose an insert task paradigm where a sequential code submits all computational tasks. In this case, the dependency graph is dynamically discovered at runtime according to how the data is used, which is indicated through keywords such as INPUT, OUTPUT or INOUT. INTEL TBB is another framework which also allows the use of a task-based approach. Within this framework, the task dependency graph is constructed according to high level programming paradigms, or algorithms such as `parallel_for`, `parallel_do` or `parallel_reduce`. Another framework is CHARM++ [61] which is a parallel variant of the C++ language, and allows the writing of programs where a flow of tasks is applied to each piece of data. In addition, it provides sophisticated load balancing and a large number of communication optimization mechanisms. INTEL CNC [16] and PARSEC [13] construct an abridged representation of the DAG (with its tasks and their dependencies) with a structure agnostic to algorithmic subtleties, where all intrinsic knowledge about the complexity of the underlying algorithm is extricated, and the only constraints remaining are annotated dependencies between tasks [22]. However, as noted in [112], the INTEL CNC framework which is built on top of INTEL TBB incurs some overhead compared to INTEL TBB.

For designing and implementing our sweep operation on top of generic task-based runtime systems, we compare the following approaches: hand-written (INTEL TBB), insert task (STARPU), and parametrized DAG (PARSEC). In the following, we will present the implementation of the sweep on top these frameworks. For the sake of clarity, the following presentation will be given in a 2D case, and we consider the case of vacuum boundary conditions which enables a concurrent sweep of all the quadrants.

4.2 Implementation of the sweep algorithm with INTEL TBB

To implement the sweep operation with INTEL TBB, we rely on the `parallel_do` primitive [102], that enables a dynamic scheduling of the parallel tasks. This parallel function allows a pool of threads to execute the tasks from a list, which is dynamically updated at runtime. At the beginning of the sweep, the task list is composed of four tasks, one for each quadrant:

$$\{\mathcal{T}^{q,0,0},\ q = 0,1,2,3\}.$$

Some of the running threads process these tasks and update the task list to:

$$\{\mathcal{T}^{q,1,0},\ q = 0,1,2,3\} \cup \{\mathcal{T}^{q,0,1},\ q = 0,1,2,3\}$$

and so on. The INTEL TBB `parallel_do` primitive we used, dynamically explores, executes and updates the list of ready tasks. The computations performed inside a task consist in updating the outgoing angular fluxes in one quadrant (or octant in 3D), for a single `MacroCell`. These computations are vectorized over angular directions thanks to the generic `C++` library Eigen [47] and was presented in Chapter 2.

Note 2. *It is worth noting that each `MacroCell` object encapsulates a scalar flux field, common to all quadrants. Therefore, for a concurrent sweep of all quadrants, each `MacroCell` must possess a `mutex`, to prevent two threads from processing the same data simultaneously. When a thread starts processing a given task $\mathcal{T}^{q,a,b}$, it acquires the `mutex` corresponding to the `MacroCell`(a, b). When that processing is finished, the `mutex` is released, so that another thread can start working on that `MacroCell`.*

4.3 Implementation of the sweep algorithm with PARSEC

PARSEC is a framework intended to develop parallel applications on distributed heterogeneous architectures. It features a generic data-flow runtime system, supporting a task-based implementation and targeting distributed hybrid systems. This framework relies on the dynamic scheduling of a directed acyclic graph of the considered algorithm, the nodes represent which computational kernels (tasks), and edges represent data transfers between tasks. Thanks to an algebraic description of the task dependencies, the scheduling is completely asynchronous and fully distributed. Moreover, it takes into account user defined priorities and overlaps communications by computations.

As mentioned above, the programming model exploited in PARSEC enables the separation of the major concerns in distributed computing: the kernels, the algorithm, and the data distribution. Here, the computational kernel is the same as for the INTEL TBB implementation: update of angular fluxes in one quadrant (or octant in 3D) for a single `MacroCell`.

4.3.1 Task-graph of the sweep operation

To use the PARSEC framework, the algorithm must be described as a DAG using the symbolic representation specific to PARSEC in a Job Data Flow (JDF) file. In this file, all tasks of the algorithm are defined by their execution space; their data placement or affinity; their input, output or in-out data-flows; and body. Each data-flow has incoming and outgoing edges connecting them to other tasks of DAG, or directly to memory accesses. The body specifies the computation of the task. This file is later compiled into a C code with a set of functions that will submit the tasks to runtime system, check the dependencies, release the data to the following tasks, and execute the body. The Cartesian sweep operation, either 2D or 3D, by its geometric structure, is a natural and simple candidate for this formalism. A simplified version of the 2D sweep operation, without boundary cases and one single quadrant, is given in the Listing 4.1.

Listing 4.1: JDF file of the 2D sweep for one quadrant

```
ComputePhi(a, b)
/* Execution space */
a = 1 .. ncx
b = 1 .. ncy

/* Parallel partitioning */
: mcg(a, b)

/* Parameters */
RW PSIX    <- (a != 1)   ? PSIX ComputePhi(a-1, b) : psi_x(b)
           -> (a != ncx) ? PSIX ComputePhi(a+1, b) : psi_x(b)
RW PSIY    <- (b != 1)   ? PSIY ComputePhi(a, b-1) : psi_y(a)
           -> (b != ncy) ? PSIY ComputePhi(a, b+1) : psi_y(a)

RW MCG     <- mcg(a, b)
           -> mcg(a, b)

/* Priority of this task */
; priority(a, b)

BODY
{
  computePhi_CPU ( MCG, PSIX, PSIY );
}
END
```

Only one task *ComputePhi*, described by its position on the grid (a, b), composes the algorithm. The execution space specifies that there are as many tasks as cells on the grid of size `ncx` × `ncy`. The parallel partitioning argument instructs the runtime to run the task *ComputePhi(a,b)* on the node where the data `mcg`(a, b) is located (see section 4.3.2). `mcg` is the structure describing the data distribution, and it is called data descriptor in the PaRSEC taxonomy. Thus, `mcg`(a, b) represents here the data structure of the `MacroCell` of coordinates (a, b). Each task *ComputePhi* has three in-out data dependencies:

- `PSIX` and `PSIY` are aliases that correspond to the neutron current associated to one `MacroCell`, respectively along the x and y dimensions. These two variables are labeled as read/write (`RW`) to indicate that the neutron current on the incoming faces are overwritten by those on the outgoing faces, that is the neutron currents are not stored. `PSIX` is the current along the x dimension. It comes from the previous task on this dimension $ComputePhi(a - 1, b)$, and is forwarded to the next task $ComputePhi(a + 1, b)$. `PSIY` is the current on the second dimension, and its flow is similar to that of `PSIX`. On the initial border of the domain, they are directly read from the main memory, and on the final border of the domain, they are written to the initial storage space. In the case of non reflecting nor periodic boundaries, they are directly initialized through a set of initial tasks at the beginning, and destroyed at the end.

- `MCG` is the alias on the address of the `MacroCell` object associated to the considered task. It is directly read from the main memory using the data descriptor `mcg`. This alias is used to discover some physical data encapsulated inside the `MacroCell` structure and needed to execute the computational kernel.

The priority line allows the developer to provide a hint to the scheduler helping it to prioritize the most important tasks. We use this feature to optimize the scheduling of the sweep operation (see section 4.3.3). Finally, The `BODY` section contains the computational task itself exploiting the parameters and flow aliases of the task to access the data.

4.3.2 Data distribution

Once the algorithm has been described in the PaRSEC language, the runtime needs to know how the data is distributed. A simple API must be implemented to provide this information to the scheduler. This is shown in Listing 4.2 for the case of a 2D block distribution of `ncx` × `ncy` spatial grid over a $P \times Q$ grid of processes, as shown in Figure 3.1a. The `rank_of()` function is used by the scheduler to know to which process data the belongs to, but also, in our case, for task mapping over the node as previously shown. Two tasks on different nodes can then be detected by the locality of the data they use, and communications are automatically generated: through direct accesses in the case of shared memory, or MPI asynchronous communications on distributed memory. This separation between algorithm and data, specific to PaRSEC, allows a quick evaluation of several data distributions for the same algorithm. Note that it is also possible to have data distribution depending upon subgroup of threads, useful for minimizing the data traffic on NUMA architectures. But for the experiments that will follow, we did not use this feature[1]. The `data_of()` function returns the pointer to the `MacroCell` object when this one is local. Addresses of transferred objects are internally handled by the runtime.

---

[1] In the future, we will add another level of parallelism by the mean virtual processes to take advantage of "highly" NUMA nodes.

Listing 4.2: Blocked data distribution of the `MacroCell` grid over the $P \times Q$ grid of processes.

```
// Rank of the process owning MacroCell(a,b)
int rank_of ( Parameters & param, int a, int b ) {
  int lp = a / (param.ncx / param.P);
  int lq = b / (param.ncy / param.Q);

  return lq * param.P + lp;
}

// Address of the object MacroCell(a,b)
void * data_of ( Parameters & param, int a, int b,
                 DataType & mcgData ) {

  int aa = a % (param.ncx / param.P);
  int bb = b % (param.ncy / param.Q);

  return mcgData[bb][aa];
}
```

4.3.3 Optimization of the scheduling through priorities

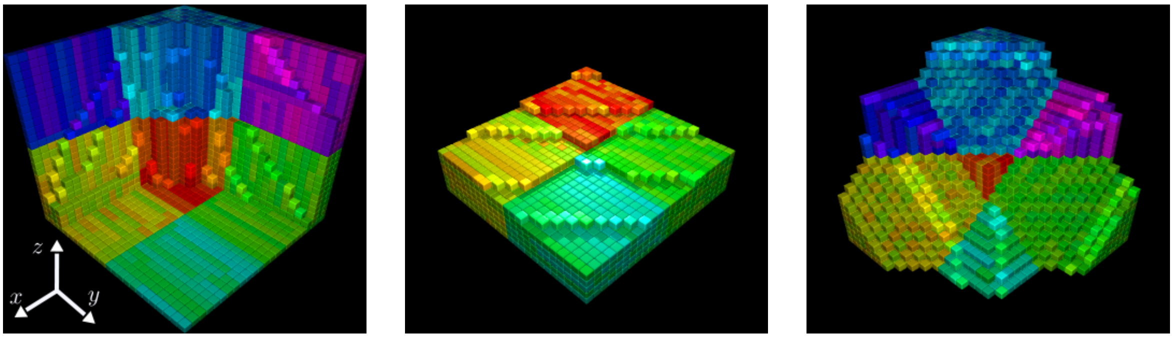
The PaRSEC runtime implements several schedulers, which may impact the order of task processing and the performance. The default scheduler, Local Flat Queue (`LFQ`), favors the memory affinity by maximizing memory reuse, by following the dependencies of tasks as defined in the JDF file (Listing 4.1). Tasks released by a dependency are added to the local queue of the working thread. In the case of the 3D sweep, it leads to a prioritization of the sweep per columns of cells (Figure 4.1a). Such a scheduling is not an efficient policy for the sweep since it does not try to maximize the wave front, and thus the number of parallel tasks available. To tackle this problem, we considered the Priority Based Queue (`PBQ`) scheduler. This scheduler, similar to `LFQ`, adds ordering of the tasks in the local queue based on optional user defined priorities. We have studied two different priorities: `PlaneZ` and `Front`. The first favors tasks belonging to the same z-plane (Figure 4.1b), while the latter favors a progression by front (Figure 4.1c). The first one gives good results when the number of process over z, R, is equal to 2, because it reduces the idle time of processors in the same (x, y) plane; while the second is more generic as it frees tasks quickly along the 3 dimensions.

4.4 Implementation of the sweep algorithm with StarPU

Basic concepts

There are two different ways to implement an algorithm on top of StarPU: either using C extensions (`pragmas`) to annotate a sequential code, or directly using the StarPU's API. In the following, we will focus on the latter. Thus, the implementation of an algorithm with StarPU is relatively straightforward and consists of the following two steps:

1. Create a *codelet* for each computational kernel in the considered algorithm. A *codelet* is a structure that holds various implementations of the kernel that it represents, and buffers manipulated by that kernel.



(a) `LFQ` (b) `PBQ` – `PlaneZ` (c) `PBQ` – `Front`

Figure 4.1: Animation snapshots showing the behavior of various scheduling strategies on the sweep progress throughout the spatial domain for one single direction. Data is distributed over a $2 \times 2 \times 2$ process grid. Threads of similar colors belong to the same node. Brightest colors are used to show what `MacroCells` are processed during the snapshots.

2. Create and submit sequentially all tasks to the runtime system; the runtime can then infer the whole task-graph corresponding to the algorithm. Each task is an association of a *codelet* and a set of parameters manipulated by the kernel corresponding to that task.

For implementing the sweep operation with StarPU, we consider a single *codelet*, which corresponds to the update of the angular fluxes on a single `MacroCell`, for one octant. The `MacroCell` object associated with the *codelet*, and the process to which it belongs, is discovered using the same data descriptor we used in the PaRSEC implementation: `mcg`. In 2D, this *codelet* defines two read/write buffers, each of which being attached to a single `MacroCell` face. These are similar to the `PSIX` and `PSIY` aliases we used in the PaRSEC implementation (see Listing 4.1). Each buffer is registered to the StarPU runtime with a *data handle*, which is an opaque pointer that designates an array. As in the PaRSEC implementation, we declare that each of these buffers is allocated on-the-fly at runtime. This *codelet* defines one computational kernel, which is the same as in the PaRSEC implementation.

Submission of tasks

Let us consider the basic implementation of the sweep operation with StarPU, in 3D, as described in Listing 4.3. This implementation submits all tasks of the first octant, before submitting those of the next one, and so on. Moreover, the submission of tasks for a single octant favors the tasks in the same column (z dimension). However, this implementation has two issues. Firstly, if the number of tasks per octant is large and exceeds the size of the window of visible tasks defined by StarPU, then the octants are going to be processed sequentially, because the StarPU runtime may not be able to visualize all the available parallelism. Thereby, the discovery of the available parallelism is limited, and the computational resources are underexploited. Secondly, due to the fact that StarPU considers the tasks according to their submission order, the considered order of tasks submission (column-wise) does not guarantee that the execution will maximize the front exploration which is the optimal evolution of the sweep as mentioned in section 4.3.3.

Thus to enforce the discovery of the available parallelism and to enforce the front exploration, it is essential to submit (or prioritize) the execution of tasks by following the sweep front. To

Listing 4.3: Basic submission of the sweep task-graph with StarPU.

```
// An octant o is defined by (fx,fy,fz) such that o=fx+2*fy+4*fz
for(fx=0; fx<2; fx++) {

  int xinc = 1 - 2*fx;
  int xbeg = fx * (ncx-1);
  int xend = ((1+fx)%2)*ncx+fx*xinc;

  for(fy=0; fy<2; fy++) {

    int yinc = 1 - 2*fy;
    int ybeg = fy * (ncy-1);
    int yend = ((1+fy)%2)*ncy+fy*yinc;

    for(fz=0; fz<2; fz++) {

      int zinc = 1 - 2*fz;
      int zbeg = fz * (ncz-1);
      int zend = ((1+fz)%2)*ncz+fz*zinc;

      for(x=xbeg; x!=xend; x+=xinc) {
        for(y=ybeg; y!=yend; y+=yinc) {
          for(z=zbeg; z!=zend; z+=zinc) {

            void *data_mcg=NULL;

            if ( rank==mcg->rank_of ( mcg, x, y, z ) ){
              data_mcg = mcg->data_of ( mcg, x, y, z );
            }

            starpu_task_insert
            ( cl_solve,
              STARPU_VALUE, &data_mcg, sizeof(void*),
              STARPU_VALUE, &CS,       sizeof(void*),
              STARPU_VALUE, &fx,       sizeof(int),
              STARPU_VALUE, &fy,       sizeof(int),
              STARPU_VALUE, &fz,       sizeof(int),
              STARPU_VALUE, &lnx,      sizeof(int),
              STARPU_VALUE, &lny,      sizeof(int),
              STARPU_VALUE, &lnz,      sizeof(int),
              STARPU_RW, mcg->getPsi( mcg, 0, fx, fy, fz, 0, y, z ),
              STARPU_RW, mcg->getPsi( mcg, 1, fx, fy, fz, x, 0, z ),
              STARPU_RW, mcg->getPsi( mcg, 2, fx, fy, fz, x, y, 0 ),
              STARPU_EXECUTE_ON_NODE, mcg->rank_of( mcg, x, y, z ),
              0 );

          }
        }
      }
    }
  }
}
```

this end, we loop over all fronts and we successively submit tasks belonging to each front for all octants. The Listing 4.4 illustrates this strategy.

Listing 4.4: Front-like submission of the sweep task-graph with StarPU.

```
const int nfront=ncx+ncy+ncz-2;

// An octant o is defined by (fx,fy,fz) such that o=fx+2*fy+4*fz
for (int f=0; f<nfront; ++f){

  for (int k=0; k<min(ncz,f+1); ++k){
    for (int j=max(0,f-ncz); j<min(ncy,f+1-k); ++j){
      const int i=f-j-k;

      // Loop over octants
      for (int fx=0; fx<2; fx++){
        for (int fy=0; fy<2; fy++){
          for (int fz=0; fz<2; fz++){

            void *data_mcg=NULL;

            if ( rank==mcg->rank_of ( mcg, x, y, z ) ){
              data_mcg = mcg->data_of ( mcg, x, y, z );
            }

            /* Submit the task $\mathcal{T}^{o,i,j,k}$ */
            starpu_task_insert
            ( cl_solve,
              STARPU_VALUE, &data_mcg, sizeof(void*),
              STARPU_VALUE, &CS,       sizeof(void*),
              STARPU_VALUE, &fx,       sizeof(int),
              STARPU_VALUE, &fy,       sizeof(int),
              STARPU_VALUE, &fz,       sizeof(int),
              STARPU_VALUE, &lnx,      sizeof(int),
              STARPU_VALUE, &lny,      sizeof(int),
              STARPU_VALUE, &lnz,      sizeof(int),
              STARPU_RW, mcg->getPsi( mcg, 0, fx, fy, fz, 0, y, z ),
              STARPU_RW, mcg->getPsi( mcg, 1, fx, fy, fz, x, 0, z ),
              STARPU_RW, mcg->getPsi( mcg, 2, fx, fy, fz, x, y, 0 ),
              STARPU_EXECUTE_ON_NODE, mcg->rank_of( mcg, x, y, z ),
              0 );

          }
        }
      }

    } // j
  } // k

} // f
```

4.5 Experiments

This section presents the performances of the PaRSEC, StarPU and Intel TBB implementations of the sweep operation, on shared and distributed multicore systems. Performance measurements were carried out in single precision, on the ivanoe and athos clusters (see Table A.1), using the 3D test cases `small` and `big` (see Table 3.2). All the experiments were performed with a concurrent sweep over all octants.

To evaluate the efficiency of the task-based implementations of the sweep operation, we will rely on the theoretical performance models presented in Chapter 3. From this perspective, it is necessary to first calibrate those models for the target architectures.

4.5.1 Task-granularity selection and parameters of the performance models

Task-granularity

To determine the best task-granularity using the Hybrid model or the Hybrid-Async simulator, it is necessary to have a good estimation of the scheduling overhead associated with the runtime in use. Moreover, it is also required to model the cache effects of the target processors, because the performance of the sweep kernel depends on whether or not the `MacroCells` fit into the processor cache memory. Such a strategy will introduce some low level details into the performance models. Recalling that our goal is to design a simple performance model capable of returning optimal domain partitioning, as a function of the problem size and target machine parameters, we choose to rather adopt an experimental process for evaluating the optimal task-granularity.

Figure 4.2 presents the single-core performances per `MacroCell`, of the PaRSEC, StarPU and Intel TBB implementations of the sweep operation. There are 5 `MacroCells` along each dimension, whose size varies from $5 \times 5 \times 5$ to $100 \times 100 \times 100$ (we consider a cubic `MacroCell`). This experiment was conducted on the athos machine. We observe that for a `MacroCell` of size $5 \times 5 \times 5$, the performance of the Intel TBB implementation is better than that of PaRSEC and StarPU, and reaches a plateau of 16.8 Gflop/s, starting from `MacroCell` of size $10 \times 10 \times 10$. However, the performance of the PaRSEC implementation increases quickly and stabilizes at 18.1 Gflop/s starting from a `MacroCell` of size $20 \times 20 \times 20$. Surprisingly, the performance of the StarPU implementation is lower than that of PaRSEC, up to `MacroCell` of size $50 \times 50 \times 50$. As the computational kernel used in both implementations is the same, this result suggests that the scheduling overhead per task for StarPU and Intel TBB is larger than that of PaRSEC, and thus StarPU is suited to schedule coarser tasks. In the following of this study, for all the three implementations, we will use `MacroCells` of size $20 \times 20 \times 20$ for the `big` test case, and $10 \times 10 \times 10$ for the `small` test case.

Parameters of the performance models

On a given machine, we recall that the input parameters to be used for the performance models are the following:

- The average computation time per spatial cell, per angular direction and per energy group, obtained from a sequential run of the sweep on the target machine: T_{grind}
- The scheduling overhead per task, associated with the considered runtime: δ
- The network performance of the target machine and the communication overhead (o_c)

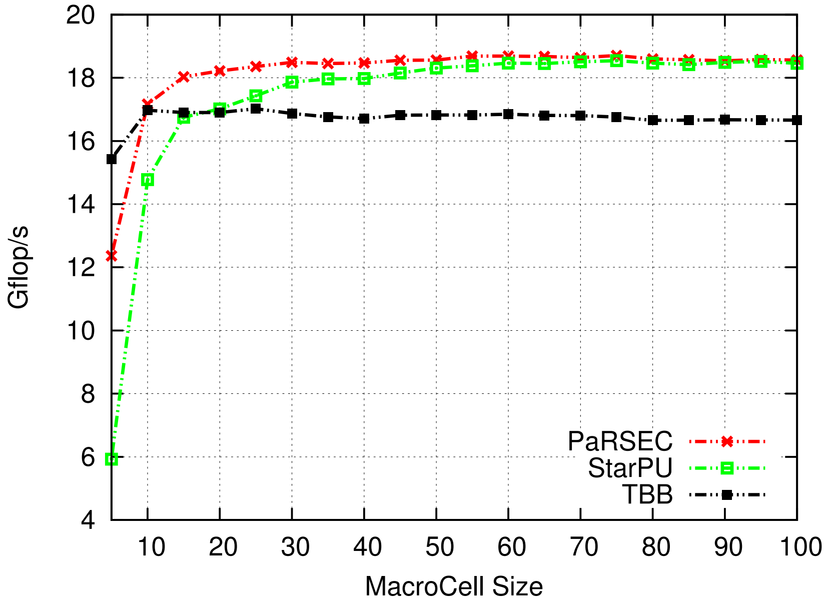


Figure 4.2: Single-core performances of the PaRSEC, StarPU and Intel TBB implementations of the sweep operation, averaged by the number of `MacroCells`: 25 (there are five `MacroCells` along each dimension). This experiment was conducted on a computing node of the athos platform (Intel Xeon E5-2600 V2 processors). The angular discretization is a S_{16} Level Symmetric quadrature (288 directions).

The evaluation of T_{grind} has been presented in section 3.5.1. Here, we recall its value for the ivanoe machine and we give its value for the athos machine (see Table 4.1). The scheduling and communication overheads are obtained from a fit of the Hybrid-Async model to real measurements. The network performance is given by a NetPIPE benchmark on the target computer. Figure 4.3 presents the network latency of the ivanoe and athos computers.

		`small`	`big`
N_{dir}		288	288
Discretizations	$N_u \vert u \in (x, y, z)$	120	480
`MacroCell` size		$10 \times 10 \times 10$	$20 \times 20 \times 20$
T_{grind} (ns)	IVANOE	2.12	1.92
	ATHOS	2.28	1.57
GFlops		12.44	796.26

Table 4.1: Characteristics of the test cases used. GFlops is the number of floating point operations required for a single complete sweep. We count 25 floating point operations per spatial cell per angular direction (see section 1.3.3).

4.5.2 Shared memory performances

We have conducted a comparative study of the performances obtained with the three implementations of the sweep, using respectively Intel TBB, PaRSEC and StarPU. This study

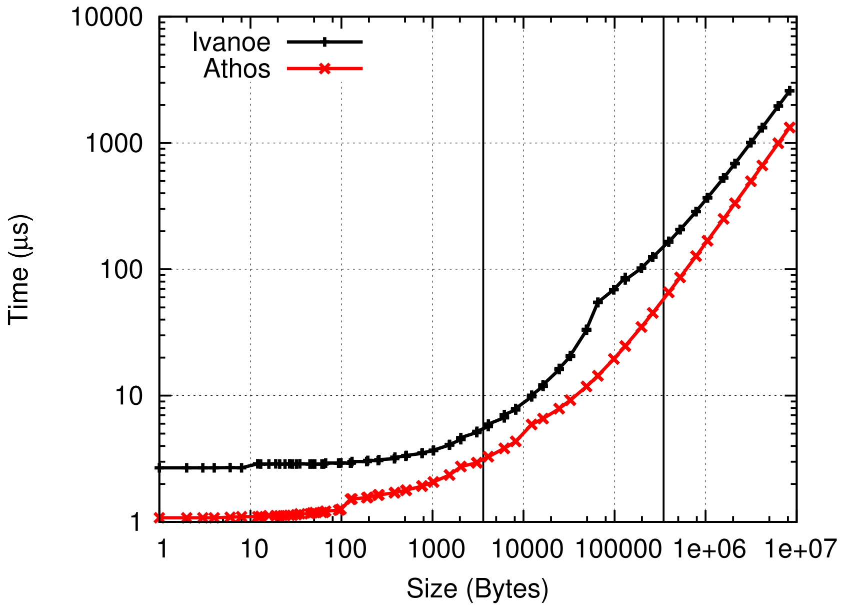


Figure 4.3: Performance of the network interconnect of the IVANOE and ATHOS platforms. Measurements are obtained using NetPIPE utility, compiled with OpenMPI 1.6.5. The two vertical lines define the range to which messages sent through the network belong to, for `small` and `big` test cases, according to task granularity.

was realized using a single computing node of the ATHOS machine, and the result is presented on Figure 4.4. The following comparison will be on 23 cores. Indeed, StarPU allocates one thread per process for its internal runtime management which cause a performance drop when using all the 24 cores of the computing node.

The three implementations of the sweep were run with the same task-granularity: $20 \times 20 \times 20$ cells per `MacroCell`. We evaluated the Hybrid-Async simulator, without any scheduling overhead, and using T_{grind} obtained from a sequential run of the PaRSEC implementation. The maximum performance predicted by the Hybrid-Async simulator is 404.2 Gflop/s at 23 cores. This performance gives an upper-bound on the performance that can be obtained from an implementation of the sweep operation, on the considered computing node. All three implementations of the sweep achieve nearly the same performances for a run on a single core. This is unsurprising because only a single task is processed; and consequently neither the scheduling policy nor the data locality can impact the performances. However, we found that the performance of the PaRSEC implementation of the sweep, for a run using 23 cores, reaches 367.6 Gflop/s. This corresponds to 90.8% of the predicted performance by the Hybrid-Async simulator, and to 35.45% of the theoretical peak performance of the considered computing node. Moreover, the parallel efficiency of the PaRSEC implementation is 90.9% using 23 cores, and it is respectively 6.76% and 13.06% faster than the implementations with Intel TBB and StarPU. The parallel pattern being constant, we interpret this improved speed-up as a sign for a reduced scheduling overhead for the PaRSEC framework. Indeed, with the PaRSEC framework, the graph of task is not unfolded in memory, thanks to the parametric representation of the DAG: only ready tasks to be executed exist in the system (see [13]). In addition, the PaRSEC implementation does not have any additional cost associated with insertion of tasks as in StarPU.

The results presented in this section show that the PaRSEC runtime is able to give good

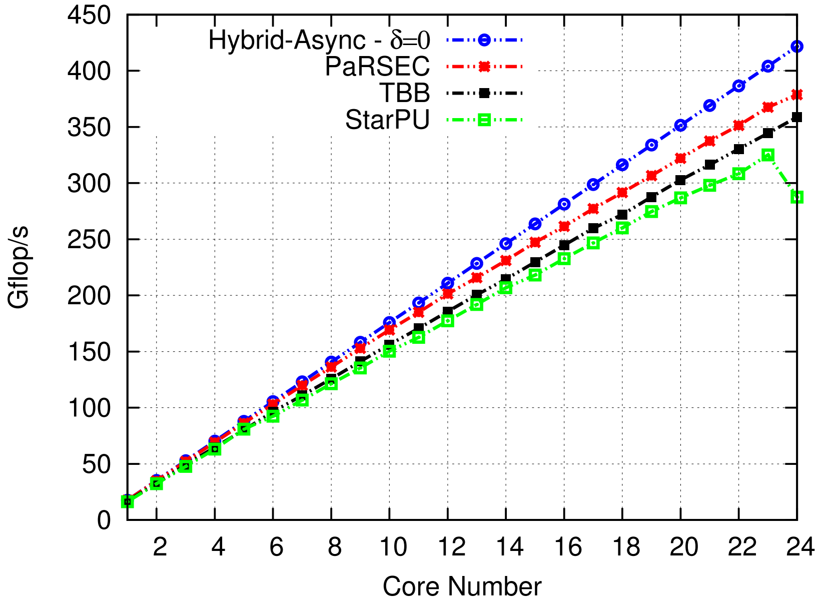


Figure 4.4: Performances of the implementations of the sweep operation using `big` test case (see Table 3.2), on a single 24-cores computing node of the ATHOS supercomputer. All measurements were obtained using the same task-granularity of 20 × 20 × 20 cells per task.

performances per core, even when the task granularity is small, as compared to StarPU. A small `MacroCell` size allows to extract a larger number of tasks from the sweep operation, which is a necessary condition for maintaining a good strong-scalability on large number of cores. For this reason, we are going to focus on the PaRSEC framework for distributed memory performance studies.

4.5.3 Distributed memory performances

On a distributed memory machine, the performance of the sweep operation depends strongly on the domain partitioning. Using Hybrid and Hybrid-Async performance models, we want to determine the optimal process grid partitioning, according to the problem size and the characteristics of the target computer. In this section, we present the performances of the PaRSEC implementation of the sweep operation on distributed multicore architectures, using optimal process grid partitioning.

4.5.3.1 Optimal partitionning of the spatial domain

For each number of cores, we use the Hybrid and Hybrid-Async performance models to give us the optimal data distribution (P, Q, R) that minimizes the sweep execution time. Figure 4.5 compares the experimental results against the predicted results from Hybrid model and Hybrid-Async simulator, running the `big` test case with `MacroCells` of size 20 × 20 × 20, on 48 nodes of the IVANOE platform. We first set the communication overhead to $o_c = 0$ and the scheduling overhead to $\delta = 0$. We found that the predicted performances by the Hybrid-Async follows the same trend as actual measurements from PaRSEC, except for the partitioning $(24, 2, 1)$. Moreover, the PaRSEC implementation and Hybrid-Async simulator gave the same optimal partitioning: $(12, 2, 2)$. Then, we fitted the Hybrid-Async data to the experimental measure-

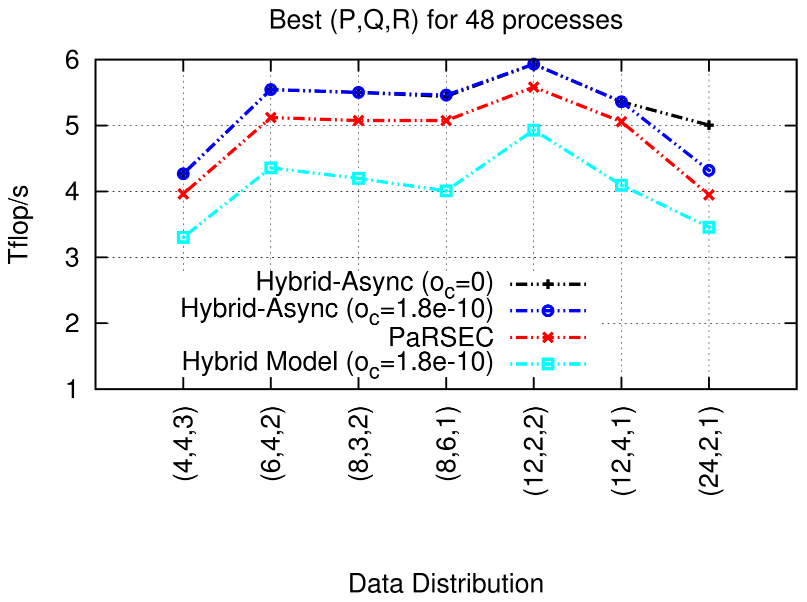


Figure 4.5: Sensitivity of Hybrid and Hybrid-Async performance models to data distributions using 48 nodes (576 cores) of the IVANOE cluster, and the `big` test case. Predicted performances by the Hybrid model and Hybrid-Async simulator are obtained using a scheduling overhead value of $\delta = 0$ s.

ments and we found that $o_c = 1.8 \cdot 10^{-10}$ s/Byte is the value of the communication overhead that minimizes the discrepancy between the Hybrid-Async data and actual measurements. Using this value of o_c, the predicted performances complies with actual measurements for each of the partitioning. However, the predicted performances by the Hybrid model, while following the same trend as actual measurements, remain lower than actual measurements. This is unsurprising because, this Hybrid model overestimates the computation step number as explained in section 3.3.1.

The experiment we conducted here has shown that both Hybrid model and Hybrid-Async simulator are able to give optimal data distribution. Thus, all the performance measurements that we will present in the following, are obtained using optimal domain partitioning.

4.5.3.2 Comparison of *Hybrid* and *Flat* approaches

In order to compare *Hybrid* and *Flat* approaches, we would have needed a hand-written MPI implementation of the sweep. Nonetheless, we can mimic its behavior with the PaRSEC implementation. To achieve this, we used as many processes as available cores to perform the experiment; each process being bound to one core. In reality, the PaRSEC framework runs an extra thread per process to manage the communications. Thus, for preventing the communication thread from disrupting the computation progress, we dedicated two cores per process: one for the computation, and one for the communication. Therefore, to run the *Flat* experiments with PaRSEC, the number of processes started on a node is equal to half the number of cores on the node. To ensure a fair comparison of *Hybrid* and *Flat* approaches, *Hybrid* performance measurements were performed by launching one process per node and a number of computational threads equal to half the number of cores. Hence, with both models, the same amount of data is going through the interconnection network.

We performed this experiment on the IVANOE machine, and the result is presented on Figure 4.6. In this particular case, 6 computation cores are being used on each node. Performance measurements using two nodes, that is 12 cores, show that the *Hybrid* implementation is 30% times faster than the *Flat* one. To justify this discrepancy, first it must be noted that although intra-node MPI communications can be done via shared memory, it remains that they conduct to a poor usage of caches and saturation of the memory buses. In contrast, these effects are much reduced when using threads in shared memory, and this may justify the observed difference. At 384 cores, the *Hybrid* implementation is 1.9 times faster than the *Flat* one. These results confirm that the *Hybrid* approach is more efficient than the *Flat* one.

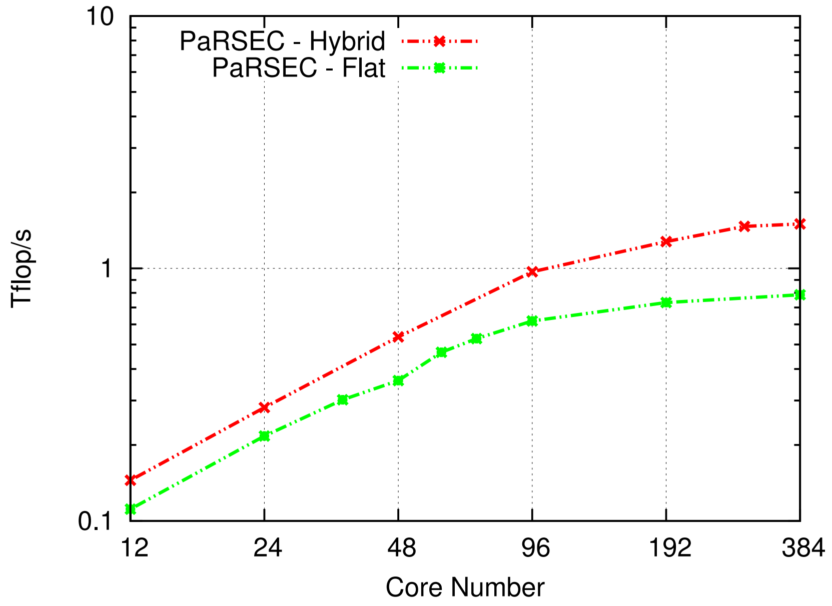


Figure 4.6: Comparison of *Hybrid* and *Flat* approaches using `small` test case on the IVANOE cluster. We used 64 nodes and 6 computations cores per node. The performance measurements were obtained using optimal partitioning, according to predictions of Hybrid and Adams *et al.* models.

4.5.3.3 Hybrid-Async simulator using `small` test case

Figure 4.7 shows the experimental results of the PaRSEC implementation of the sweep, compared to the predicted performance from the Hybrid model and the Hybrid-Async simulator. This experiment was conducted by running the `small` test case on the IVANOE platform. By setting $\delta = 0$ and $o_c = 0$, we observe that up to 96 cores (8 nodes partitioned into $(2, 2, 2)$), the predicted performance from Hybrid-Async simulator complies well with the actual measurements from the PaRSEC implementation. For this number of cores, the Hybrid-Async simulator predicts a performance of 1.01 Tflop/s, while the PaRSEC implementation achieves 0.84 Tflop/s and corresponds to a parallel efficiency of 81.91%. At 768 cores the PaRSEC performances is 1.56 Tflop/s and the corresponding parallel efficiency drops to 18.85%. Moreover, the performance predicted by the Hybrid-Async simulator, for 768 cores, is 2.54 Tflop/s corresponding to a theoretical parallel efficiency of 30.68%, which is 1.6 times more efficient than actual measurements. However, one should be aware that when using 768 cores with the `small` test case and `MacroCells` of size $10 \times 10 \times 10$, the average number of tasks per core is 18; and thereby

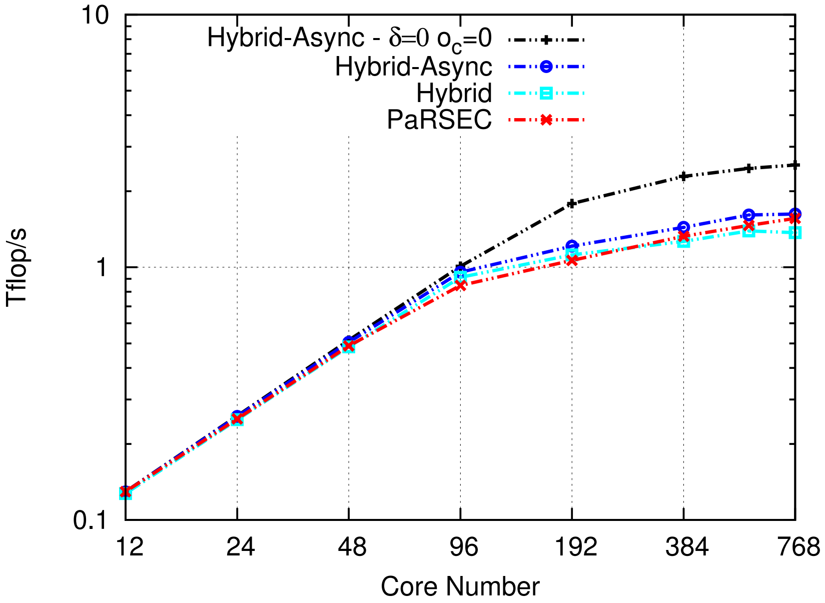


Figure 4.7: Performance comparison of the PaRSEC implementation of the sweep against predicted performances by Hybrid model and Hybrid-Async simulator. This experiment was conducted on the IVANOE cluster, using the `small` test case. Using $\delta = 8.1 \cdot 10^{-6}$ s, $o_c = 4.2 \cdot 10^{-10}$ s/Byte, the prediction of the Hybrid-Async complies with actual measurements.

the runtime scheduling overhead is not negligible. To assess this hypothesis, we performed a fit of the experimental data to the simulator data. We found that using $\delta = 8.1 \cdot 10^{-6}$ s and $o_c = 4.2 \cdot 10^{-10}$ s/Byte, the predicted performances agrees well with actual measurements. Note that this value of the communication overhead is different from that which was obtained in the `big` test case ($o_c = 1.8 \cdot 10^{-10}$ s/Byte).

We used this value of o_c to run the Hybrid model. We observe that the optimal partitioning is, each time, the same as that of Hybrid-Async, but the predicted performance is lower. Moreover, starting from 384 cores, predicted performance from the Hybrid model is less than that which is obtained from actual measurements. As already discussed previously, this behaviour is related to overestimation of the computation steps in Hybrid model.

4.5.3.4 Hybrid-Async simulator using `big` test case

Figure 4.8 presents the same study as presented in the previous paragraph, but using the `big` test case. The computer used is the same IVANOE platform. This test case comprises 110592 tasks, that is 8 times larger than the number of tasks in the `small` test case, and the average number of tasks per core when using 768 cores is 144. In addition, the larger `MacroCell` size used for this test case, 20 × 20 × 20, improves the performance per core of the sweep kernel. Thus, on a single computing node, the performance of the PaRSEC implementation reaches 143.9 Gflop/s when using 12 cores, and corresponds to 51.1% of the theoretical peak performance of the node. This high performance per node is explained by the usage of SIMD units and a good data locality exposed by the sweep kernel, thus improving the arithmetic intensity. When using 768 cores, the performance of the sweep implementation with PaRSEC is 6.1 Tflop/s, which corresponds to 33.9% of the theoretical peak performance of the 64 nodes of IVANOE, and the parallel efficiency is 68%.

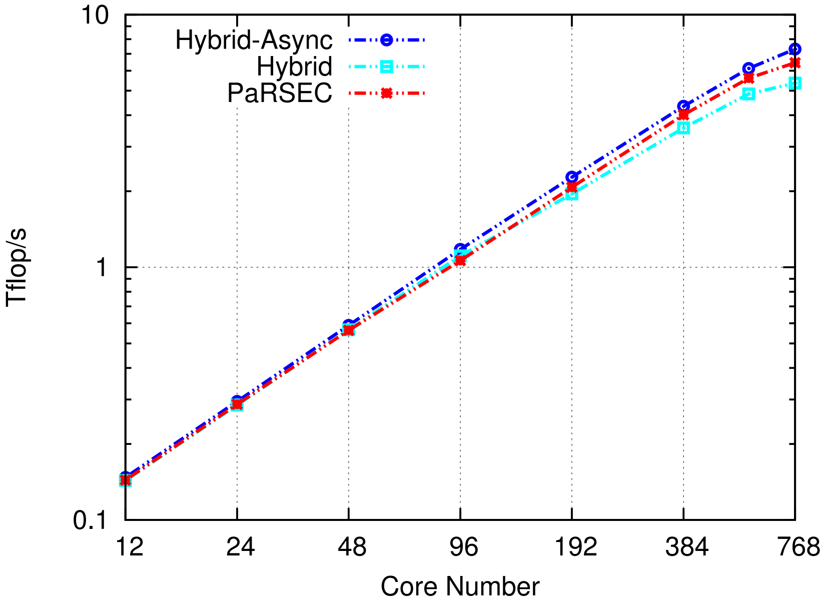


Figure 4.8: Performance of the PaRSEC implementation of the sweep using `big` test case on the IVANOE cluster.

To assess the PaRSEC performances on this test case, we ran both the Hybrid model and Hybrid-Async simulator using the same value of the communication overhead as used when we studied the optimal partitioning: $o_c = 1.8 \cdot 10^{-10}$ s/Byte. Unsurprisingly, the predicted performances from the Hybrid-Async model are close to actual measurements and at 768 cores the PaRSEC performances is 88.3% of the predicted value.

► In this chapter, we have presented a massively parallel implementation of the sweep operation, using task-based models on top of generic runtime systems: Intel TBB, PaRSEC and StarPU. We compared actual performances obtained from these implementations to the performances predicted by the theoretical models.

First, we found that the single-core performance of the StarPU implementation is lower than that of PaRSEC, suggesting that the internal cost of the StarPU runtime is higher than that of PaRSEC. Moreover, on a 24-cores NUMA node based on Intel Ivy-Bridge processors, the performance of the PaRSEC implementation reaches 367.6 Gflop/s, corresponding to 35.4% of the theoretical peak of the node. This performance is higher than those of StarPU and Intel TBB implementations (358.7 Gflop/s and 287.57 Gflop/s respectively).

On distributed multicore machines, we assessed that our Hybrid-Async simulator and Hybrid model are capable to predict the optimal partitioning as a function of the machine parameters and the test case in use. However, the time required to run the Hybrid-Async simulator is higher than that is required to run the Hybrid model because the simulator is more precise, and thus better predict the sweep computation time. Therefore, we will use Hybrid model for evaluating the optimal partitioning and the Hybrid-Async simulator for evaluating the maximum sustainable performances. Using 64 nodes of IVANOE platform, based on Intel Nehalem processors, the PaRSEC implementation with optimal partitioning, achieves 6.1 Tflop/s corresponding to 33.9% of the theoretical peak of the considered nodes.

Having assessed the performance of the sweep operation on modern massively parallel architectures, we can therefore build a massively parallel neutron transport solver on these architectures. This solver presented in Chapter 5.

Chapter 5

Full-core SN Calculations on Massively Parallel Architectures

Contents

5.1 Acceleration of source iterations (SI) 86

5.1.1 Diffusion Synthetic Acceleration (DSA) method 86

5.1.2 Piecewise DSA method (PDSA) 88

5.2 Validation and performances of Domino 91

5.2.1 Benchmarks 91

5.2.2 Validation and performances of the source iterations scheme 94

5.2.3 Efficiency of the PDSA scheme 97

5.2.4 Full-core 3D PWR calculations 101

In this thesis, we consider the discrete ordinates method for solving the neutron transport equation as presented in Chapter 1. We have seen that this method involves iterating over the scattering source for solving the monokinetic neutron transport equations. Each scattering iteration (also referred source iteration) involves a sweep over the spatial domain for a set of angular directions. As mentioned, this sweep operation gathers the vast majority of computations within the discrete ordinates method. In Chapter 4, we have presented a massively parallel implementation of this sweep operation on distributed multicore-based supercomputers, using the PaRSEC task-based runtime system. We have shown that the efficiency of this implementation compares well with the predicted performances from theoretical performance models. In this chapter, we consider the whole S_N algorithm as implemented in the Domino solver which integrates our task-based implementation of the sweep.

In section 5.1, we first recall the source iteration (SI) scheme and its convergence acceleration using the classical DSA in order to deal with strongly diffusive problems. Then, after recalling some limitations of the DSA in a parallel context, we present an implementation of a new acceleration scheme: PDSA, which extends the DSA scheme. In section 5.2, we present a validation study to assess the accuracy of the S_N method and its performance on a class of benchmarks including 3D PWR full-core models.

5.1 Acceleration of source iterations (SI)

As mentioned in section 1.3.1, the convergence of the scattering iterations (Algorithm 3) is very slow in highly diffusive media ($\Sigma_s \approx \Sigma_t$), and an acceleration scheme must be used to remedy this issue. One of the widely used acceleration scheme in this case, is the Diffusion Synthetic Acceleration (DSA) [69].

5.1.1 Diffusion Synthetic Acceleration (DSA) method

General presentation of the DSA

Here we just recall basics of this method, and the reader can refer to the paper for more details regarding its effectiveness and the Fourier analysis characterizing its convergence properties. Let us define $\epsilon^{k+\frac{1}{2}} = \psi - \psi^{k+\frac{1}{2}}$, as the error on the solution obtained after the $k+\frac{1}{2}^{\text{th}}$ iteration of the source iterations (SI) scheme, relative to the exact solution ψ, as defined by equation (1.15). By subtracting the equation on Line 3 of the Algorithm 3, in equation (1.15), the error ϵ on the angular flux satisfies the following transport equation:

$$L\epsilon^{k+\frac{1}{2}}(\vec{r},\vec{\Omega}) = \int_{S_2} d\vec{\Omega}'\Sigma_s(\vec{r},\vec{\Omega}'\cdot\vec{\Omega})\epsilon^{k+\frac{1}{2}}(\vec{r},\vec{\Omega}') + \Sigma_{s0}\left(\phi^{k+\frac{1}{2}} - \phi^k\right), \tag{5.1}$$

which is as difficult to solve as the original fixed-source transport problem (1.15). However, if an approximate solution $\tilde{\epsilon}^{k+\frac{1}{2}}$ of this equation was available, the scalar flux could be updated to:

$$\phi^{k+1} = \phi^{k+\frac{1}{2}} + \tilde{\epsilon}^{k+\frac{1}{2}}.$$

The idea of the DSA method is to use a diffusion approximation instead of solving transport equation (5.1). In Domino, the diffusion approximation relies on the simplified P_N (SP_N) method. This method was firstly introduced in [43], but in our case we focus on the mixed-dual formulation as presented in [73]. The choice of the diffusion approximation is firstly motivated by the simplicity of the diffusion operator, hence allowing to solve more easily the transport equation on the error (5.1). The second advantage of using this approximation is related to the error attenuation between successive iterations of the SI scheme. Indeed, the Fourier analysis performed in [69] shows that DSA attenuates low-frequency errors left by the transport sweep operation. The latter operation attenuates almost only high-frequency errors in highly diffusive problems.

Solution of the diffusion problem

Equation 5.1 is approximated by a monogroup SP1 problem with an isotropic cross-section (Σ_{s0}), as presented by the following problem in mixed dual formulation:

Problem 4. *Find* $(\epsilon, \vec{J}) \in L^2(\mathcal{D}) \times H(\mathcal{D}, div)$ *such that:*

$$\begin{cases} div\ \vec{J}(\vec{r}) + \Sigma_a \epsilon(\vec{r}) = S(\vec{r}) & in\ \mathcal{D} \\ \vec{\nabla}\ \epsilon(\vec{r}) + \dfrac{1}{D}\vec{J}(\vec{r}) = \vec{0} & in\ \mathcal{D} \\ \epsilon = 0 & on\ \partial\mathcal{D} \end{cases} \tag{5.2}$$

where:

$$S(\vec{r}) = \Sigma_{s0}\left(\phi^{k+\frac{1}{2}}(\vec{r}) - \phi^k(\vec{r})\right)$$

is the source term; $\Sigma_a = \Sigma_t - \Sigma_{s0}$ *the absorption cross-section, and* $D = \frac{1}{3\Sigma_a}$ *the diffusion coefficient.*

The complete description of the SP_N method is beyond the scope of this dissertation, but rather we just give its main characteristics as implemented in our SP_N solver DIABOLO [94]. Let us consider the general case of Problem 4 where the flux at the boundary is defined as:

$$\epsilon = \epsilon_b \text{ on } \partial\mathcal{D}.$$

Thus, multiplying the first line of equation (5.2) by $v \in L^2(\mathcal{D})$, the second by $\vec{w} \in H(\mathcal{D}, div)$, and finally applying the Green formula, we obtain the following mixed-dual variational problem [23]:

Problem 5. *Find* $(\epsilon, \vec{J}) \in L^2(\mathcal{D}) \times H(\mathcal{D}, div)$ *such that:*

$$\begin{cases} \displaystyle\int_{\mathcal{D}} div(\vec{J}(\vec{r}))\ v(\vec{r})d\vec{r} + \int_{\mathcal{D}} \Sigma_a(\vec{r})\epsilon(\vec{r})v(\vec{r})d\mathcal{D} = \int_{\mathcal{D}} S(\vec{r})v(\vec{r})d\vec{r}, \quad \forall v \in L^2(\mathcal{D}) \\ \displaystyle\int_{\mathcal{D}} \frac{1}{D(\vec{r})}\vec{J}(\vec{r}) \cdot \vec{w}(\vec{r})d\vec{r} - \int_{\mathcal{D}} \epsilon(\vec{r})div(\vec{w}(\vec{r}))d\vec{r} = -\int_{\partial\mathcal{D}} \epsilon_b(\vec{r})\vec{w}(\vec{r}) \cdot \vec{n}d\Gamma, \quad \forall \vec{w} \in H(\mathcal{D}, div), \end{cases} \tag{5.3}$$

where $\vec{n}$ *is the unit normal vector to the boundary* $\partial\mathcal{D}$.

Then, the Problem 5.3 is discretized spatially using the RTk mixed-dual finite element described in [98, 89]. To apply the DSA, computations are made with the RT0 element, which has 7 (resp. 4) degrees of freedom (DoFs) per cell in 3D (resp. 2D): 1 DoF for the scalar unknown and 6 (resp. 3) DoFs for the vector unknown (see Figure 5.1).

The DSA acceleration scheme [2] was proven effective, provided that the spatial discretization of the transport equation is consistent with the spatial discretization of the diffusion solver [70]. In [50], the author proved that this consistency requirement is lifted when using a Diamond Differencing scheme of order k for the transport equation, and RTk finite elements as a discretization scheme for the diffusion solver. As our SP_N solver uses also RTk finite elements, we

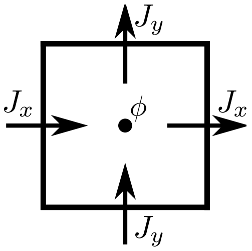


Figure 5.1: RT0 finite element in 2D: 5 DoFs (4 for the current and 1 for the scalar flux).

are assured of the stability of the DSA scheme. However, when integrated into a parallelized transport solver, DSA may become a bottleneck for the scalability of the transport solver if, for instance, a serial implementation of the diffusion solver is used. On the other hand, if the diffusion solver is parallelized, as presented in [10, 58] using a domain decomposition method, the iteration count to the solution increases with the number of subdomains, and can lead to a poor global scalability [121]. To remedy this issue, a variant of the DSA has been recently proposed by F. Févotte in [37].

5.1.2 Piecewise DSA method (PDSA)

The general presentation and the convergence proof of the PDSA method are given in [37].

General algorithm

We assume that the spatial domain $\mathcal{D}$ is split, along the 3 dimensions of the space, into $N = P \times Q \times R$ non-overlapping subdomains $\mathcal{D}_I$ such that: $\mathcal{D} = \cup_{I \in \mathcal{I}} \mathcal{D}_I$, where

$$\mathcal{I} = [\![1, P]\!] \times [\![1, Q]\!] \times [\![1, R]\!].$$

We set: $\Gamma_{IJ} = \partial\mathcal{D}_I \cap \partial\mathcal{D}_J$ the non-empty interfaces between subdomains of index I and J; $\Gamma_I = \partial\mathcal{D} \cap \partial\mathcal{D}_I$; and $\vec{n}_I$ the unit normal vector to $\partial\mathcal{D}_I$.

The first step of the PDSA method is, as in the case of the DSA method, an S_N transport sweep operation on the whole spatial domain. The second step, consisting of setting an approximation of the error on the scalar flux, is split in two sub-steps: we successively solve two diffusion problems, on each subdomain, respectively with homogeneous Neumann boundary conditions (equation (5.4)), and non-homogeneous Dirichlet boundary conditions (equation (5.5)).

$$\begin{cases} div\ \vec{J}_N^I(\vec{r}) + \Sigma_a \epsilon_N^I(\vec{r}) = S^I(\vec{r}) & \text{in } \mathcal{D}_I \\ \vec{\nabla}\ \epsilon_N^I(\vec{r}) + \dfrac{1}{D}\vec{J}_N^I(\vec{r}) = \vec{0} & \text{in } \mathcal{D}_I \\ \epsilon_N^I = 0 & \text{on } \partial\Gamma_I \\ \vec{\nabla}\epsilon_N^I \cdot \vec{n} = 0 & \text{on } \Gamma_{IJ} \end{cases} \tag{5.4}$$

$$\begin{cases} div\ \vec{J}_D^I(\vec{r}) + \Sigma_a \epsilon_D^I(\vec{r}) = S^I(\vec{r}) & \text{in } \mathcal{D}_I \\ \vec{\nabla}\ \epsilon_D^I(\vec{r}) + \dfrac{1}{D}\vec{J}_D^I(\vec{r}) = \vec{0} & \text{in } \mathcal{D}_I \\ \epsilon_D^I = 0 & \text{on } \Gamma_I \\ \epsilon_D^I = \dfrac{\epsilon_N^I + \epsilon_N^J}{2} & \text{on } \Gamma_{IJ} \end{cases} \tag{5.5}$$

This is a major shift from the classical DSA method, as we are no longer required to get the solution of the diffusion problem on the whole spatial domain. The first advantage of this method is that the explicit global synchronizations between the resolutions of the piecewise diffusion problems are largely reduced, hence allowing to fully parallelize the DSA method without efficiency loss. In addition, as we are going to see in section 5.2.3, the effectiveness of the PDSA method is comparable to that of the classical DSA method on a class of benchmarks.

The accelerated S_N flux is finally given by:

$$\phi_{\text{accel}} = \phi + \epsilon_D.$$

Treatment of boundary conditions

The Neumann boundary conditions in equation (5.4) correspond to a reflective boundary condition $\vec{\nabla}\epsilon \cdot \vec{n} = 0$, and its implementation poses no difficulty. However, as noted in [6], imposing non-homogeneous Dirichlet boundary condition such as in equation (5.5), represents several numerical challenges. Meanwhile, thanks to the mixed-dual variational formulation (5.3) we are able to bypass this issue. Indeed, considering the particular case of the second line of this variational formulation with the Dirichlet boundary conditions of equation (5.5), we obtain:

$$\int_{\mathcal{D}_I} \frac{1}{D(\vec{r})}\vec{J}(\vec{r})\cdot\vec{w}(\vec{r})d\vec{r} - \int_{\mathcal{D}_I} \epsilon_D(\vec{r})div(\vec{w}(\vec{r}))d\vec{r} = -\int_{\Gamma_{IJ}} \epsilon_b(\vec{r})\vec{w}(\vec{r})\cdot\vec{n}_I d\Gamma, \quad \forall\vec{w} \in H(\mathcal{D}, div), \tag{5.6}$$

where

$$\epsilon_b(\vec{r}) = \frac{\epsilon_N^I(\vec{r}) + \epsilon_N^J(\vec{r})}{2}.$$

The functions ϵ_N^I, which come from the first SP_N resolution, are naturally expanded on the RT0 finite element basis of as follows:

$$\epsilon_N^I = \sum_i E_I^i v_i,$$

where E_I^i is the unknown vector of the flux over the subdomain, and v_i are the basis functions for the flux (see Figure 5.2). On the other hand, equation (5.6) is also discretized using RT0

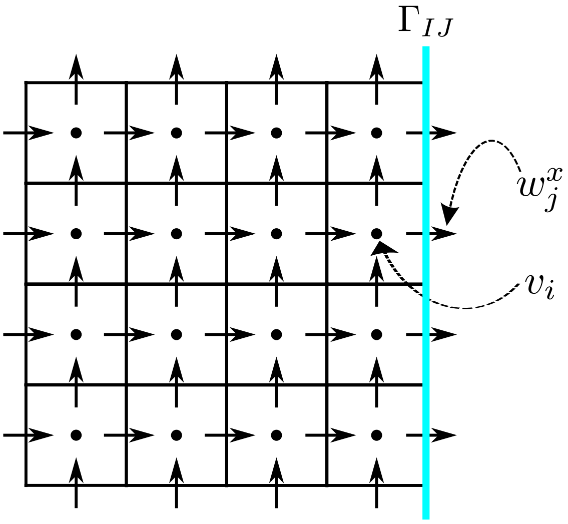


Figure 5.2: Correspondance between flux and current DoFs.

finite elements. The test functions for which it is evaluated are thus basis functions for the current unknown, hereafter denoted by $(\vec{w}_j)_j$. Without loss of generality, we suppose that the interface Γ_{IJ} is vertical (as seen for example on Figure 5.2). Therefore, $\vec{w}_j \cdot \vec{n} = w_j^x$ on Γ_{IJ}, and the integral in the right-hand side of equation (5.6) becomes:

$$B_j = \int_{\Gamma_{IJ}} \epsilon_b(\vec{r}) w_j^x(\vec{r}) d\Gamma$$

Due to the expression of RT0 basis functions, we can notice that $B_j = E_I^i$, where indices i and j are related as shown on Figure 5.2: the flux DoF indexed by i is associated to the cell whose boundary supports the current DoF indexed by j.

An illustration of the processing of the boundary conditions in the case of two subdomains is on Figure 5.3.

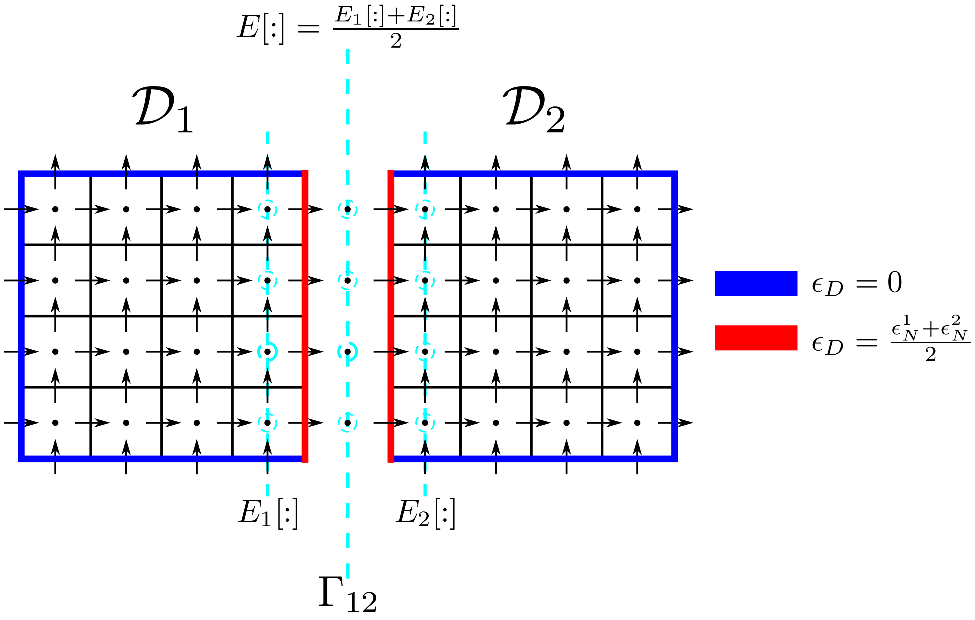

$E[:] = \frac{E_1[:]+E_2[:]}{2}$
$\mathcal{D}_1$
$\mathcal{D}_2$
$\epsilon_D = 0$
$\epsilon_D = \frac{\epsilon_N^1+\epsilon_N^2}{2}$
$E_1[:]$
$E_2[:]$
Γ_{12}

Figure 5.3: Illustration of the PDSA method on a domain split in two. The first step consists of solving two diffusion problems in parallel on $\mathcal{D}_1$ and $\mathcal{D}_2$, with Neumann boundary conditions. The second step solves also two diffusion problems with non-homogeneous Dirichlet boundary conditions: null flux boundary conditions on the external boundary of the domain and an average value of the flux at the inner interface.

Parallelization of the PDSA Method

Figure 5.4 illustrates a parallel implementation of the PDSA method in 2D, when the global domain is partitioned in two subdomains. The partitioning of the global domain uses the same block data distribution as for the sweep operation. As we mentioned previously, the diffusion problem on each subdomain is solved using our SP_N solver DIABOLO which is parallelized on shared memory system using INTEL TBB framework.

Hence, by mapping each subdomain to a single process, the resolution of the diffusion problems on $\mathcal{D}_1$ and $\mathcal{D}_2$, when applying the PDSA method, is naturaly done in parallel. Moreover, for the first step, the use of Neumann boundary conditions requires no communications with the neighboring processes. However, in the second step, each process needs to have the average

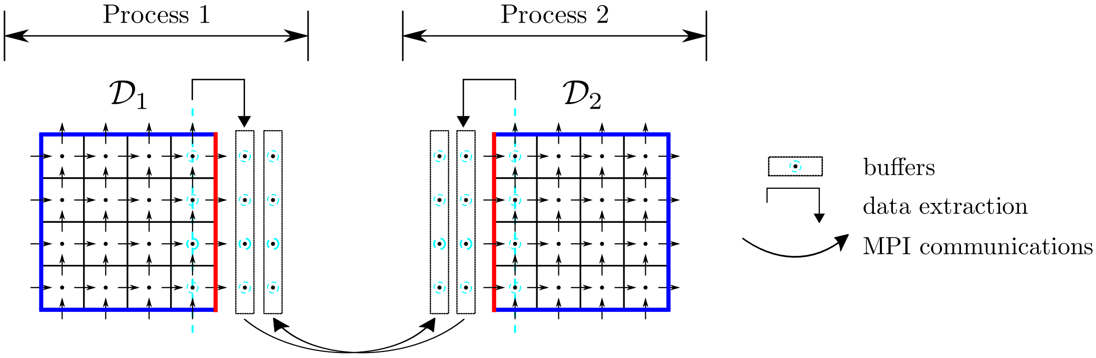


Figure 5.4: Illustration of the communication pattern in PDSA method on a domain split in two. Two point-to-point communications are needed to exchange flux at the interface between the two subdomains.

value of the scalar flux at the interfaces between its neighbors. Therefore, each process must performs send and receive operations to exchange data with its neighbors. These data exchanges are essentially point-to-point communications as only two processes are involved for each data exchange. To achieve this, we allocate two extra buffers per process: the first one is dedicated to store the extracted scalar flux at the interface between the subdomains which is then communicated to the neighboring process; while the second one is used as a reception buffer. We use asynchronous MPI communication primitives to exchange the flux at the interfaces.

5.2 Validation and performances of Domino

We assessed both the accuracy of the discretizations used in Domino and the efficiency of the PDSA method. This section presents experiments carried out to this end. We first start by presenting the different benchmarks used to perform our experiments.

5.2.1 Benchmarks

We used three different benchmarks to assess the accuracy and parallel performance of the Domino solver.

Kobayashi benchmarks [66] These one-group benchmarks are used to assess the accuracy of the flux distribution, on geometries having void regions, in a highly absorbing medium. A full description of these benchmarks is available in [66]. Among these benchmarks, we considered the Problem 1 as depicted on Figure 5.5, and the case *ii* characterized by 50% of scattering. Table 5.1 recalls the values of the cross-sections and neutron source used.

Takeda benchmarks [111] These benchmarks, consisting of four core models, are widely used for checking the validity of 3D neutron transport solvers. We considered the Model 1, which corresponds to a small Light Water Reactor (LWR) core. This model can be used with control rods inserted or not. In this study, we consider the former case, as depicted on Figure 5.6 (case 2). The two-group cross sections used in this model are characterized by a high scattering ratio ($c \simeq 0.98765$). Therefore, this model is a good candidate to show the effectiveness of the scattering acceleration methods.

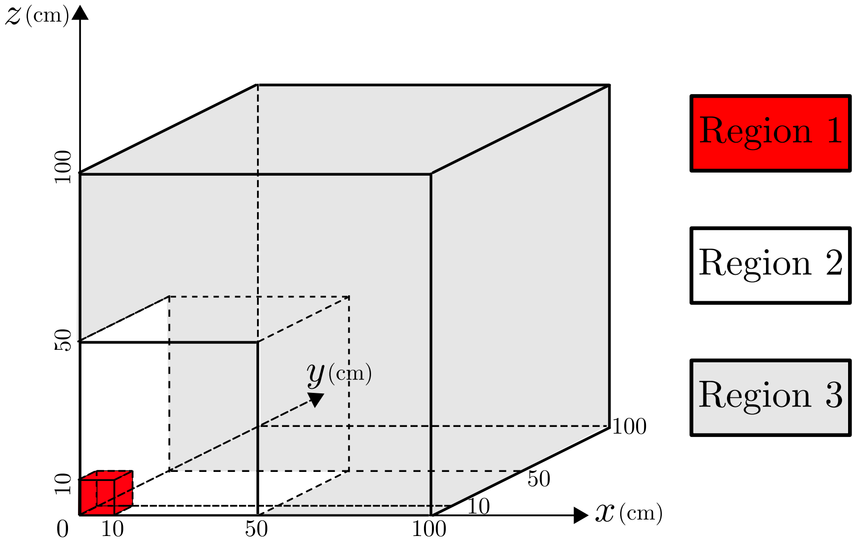
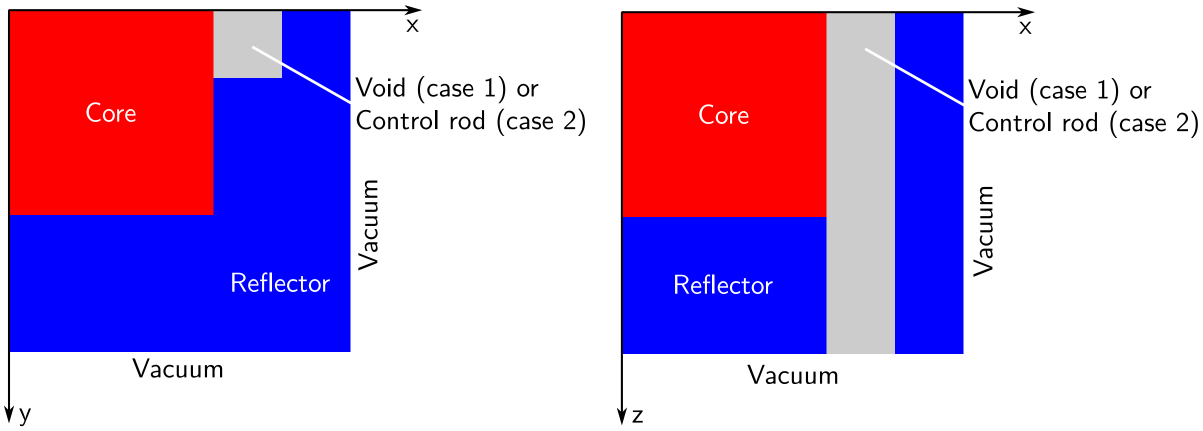

z (cm)
100
50
10
0
10
50
100
y (cm)
x (cm)
100
50
10
Region 1
Region 2
Region 3

Figure 5.5: Configuration of the KOBAYASHI benchmark [66] (Problem 1).

Region	S $(n\ cm^{-3}s^{-1})$	Σ_t (cm^{-1})	Σ_s (cm^{-1}) Problem i	 Problem ii
1	1	0.1	0	0.05
2	0	10^{-4}	0	0.5×10^{-4}
3	0	0.1	0	0.05

Table 5.1: One-group cross-sections and source strength S for the KOBAYASHI benchmark.


x
Core
Void (case 1) or
Control rod (case 2)
Vacuum
Reflector
Vacuum
y
x
Core
Void (case 1) or
Control rod (case 2)
Vacuum
Reflector
Vacuum
z

(a) XY (b) XZ

Figure 5.6: Core configuration the TAKEDA benchmarks [111].

The boundary conditions associated with KOBAYASHI and TAKEDA benchmarks are reflective. However, since we consider only vacuum boundary condition in this study, we have replicated the spatial domain for both benchmarks in order to use vacuum boundary conditions.

PWR 900 MW benchmarks These benchmarks correspond to a PWR 900 MW core, and enable to perform 2, 8 and 26 energy groups calculations. A full description of these benchmarks is available in [24]. It corresponds to a simplified 3D PWR first core loaded with 3 different types of fuel assemblies characterized by a specific Uranium-235 enrichment (low, medium and highly enriched uranium). There are no control rods inserted in this core model. Along the z-axis, the 360 cm assembly is axially reflected with 30 cm of water which results in a total core height of 420 cm. The 3 types of fuel assemblies appear on Figure 5.7 where the central assembly corresponds to the lowest enrichment, while the last row of fuel assemblies has the highest enrichment to flatten the neutron flux. Each fuel assembly is a 17×17 array of fuel pins, with a lattice pitch of 1.26 cm that contains 264 fuel pins and 25 water holes. The boundary condition associated with this benchmark problem is a pure leakage without any incoming angular flux. The associated nuclear data, 2-group, 8-group and 26-group libraries, derive from a fuel assembly heterogeneous transport calculation performed with the cell code DRAGON [42].

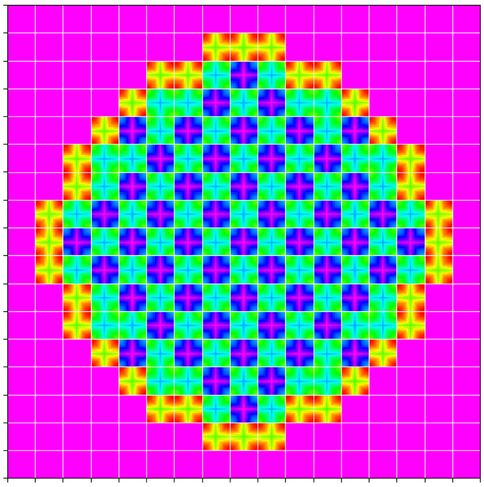


Figure 5.7: Radial view of the PWR 900 MW model [24].

Table 5.2 summarizes the discretization parameters for the considered benchmarks.

- For the KOBAYASHI benchmark, we considered a uniform 2-cm mesh step, leading to a spatial mesh of $100 \times 100 \times 100$ cells. We set the `MacroCell` sizes to $20 \times 20 \times 20$. A S_{16} angular quadrature.

- The spatial mesh size for the TAKEDA benchmark is the same as that of the `small` test case we used in Chapter 4, and we use the same `MacroCell` size. However, the angular quadrature used for the TAKEDA benchmark is S_8 (80 angular directions).

- The spatial mesh used for the PWR benchmarks is based on a pin-cell mesh in the $x - y$ plane. Each pin-cell is then subdivided into 70 (resp. 84) for the 26-group (resp. 2-group and 8-group). The spatial mesh is then refined by $2 \times 2 \times 2$ for the 8-group and 26-group, and by $2 \times 2 \times 9$ for the 2-group. The larger spatial mesh for the 2-group case enables to

	KOBAYASHI (Problem 1)	TAKEDA (Model 1)	PWR		
N_{G}	1	2	2	8	26
N_x	100	120	2×289		
N_y	100	120	2×289		
N_z	100	120	9×84	2×84	2×70
N_{dir}	288	80	168	80	288
N_{dof}	864.0 $\times 10^6$	829.44 $\times 10^6$	254.58 $\times 10^9$	107.76 $\times 10^9$	1.05 $\times 10^{12}$
Flops	7.2 $\times 10^9$	6.91 $\times 10^9$	2.12 $\times 10^{12}$	898.0 $\times 10^9$	8.75 $\times 10^{12}$
A_x	20	10	20		
A_y	20	10	20		
A_z	20	10	20		
$\epsilon_{k_{\mathrm{eff}}}$	–	10^{-6}	10^{-6}	10^{-6}	10^{-5}
ϵ_{ψ}	–	10^{-5}	10^{-6}	10^{-5}	10^{-5}
I_g	1	1	1	5	4

Table 5.2: Description of benchmarks and calculation parameters.

study the strong scalability of our implementation at high core count. For all the three benchmarks, we use `MacroCell`s of size $20 \times 20 \times 20$.

The calculation of DoF numbers consider 3 DoFs per cell, per energy group and per angular direction. $\epsilon_{k_{\mathrm{eff}}}$ and ϵ_{ψ} define the thresholds used to check the stopping criterions at iteration $n+1$ of the power algorithm, respectively on the eigenvalue and on the fission source as follows:

$$\frac{k_{\mathrm{eff}}^{n+1} - k_{\mathrm{eff}}^{n}}{k_{\mathrm{eff}}^{n}} < \epsilon_{k_{\mathrm{eff}}}, \qquad \frac{||\mathcal{F}\psi^{n+1} - \mathcal{F}\psi^{n}||}{||\mathcal{F}\psi^{n}||} < \epsilon_{\psi}. \tag{5.7}$$

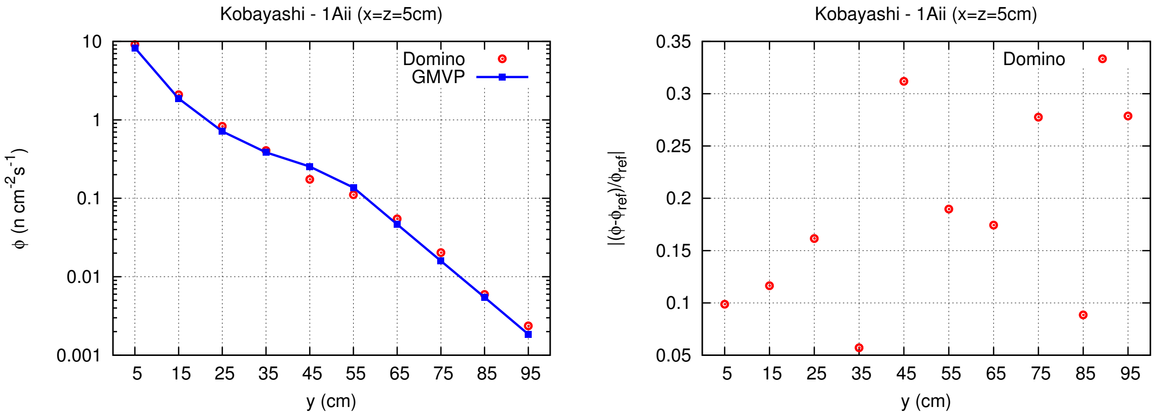
The following experiments were conducted by launching one MPI process per computing node and as many threads as available cores; keeping one core per node for the communication thread. All experiments were conducted in single precision. Computation times do not include setup (reading of cross-section files from the hard disk), but include all communications and stopping criterion checks. For all the experiments presented in the following, the setup time is less than a minute.

5.2.2 Validation and performances of the source iterations scheme

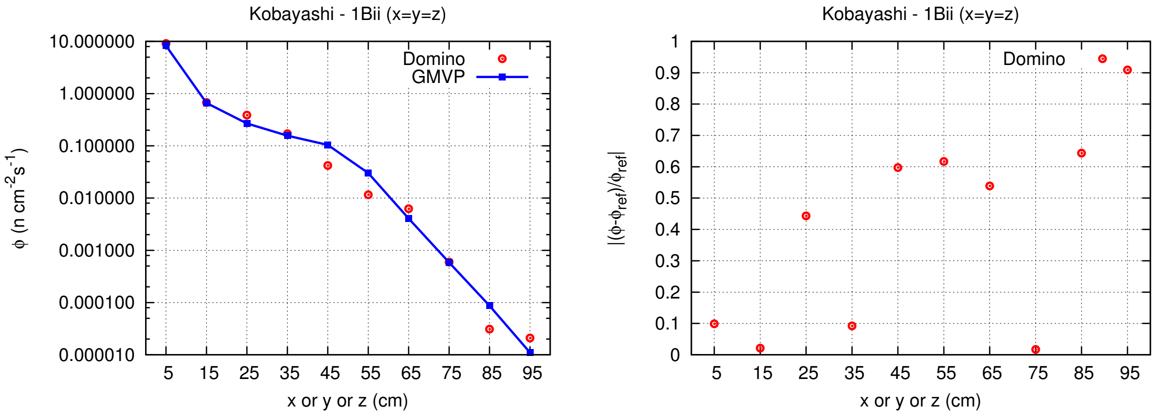
In this section, we present the performances of the source iterations scheme (S_N-only), without using the acceleration.

Kobayashi benchmark

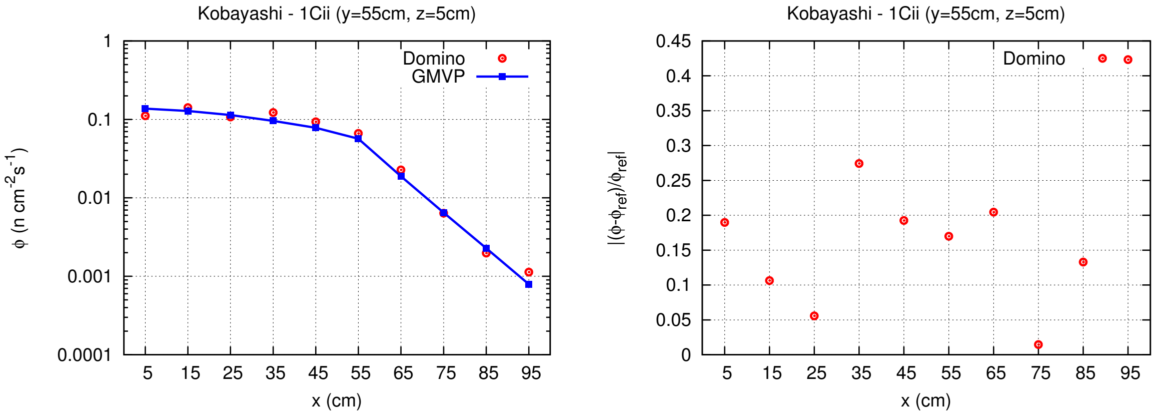
To check the accuracy of the source iterations as implemented DOMINO, we first consider the Problem 1*ii* of the KOBAYASHI benchmarks. The computation settings are defined in Table 5.2. We ran this benchmark on a single 24-cores computing node of the ATHOS platform, and the convergence was reached in 20 iterations. Figure 5.8 presents solutions obtained from DOMINO and the extracted reference results from [66]. The reference fluxes are defined for three set of



(a) $1Aii$ – Flux (b) $1Aii$ – Relative error on the flux



(c) $1Bii$ – Flux (d) $1Bii$ – Relative error on the flux



(e) $1Cii$ – Flux (f) $1Cii$ – Relative error on the flux

Figure 5.8: Comparison of the neutron fluxes for the Kobayashi problem $1ii$ (50% of scattering), obtained from DOMINO, with reference values obtained from GMVP Monte-Carlo code [66].

points: A (Figure 5.8a and Figure 5.8b), B (Figure 5.8c and Figure 5.8d) and C (Figure 5.8e and Figure 5.8f). We observe that the flux obtained from DOMINO follow the same trend as reference values. However, the relative errors on the flux are higher: 0.9 along the line $x = y = z$ (Figure 5.8d), at the mesh point defined by $x = 95$ cm. This observation is explained by the ray-effects as shown in [66]. It is possible to mitigate the ray effects using for example first-collision source approximation methods or the Gauss-Legendre (GL) quadrature formula with large of number of directions. Such a strategy is implemented in the radiation transport code DENOVO [36], enabling to get accurate solutions. Meanwhile, for our target applications (PWR core simulations), the ray-effects are negligible. Indeed, in a reactor core the neutron sources are uniformly distributed inside the reactor core, enabling to dramatically mitigate the ray effects.

Table 5.3 presents the computation time of the KOBAYASHI problems. The computation time

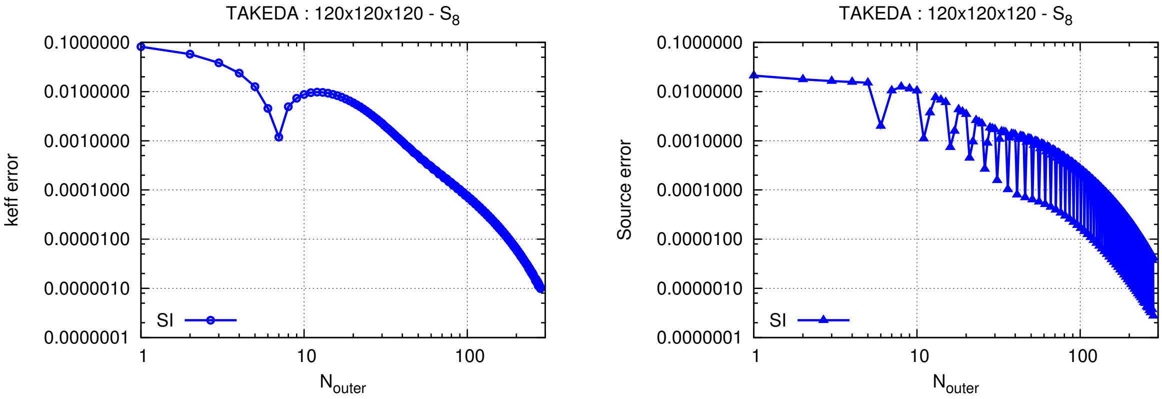
	Solution time (s) Problem 1ii
DOMINO	0.67
DENOVO	3.1

Table 5.3: S_N-only solution time for the Kobayashi problems 1ii, on a single computing node (dual Intel Xeon E5-2697v2 processors) of the ATHOS platform. Solution times for DENOVO were obtained on 16 processors of the Jaguar XT5 supercomputer, and correspond to the case where weighted diamond difference (WDD) spatial discretization scheme is used [36].

required to solve problem 1ii with DOMINO is 0.67 s, using a single 24-cores computing node of the ATHOS supercomputer. For the same problem, DENOVO code requires 3.1 s to get the solution using 16 processors [36] of the Jaguar XT5 supercomputer. Meanwhile, one should keep in mind that because we do not take into account symmetry boundary conditions, the spatial mesh that we used is eight times larger. In addition, the processor architectures used in both cases are different. Therefore, the aim of this comparison is to give an order of magnitude of how DOMINO compares to another existing neutron transport code.

Takeda benchmark

The second problem we considered for studying the performance of the source iterations is the Model 1 of the TAKEDA benchmark. We used a spatial mesh resolution of 0.416 cm, and a S_8 Level Symmetric quadrature. The resulting spatial mesh contains 120 × 120 × 120 cells. For the external iterations, the stopping criterion on the eigenvalue and on the fission source term are respectively set to: $\epsilon_{k_{\text{eff}}} = 10^{-6}$ and $\epsilon_{\psi} = 10^{-5}$. As there is no up-scattering in the TAKEDA benchmark, we set the number of Gauss-Seidel iterations (I_g) to one. Also, the number of source iterations (N_{src}) is fixed at one. Thereby, the convergence of the source iterations is determined by the convergence of the power algorithm on external iterations. We carried out this experiment on a single computing node of the ATHOS platform, and the result is presented in Figure 5.9. The convergence is reached in 282 iterations and the obtained eigenvalue is 0.962386. This value is well within the Monte Carlo reference error bar (0.9624 ± 0.0005, see [111]), therefore justifying the convergence of the source iterations scheme. The drop point in the convergence curve of the k_{eff}, as depicted in Figure 5.9a, highlights a non-monotonic convergence on the eigenvalue. Indeed, the relative error on the k_{eff} decreases until the iteration 7 and then changes the sign. In Figure 5.9b, the drop points correspond to the iterations where the Chebyshev acceleration method is applied (more details on the implementation of this method is available in [96]).



(a) Convergence on the k_{eff}.

(b) Convergence on the source term.

Figure 5.9: Convergence of the inverse power algorithm, without the acceleration, using TAKEDA benchmark.

We evaluated the region-averaged flux obtained from DOMINO on the Model 1 of the TAKEDA benchmarks. The relative discrepancies between the DOMINO solution and the reference solution is presented in Table 5.4. The maximum value of the discrepancies between the DOMINO solution and the reference solution is 0.44% (resp. 0.16%) on the fast (resp. thermal) group (see Table 5.4). These lower discrepancies illustrate that the convergence on the scalar flux is reached.

	$\lvert\delta\phi_1\rvert$ (%)	$\lvert\delta\phi_2\rvert$ (%)
Core	0.44	0.16
Control rod	0.33	0.20
Reflector	0.01	0.24

Table 5.4: Discrepancies on the scalar flux, between DOMINO and reference solution extracted from [111], for the Model 1 of the TAKEDA benchmark.

The study we presented in this section has validated the accuracy of the discretization schemes and the convergence of the source iterations scheme. In the next section, we are going to study the capabilities of the PDSA method for the reduction of the number of source iterations number.

5.2.3 Efficiency of the PDSA scheme

As mentioned, the application of the PDSA method after each source iteration, requires solving two successive SP_N problems. The solutions to the SP_N problems need to be converged to ensure the convergence of the PDSA scheme. As explained in [37], the convergence of the PDSA scheme can be determined using a parameter $\rho_{\text{PDSA}}^{\max}$ which depends on the scattering ratio c of the problem, and on the domain partitioning. This parameter, which represents the amplification factor of the PDSA scheme, is defined by equation (5.8):

$$\rho_{\text{PDSA}}^{\max} = \rho_{\text{DSA}}^{\max} + \tilde{\rho}_{\text{d}}^{\max} R_{\text{PDSA}}(\theta), \tag{5.8}$$

where: $\rho_{\mathrm{DSA}}^{\max}$ is the spectral radius of the DSA scheme; $\tilde{\rho}_{\mathrm{d}}^{\max}$ is an expression depending on the scattering ratio of the problem; and $R_{\mathrm{PDSA}}(\theta)$ is a closed-form formula which depends on a parameter θ, characterizing the optical thickness of subdomains and defined by:

$$\theta = \sqrt{3(1-c)}\tau, \tag{5.9}$$

where τ is the optical thickness of a subdomain[1]. According to the study performed in [37], R_{PDSA} converges towards 0 for optically thick subdomains. In this case, $\rho_{\mathrm{PDSA}}^{\max} \approx \rho_{\mathrm{DSA}}^{\max}$, so that the efficiency of the PDSA scheme is comparable to that of the classical DSA scheme. It should be noted that the criterion in equation (5.8) is defined for 1D, homogeneous cases. For a 3D, heterogeneous problem, R_{PDSA} is evaluated by considering the lowest optical thickness over all subdomains, and over the three dimensions. The uniform optical thickness obtained in this way is somewhat penalizing; a more precise approach would consist in evaluating the optical thickness using a homogenized set of cross-sections per subdomain. We will use the criterion to highlight the efficiency of the PDSA scheme for multidimensional partitionings. Let us first study the case of a single subdomain.

Case of a single subdomain

When there is only a single subdomain, then the PDSA scheme is equivalent to the classical DSA scheme, except that the former requires one extra SP_N resolution. Figure 5.10 shows the variation of outer iterations (N_{outer}) as a function of the number of SP_N iterations (N_{SPN}), using the Model 1 of the TAKEDA benchmarks (see section 5.2.1). We observe that by imposing one SP_N iteration, the convergence is reached in 38 outer iterations. Furthermore, N_{outer} decreases when N_{SPN} increases, and reaches a minimal value of $N_{\mathrm{outer}} = 14$ from $N_{\mathrm{SPN}} = 4$, suggesting that

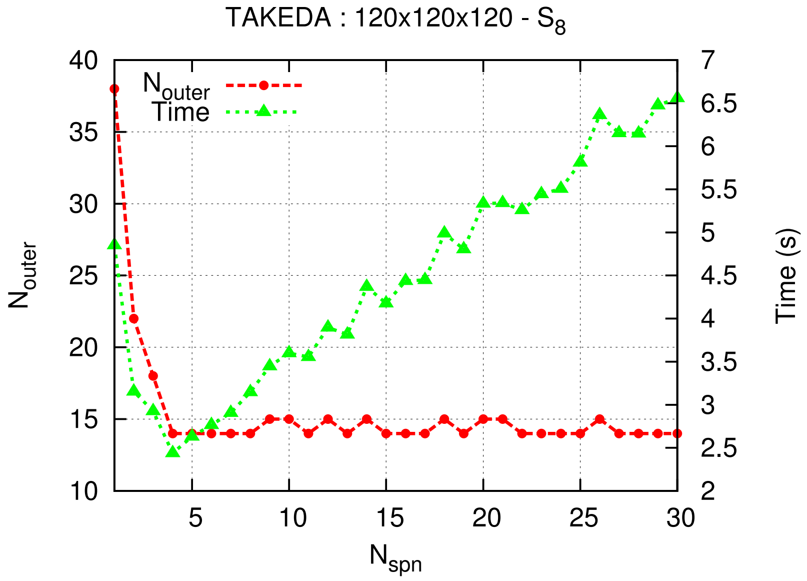


Figure 5.10: Convergence of the PDSA scheme using a partitioning of $(1, 1, 1)$.

---

[1] Let a 1D domain composed of n regions, of different properties; Σ^i, l_i, for $i = 1, 2, \ldots, n$ being respectively the total cross-section and length of each region. Then the optical thickness of the whole domain is defined by $\tau = \Sigma^1 l_1 + \Sigma^2 l_2 + \cdots + \Sigma^n l_n$.

the convergence of the SP_N solver is reached with this value of N_{SPN}. The corresponding eigenvalue is 0.962437 (see Table 5.5), which is slightly different (0.962386) than that was obtained without using the acceleration method.

Figure 5.10 also shows the total computation time as a function of N_{SPN}. As expected, the computation time follows the same trend as N_{outer} for the first four iterations, and reaches a minimal value of 2.43 s. However, after $N_{\text{SPN}} = 4$ the total computation time increases because the number of outer iterations is then constant. This study illustrates that for a given problem, there exists an optimal value of N_{SPN} which minimizes the total computation time.

k_{eff}	k_{eff} (Ref. Monte Carlo)	Δk_{eff} (pcm[1])	N_{outer} w/ PDSA	w/o PDSA	Time (s) w/ PDSA	w/o PDSA
0.962437	0.9624 ± 0.0005	3.77	14	281	2.43	27.46

Table 5.5: Eigenvalue and computation time for the Model 1 of the TAKEDA benchmark, on a 24-cores computing node of the ATHOS platform. For this experiment, we fixed the number of SP_N iterations to $N_{\text{SPN}} = 4$.

Multidimensional case

According to a preliminary experimental study, we found that in a multidimensional case, the optimal value of N_{SPN} required to minimize N_{outer} may be greater than that obtained in the case of a single domain. Therefore, for the following study, we used $N_{\text{SPN}} = 15$ to ensure the convergence of the SP_N solver. The goal of this study is to assess the convergence of the PDSA scheme for multidimensional partitionings. We considered the TAKEDA benchmark presented in section 5.2.1 and we evaluated the convergence of the solver for several partitionings (between $(1, 1, 1)$ to $(10, 10, 10)$) of the global domain . For each partitioning, we evaluated $\rho^{\text{max}}_{\text{PDSA}}$, and N_{outer}. The result is depicted in Figure 5.11. Each point corresponds to a given partitioning and thus is associated with a specific value of $\rho^{\text{max}}_{\text{PDSA}}$. The gradient colors are used to indicate the value of $P+Q+R$, for a partitioning (P, Q, R). For all the considered partitionings, the PDSA scheme converges and enables to reduce N_{outer} as compared to the source iteration (SI). Furthermore, each value of $\rho^{\text{max}}_{\text{PDSA}}$ is associated with several partitionings. Partitionings with larger values of $P + Q + R$ are those giving larger number of iterations. This latter observation suggests that the $\rho^{\text{max}}_{\text{PDSA}}$ indicator does not captures 3D effects. However, it seems to conservatively indicate partitionings where PDSA will converge.

Important remark. *For a fixed number of cores, the partitioning of a spatial domain according to a* Flat *implementation will feature a larger number of subdomains as compared to that of* Hybrid *approach. Consequently, the sum $P+Q+R$ will be larger, and consequently the convergence of a* Flat *implementation will be slower (Figure 5.11) than that of a* Hybrid *implementation.*

Figure 5.12 presents the convergence of the PDSA scheme for a multidimensional partitioning of $(4, 4, 2)$. For this partitioning, we found that $N_{\text{SPN}} = 3$ iterations for the SP_N solver minimizes the total computation time which is 2.38 s, and the corresponding number of outer iterations is $N_{\text{outer}} = 62$. According to this observation, the following results are obtained by selecting optimal values of N_{SPN}.

---

[1] pcm (per cent mille) is a widely used unit in nuclear engineering community for quantifying the reactivity of a reactor core.

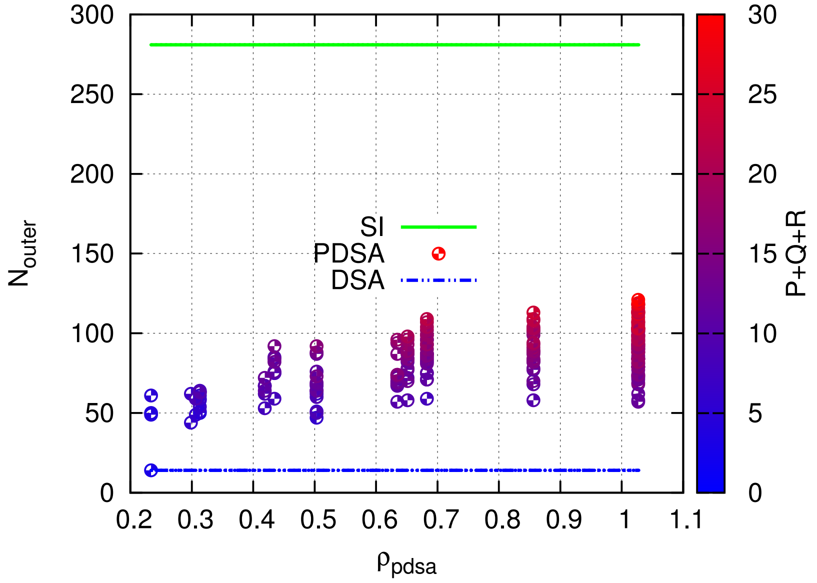


Figure 5.11: Convergence of the PDSA scheme as a function of $\rho^{\max}_{\mathrm{PDSA}}$.

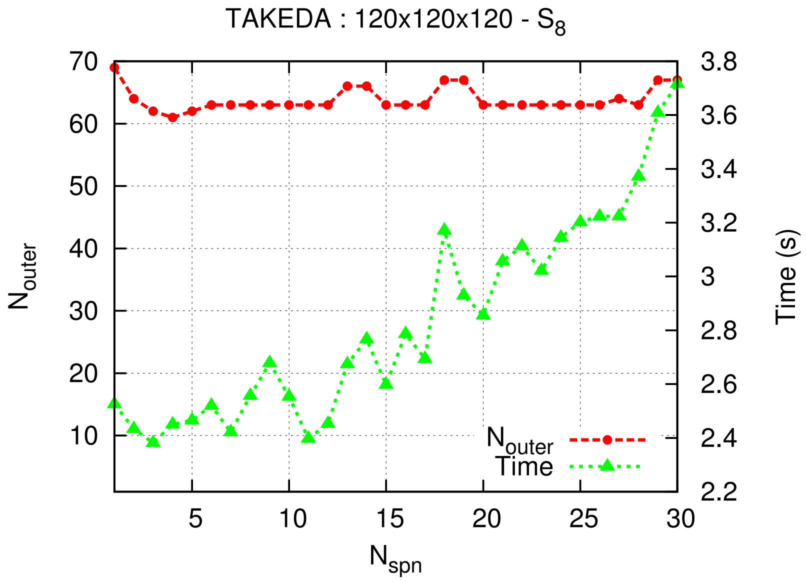


Figure 5.12: Convergence of the PDSA scheme using a partitioning of (4, 4, 2).

5.2.4 Full-core 3D PWR calculations

In this section, we present full-core k_{eff} computations using the 3D PWR core models described in section 5.2.1. The calculation parameters are defined in Table 5.2. From a preliminary study with a single subdomain, we found that the optimal number of SP_N iterations is one, for each of the three benchmarks. Therefore, all the following results are obtained using this value.

2-group PWR core model

Table 5.6 presents a strong scalability study on the ATHOS platform, using the 2-group PWR core model. As expected, for this benchmark characterized by a high scattering ratio ($c \approx 0.97$), the convergence of the source iterations is very slow. For instance, without using the PDSA scheme, 2116 external iterations are required to reach the convergence with a partitioning of $(2, 2, 1)$. The total computation time associated with this partitioning is 7724.4 s, of which 81.1% represents the time spent in the sweep operation, highlighting that the sweep operation is still dominant for the considered benchmark. Using the PDSA scheme, the number of external iterations is largely reduced to 92 corresponding to an acceleration of 23. Indeed, for this benchmark $\rho^{\text{max}}_{\text{DSA}} \approx 0.21$ which is similar to the value of $\rho^{\text{max}}_{\text{PDSA}}$ associated with the partitioning $(2, 2, 1)$. Thereby, the convergence of the PDSA scheme is optimal. Furthermore, the total computation time drops to 451.6 s representing a speed-up (relative to the case without PDSA) of 17.1. It should be noted that this speed-up is less than the optimal value of 23, even if this is the case for the sweep operation (improvement by a factor of 22.4). This is justified by the additional computational costs associated with the PDSA scheme: the two SP_N resolutions ($T_{\text{spn}} = 47.8$ s) and the communications required to exchange data between subdomains ($T^{\text{comm}}_{\text{PDSA}} = 35.7$ s). Considering the run times when the PDSA scheme is used, we obtain that the performance of the sweep operation reaches a 2.7 Tflop/s on four computing nodes of ATHOS, corresponding to 65.2% of the theoretical peak performance of the four nodes. Experiments with other partitionings

	Partitioning	$(2, 2, 1)$	$(2, 2, 2)$	$(4, 2, 2)$	$(4, 4, 2)$
	N_{cores}	96	192	384	768
$\rho^{\text{max}}_{\text{PDSA}}$		0.21	0.21	0.21	0.21
w/o PDSA	N_{outer}	2116	2120	2116	2125
	T_{sweep} (s)	6270.2	3831.0	2609.3	2289.8
	T_{total} (s)	7724.4	4543.6	2974.2	2479.0
	% sweep	81.1	84.3	87.7	92.3
w/ PDSA	N_{outer}	92	95	97	81
	T_{sweep} (s)	279.9	144.6	77.0	41.0
	T_{spn} (s)	47.8	24.6	12.4	5.1
	$T^{\text{comm}}_{\text{PDSA}}$ (s)	35.7	30.4	15.6	7.9
	T_{total} (s)	451.6	245.7	129.2	65.0
	% sweep	61.9	58.8	59.5	63.0
Perf. (Tflop/s)	sweep	2.7	3.5	5.3	8.3
	DOMINO	0.8	1.6	3.2	5.3
speed-up		17.1	18.5	23.0	45.7

Table 5.6: Solution times for a S_{12} 2-group 3D PWR k_{eff} computation on the ATHOS platform.

yield similar trends. In particular, using 768 cores, the performance of the sweep operation is 8.3 Tflop/s, corresponding to 24.72% of the peak of the corresponding 32 nodes.

In Table 5.7, we report a performance comparison between DOMINO and the radiation transport code DENOVO on the 2-group problem. We used 64 computing nodes of the ATHOS supercomputer, distributed into a grid of $4 \times 4 \times 4$ processes. The corresponding global computation time for the whole solver is 0.8 min. As a comparison, the DENOVO code solves a slightly different version of this 2-group benchmark in 2.05 min on the Jaguar XT5 supercomputer [28] (18688 compute nodes, each with dual 2.6 GHz AMD 6-core Istanbul processor). Please note that this machine differs from ATHOS and as a consequence the performance comparison is not very precise. In addition, the axial meshes used in DOMINO and DENOVO are slightly different (756 vs 700). Nonetheless, we hope that this comparison provides a correct trend on how DOMINO compares to DENOVO.

	DENOVO PWR 2g [28]	DOMINO PWR 2g
Computer	Jaguar	ATHOS
$N_x \times N_y$	578×578	578×578
N_z	700	756
N_{dir}	168	168
N_{G}	2	2
ϵ_{keff}	1×10^{-3}	1×10^{-6}
N_{dof} $(\times 10^9)$	78.6	84.9
N_{cores}	20400	1536
k_{eff}	–	1.019573
T_{total} (min)	2.05	0.8

Table 5.7: DOMINO-DENOVO Comparison for a S_{12} 2-group 3D PWR k_{eff} computation.

8-group PWR core model

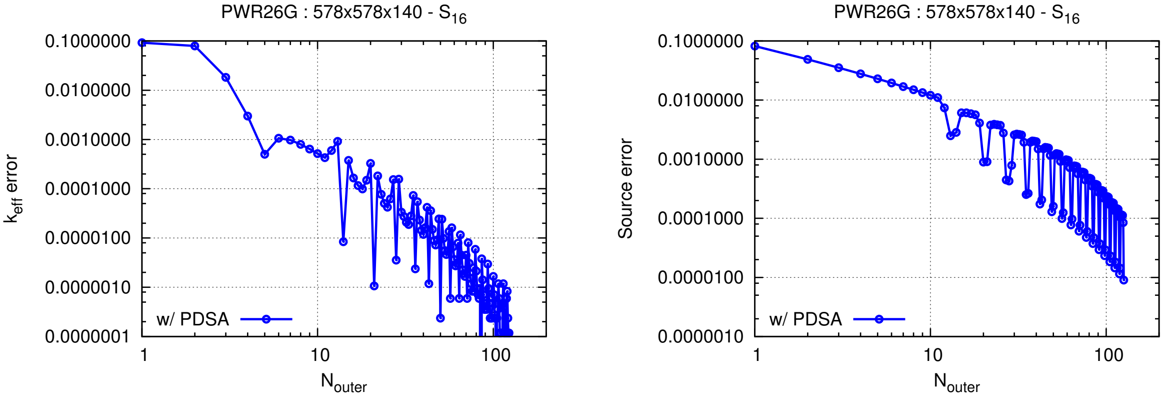
Table 5.8 presents performance results of a S_{12} 8-group 3D PWR k_{eff} computations using 64 computing nodes of the ATHOS cluster partitioned into $(4, 4, 4)$. As in the 2-group benchmark presented previously, $\rho^{\max}_{\text{PDSA}}$ with this partitioning is similar to $\rho^{\max}_{\text{DSA}}$. The convergence on this benchmark is reached in 65 external iterations, and the obtained eigenvalue is $k_{\text{eff}} = 1.009408$. This number of external iterations is similar to that was obtained for a run with a single subdomain. The total computation time is 128.85 s of which 91.53 s comes from the sweep operation, illustrating that the sweep operation is still dominant (71% of the total time).

(P, Q, R)	$\rho^{\max}_{\text{PDSA}}$	N_{outer}	T_{sweep} (s)	T_{spn} (s)	$T^{\text{comm}}_{\text{PDSA}}$ (s)	T_{total} (s)
$(4, 4, 4)$	0.19	65	91.53	7.84	0.86	128.85

Table 5.8: Solution times for a S_{12} 8-group 3D PWR k_{eff} computation.

26-group PWR core model

Figure 5.13 presents the convergence of a k_{eff} computation on the 26-group 3D PWR core model, using the PDSA scheme. We used 64 computing nodes of the ATHOS cluster partitioned into (4, 4, 4). As in Figure 5.9b, the drop points in Figure 5.13b correspond to the iterations where the Chebyshev acceleration method is applied. The detailed computing times for this benchmark are reporterd in Table 5.9. The convergence is reached in 126 outer iterations, for a global solver time of 2763.52 s. The obtained eigenvalue is $k_{\text{eff}} = 1.008358$. As in the case of 8-group, we did not observe any increase on the number of external iterations as compared to a run with a single domain. This is a remarkable result, highlighting the perfect efficiency of the PDSA method on representative benchmarks of our target applications.



(a) k_{eff} error (b) Fission source error

Figure 5.13: Convergence of DOMINO using the 26-group PWR benchmark. We used 64 computing nodes partitioned into (4, 4, 4).

(P, Q, R)	$\rho_{\text{PDSA}}^{\max}$	N_{outer}	T_{sweep} (s)	T_{spn} (s)	$T_{\text{PDSA}}^{\text{comm}}$ (s)	T_{total} (s)
(4, 4, 4)	0.14	126	2226.42	56.56	147.2	2763.52

Table 5.9: Solution times for a S_{16} 26-group 3D PWR k_{eff} computation.

► In this chapter, we have studied the accuracy and the performances of our massively parallel approach for solving the neutron transport equation according to the discrete ordinates method. We have first integrated our task-based implementation of the sweep with PARSEC in the DOMINO solver, then we integrated a new acceleration method for the scattering iterations required to speed-up the convergence for strongly diffusive problems. All the experiments have been carried-out using DOMINO solver, and we used three class of benchmarks.

The KOBAYASHI benchmarks series were used to assess the accuracy of the discretization schemes and to study the performances of the source iterations. We considered the cases with 50% of scattering. We found that for some regions, the discrepencies between the flux obtained from DOMINO and the reference solution are particularly large (rel. error of 2.4) due to ray-effects. There exists numerical methods that can help to mitigate the ray-effects. However, for strongly diffusive problems such as PWR cores the ray-effects are not present. The runtime performances of the DOMINO solver compares favorably with the DENOVO on the KOBAYASHI problems.

The second benchmark that we have studied is the TAKEDA benchmark where the control rods are inserted. On this benchmark with a high scattering ratio, we have assessed both the accuracy of the solution and the performances of the PDSA scheme. We have highlighted the correlation between the convergence of PDSA scheme with an indicator, $\rho_{\mathrm{PDSA}}^{\max}$, depending on the partitioning of the domain. The lower bound of this indicator is the spectral radius of the classical DSA method. Therefore, we have found that the convergence of the PDSA scheme improves for partitionings with lower values of this indicator. According to our study, we found that even though $\rho_{\mathrm{PDSA}}^{\max}$ is defined for 1D cases, it enables us to conservatively predict convergence of 3D heterogeneous problems.

Finally, we have studied the performances of DOMINO on 3D PWR core models. For these benchmarks, the convergence of PDSA scheme is optimal and requires roughly the same number of iterations as a global DSA scheme to reach the convergence. Thereby, the performance of the DOMINO solver is very satisfactory. For instance, on a 2-group k_{eff} computation, the performance of the sweep operation reaches 8.3 Tflop/s using 768 cores of the ATHOS platform. This performance corresponds to 24.72% of the the theoretical peak of the considered nodes.

Conclusion and Future Work

Conclusion

The goal of the research presented in this thesis was to study, and to propose a suitable solution, to the challenges posed by the use of hierarchical massively parallel computers, for solving the neutron transport equation according to the discrete ordinates method (S_N). The addressed challenges were manifold: design of efficient algorithms capable to handle the SIMD paradigm on modern CPUs; efficient utilization of multicore-based clusters by means of emerging task-based models on top of generic runtime systems; and the use of efficient numerical methods.

From this perspective, we have designed theoretical performance models of the sweep operation intended to bound its performances both on a single CPU and on hierarchical multicore-based architectures. These performance models have been used to establish a strategy that we have adopted to meet the above mentioned challenges.

At the processor core level, we have shown that the SIMD capabilities of modern processors impose some constraints on the design of efficient computational kernels. In the case of the sweep operation, we have studied different strategies for the vectorization of the computations. We have theoretically proved that the efficiency of the vectorization over the spatial variable is limited by the padding that must be used to ensure data alignment. However, the vectorization over the angular variable allows to reduce the overhead due to the paddings. In order to explicitly take advantage of the vector units, we have studied and validated the use of generic programming models in C++ for exploiting directly the SIMD units corresponding to the target architecture. We have shown that this strategy allows to preserve the modularity of the code and to enhance its performance portability. On a Westmere processor, the performance of the angular vectorization reaches a maximum of 13.6 Gflop/s on a single CPU, corresponding to 62.9% of the theoretical peak performance of that CPU.

From this efficient single threaded implementation of the sweep kernel, we addressed the challenges posed by the parallel execution of the sweep on multi-processor architectures. We first built an accurate parallel sweep simulator in order to explore the different parallel spatial decomposition of the transport sweep. Furthermore, we used our sweep simulator to justify the need for a task-based implementation of the sweep operation in order to maximize its performances on multicore-based architectures. Then, we compared different emerging task-based models on top of generic runtime systems (INTEL TBB, STARPU, PARSEC). As a result, the PARSEC framework based on parametrized DAG model, produced the most efficient implementation for the Cartesian transport sweep. This PARSEC based implementation helped us to show that our performance model accurately predicts optimal partitionings. Using optimal partitioning, the performance of the sweep operation reaches 6.1 Tflop/s of 768 cores of the IVANOE supercomputer, which corresponds to 33.9% of the theoretical peak performance of this set of computational resources.

Finally, we addressed the challenge of converging the scatter iterations in highly diffusive media such as the PWR cores. We have implemented and studied the convergence of a new acceleration scheme (PDSA) that naturally suits our Hybrid parallel implementation. The efficiency of the PDSA scheme have been investigated on the reference TAKEDA benchmark, according to a convergence indicator defined by the PDSA scheme. This indicator depends on the optical thickness of the subdomains and has a lower bound equal to the spectral radius of the classical DSA scheme. We have shown that when this indicator is similar to the spectral radius of the classic DSA scheme, as in the case of PWR cores, the number of external iterations required for convergence is the same as that of DSA.

The combination of all these techniques have enabled us to develop a massively parallel version of the DOMINO solver. It is capable of tackling the challenges posed by the neutron transport simulations and compares favorably with state-of-the-art solvers such as DENOVO. For a typical 26-group PWR calculations involving 1.02×10^{12} DoFs, the time to solution required by the DOMINO solver is 45 min using 1536 cores. Consequently, this DOMINO solver can be used by nuclear power plant operators such as EDF for improving the efficiency and safety of nuclear power plants.

Future work

The work presented in this thesis has raised several open questions for future research.

One such question is oriented towards the design of efficient kernels for accelerator-based architectures. Early works on this topic such as in [44, 57] require the use of quadrature formula featuring larger numbers of angular directions in order to achieve acceptable performance on GPU-based clusters. Recently, a new multilevel decomposition in energy has been introduced in [35], which is based on the use of Krylov methods for solving multigroup equations. This decomposition enables to extract a larger number of parallel tasks, for problems having large number of energy groups which is a requirement for maximizing the occupation of GPUs. However, for problems with few angles and energy groups a solution for maximizing the efficiency would consists of merging the angular and spatial variables.

In Chapter 3, we have presented our Hybrid model and the Hybrid-Async simulator that can be used to predict the computation time of the sweep operation. One of the parameters used in these models is T_{grind}: the computation time per cell, per group, and per angular direction. This parameter was obtained through experimental measurements. A challenge raised by this process is the sensitivity of the model predictions to T_{grind}. Therefore, it is of great interest to accurately estimate this parameter according to machine-dependent constants such as sizes and hierarchy of caches, clock rate, instruction and the microarchitecture of the target CPU.

Furthermore, our performance models can be extended to describe the behavior of the whole S_N algorithm including the computation time required by the SP_N resolutions used by the PDSA scheme. To achieve this, it is required to develop a performance model of the SP_N method as in [93].

In a previous study [25], on shared memory systems, we found that initializing the external iterations of the S_N algorithm by a flux obtained from a SP_N calculation can reduces the number of external iterations required by the transport solver by a factor up to 2.5. Such a strategy can therefore significantly reduce the global computation time. To efficiently take advantage of this result, it is required to have a distributed implementation of the SP_N solver as in [10, 58].

Finally, in this thesis we have studied the stationary form of the neutron transport equation.

Hence, this work provides an entry point towards time-dependent neutron transport simulation as in [87]. However, time integration requires the storage of the angular flux. As we have shown in Chapter 2, the storage of the currents decreases the arithmetic intensity of the sweep kernel. Therefore, efficient time-dependent neutron transport simulation represents a big challenge for future research.

Appendix A

Presentation of the test platforms

We consider four computing platforms, based on x86 processors, as presented on Table A.1.

Machine Name		BIGMEM	IVANOE	ATHOS
Memory per node	(GB)	1024	24	64
Processor Name		Intel Xeon E7-8837	Intel Xeon X5670	Intel Xeon E5-2697 v2
SIMD width	(bits)	128	128	256
Frequency	(GHz)	2.66	2.93	2.7
N_{socket}		4	2	2
N_{cores}/socket		8	6	12
N_{cores}/node		32	12	24
SP Th. peak perf./node	(GFlop/s)	680.96	281.28	1036.8
Interconnect		–	InfiniBand QDR	InfiniBand FDR
Compiler		gcc 5.1	gcc 4.7.2	gcc 5.1

Table A.1: Characteristics of the target machines.

Appendix B

Publications

Publications in journal

[1] S. Moustafa, I. Dutka-Malen, L. Plagne, A. Ponçot, and P. Ramet. “Shared memory parallelism for 3D Cartesian discrete ordinates solver”. In: *Annals of Nuclear Energy* 82 (2015). Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013, SNA + MC 2013. Pluri- and Trans-disciplinarity, Towards New Modeling and Numerical Simulation Paradigms, pp. 179–187. ISSN: 0306-4549.

[2] S. Moustafa, M. Faverge, L. Plagne, and P. Ramet. “Cartesian SN Neutron Transport Simulation on Massively Parallel Architectures”. In preparation for submission to Journal of Computational Physics.

Publications in conferences with proceedings

[1] S. Moustafa, M. Faverge, L. Plagne, and P. Ramet. “Parallel 3D Sweep Kernel with PaRSEC”. In: *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS), 2014 IEEE Intl Conf on.* Aug. 2014, pp. 1253–1254.

[2] S. Moustafa, M. Faverge, L. Plagne, and P. Ramet. “3D Cartesian Transport Sweep for Massively Parallel Architectures with PaRSEC”. In: *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International.* May 2015, pp. 581–590.

[3] S. Moustafa, F. Févotte, B. Lathuilière, and L. Plagne. “Vectorization of a 2D–1D Iterative Algorithm for the 3D Neutron Transport Problem in Prismatic Geometries”. In: *SNA+ MC 2013-Joint International Conference on Supercomputing in Nuclear Applications+ Monte Carlo.* EDP Sciences. 2014, p. 04102.

Bibliography

[1] Michael P Adams, Marvin L Adams, W Daryl Hawkins, Timmie Smith, Lawrence Rauchwerger, Nancy M Amato, Teresa S Bailey, and Robert D Falgout. "Provably Optimal Parallel Transport Sweeps on Regular Grids". In: *Proc. International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering, Idaho*. 2013 (cited on pages 3, 43).

[2] Raymond E Alcouffe. "Diffusion synthetic acceleration methods for the diamond-differenced discrete-ordinates equations". In: *Nuclear Science and Engineering* 64.2 (1977), pp. 344–355 (cited on page 87).

[3] AMD AMD. "Core Math Library (ACML)". In: *URL http://developer. amd. com/acml. jsp* (2012) (cited on page 25).

[4] VS Anil Kumar, Madhav V Marathe, Srinivasan Parthasarathy, Aravind Srinivasan, and Sibylle Zust. "Provable algorithms for parallel generalized sweep scheduling". In: *Journal of Parallel and Distributed Computing* 66.6 (2006), pp. 807–821 (cited on page 43).

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures". In: *Proceedings of the 15th International Euro-Par Conference*. Springer, Aug. 2009. DOI: `10.1007/978-3-642-03869-3_80` (cited on pages 66, 67).

[6] I. Babuška and Manil Suri. "The Treatment of Nonhomogeneous Dirichlet Boundary Conditions by Thep-version of the Finite Element Method". In: *Numer. Math.* 55.1 (Jan. 1989), pp. 97–121. ISSN: 0029-599X. DOI: `10.1007/BF01395874` (cited on page 89).

[7] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortı, and G. Quintana-Ortı. "Parallelizing Dense and Banded Linear Algebra Libraries using SMPSs". In: *Concurrency and Computation: Practice and Experience* 21.18 (2009), pp. 2438–2456 (cited on page 67).

[8] R. S. Baker and K. R. Koch. "An SN algorithm for the massively parallel CM200 computer". In: (1998) (cited on pages 3, 42).

[9] Randal S. [Los Alamos National Laboratory] Baker. *PARTISN on Advanced/Heterogeneous Processing Systems*. Feb. 2013. DOI: `10.2172/1063248` (cited on pages 3, 44).

[10] M. Barrault, B. Lathuilière, P. Ramet, and J. Roman. "Efficient parallel resolution of the simplified transport equations in mixed-dual formulation". In: *Journal of Computational Physics* 230.5 (2011), pp. 2004–2020. ISSN: 0021-9991. DOI: `http://dx.doi.org/10.1016/j.jcp.2010.11.047` (cited on pages 88, 106).

[11] Paul M Bennett. "Sustained systems performance monitoring at the US Department of Defense high performance computing modernization program". In: *State of the Practice Reports*. ACM. 2011, p. 3 (cited on page 3).

[12] Alex F Bielajew and DWO Rogers. “PRESTA: the parameter reduced electron-step transport algorithm for electron Monte Carlo transport”. In: *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms* 18.1 (1986), pp. 165–181 (cited on page 2).

[13] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Hérault, Pierre Lemarinier, and Jack Dongarra. “DAGuE: A generic distributed DAG engine for High Performance Computing”. In: *Parallel Computing* 38.1-2 (2012) (cited on pages 66, 67, 77).

[14] Judith F Briesmeister et al. “MCNPTM-A general Monte Carlo N-particle transport code”. In: *Version 4C, LA-13709-M, Los Alamos National Laboratory* (2000) (cited on page 2).

[15] Darius Buntinas, Brice Goglin, David Goodell, Guillaume Mercier, and Stephanie Moreaud. “Cache-efficient, intranode, large-message MPI communication with MPICH2-Nemesis”. In: *Parallel Processing, 2009. ICPP’09. International Conference on.* IEEE. 2009, pp. 462–469 (cited on page 44).

[16] M. G. Burke, K. Knobe, R. Newton, and V. Sarkar. *The Concurrent Collections Programming Model.* Tech. rep. Rice University Houston, 2010 (cited on page 67).

[17] A.D. Carlson, V.G. Pronyaev, D.L. Smith, N.M. Larson, Zhenpeng Chen, G.M. Hale, F.-J. Hambsch, E.V. Gai, Soo-Youl Oh, S.A. Badikov, et al. “International Evaluation of Neutron Cross Section Standards”. In: *Nuclear Data Sheets* 110.12 (2009). Special Issue on Nuclear Reaction Data, pp. 3215–3324. ISSN: 0090-3752. DOI: `http://dx.doi.org/10.1016/j.nds.2009.11.001` (cited on page 11).

[18] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. “Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms”. In: *Journal of Parallel and Distributed Computing* 74.10 (June 2014), pp. 2899–2917 (cited on page 50).

[19] Jatin Chhugani, Anthony D Nguyen, Victor W Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. “Efficient implementation of sorting on multi-core SIMD CPU architecture”. In: *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1313–1324 (cited on page 22).

[20] Andrea Clematis and Angelo Corana. “Modeling performance of heterogeneous parallel computing systems”. In: *Parallel Computing* 25.9 (1999), pp. 1131–1145 (cited on page 49).

[21] Guillem Colomer, Rick Borrell, FX Trias, and I Rodrıguez. “Parallel algorithms for SN transport sweeps on unstructured meshes”. In: *Journal of Computational Physics* 232.1 (2013), pp. 118–135 (cited on page 43).

[22] Michel Cosnard, Emmanuel Jeannot, and Tao Yang. “Compact DAG Representation and its Symbolic Scheduling”. In: *Journal of Parallel and Distributed Computing* 64.8 (2004), pp. 921–935 (cited on page 67).

[23] F Coulomb and C Fedon-Magnaud. “Mixed and mixed-hybrid elements for the diffusion equation”. In: *Nuclear Science and Engineering* 100.3 (1988), pp. 218–225 (cited on page 87).

[24] T. Courau. *Specifications of a 3D PWR Core Benchmark for Neutron Transport.* Tech. rep. Technical Note CR-128/2009/014 EDF-SA, 2009 (cited on page 93).

[25] T. Courau, S. Moustafa, L. Plagne, and A. Ponçot. “DOMINO: A Fast 3D Cartesian Discrete Ordiantes Solver for Reference PWR Simulations and SPN Validations”. In: *International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering (M&C 2013)*. USA, May 2013 (cited on page 106).

[26] T. Courau and G. Sjoden. “3D Neutron Transport and HPC: A PWR Full Core Calculation Using PENTRAN SN Code and IBM BLUEGENE/P Computers”. In: *Progress in Nuclear Science and Technology* 2 (2011), pp. 628–633 (cited on page 42).

[27] Leonardo Dagum and Ramesh Menon. “OpenMP: An Industry-Standard API for Shared-Memory Programming”. In: *IEEE Comput. Sci. Eng.* 5.1 (Jan. 1998), pp. 46–55. ISSN: 1070-9924. DOI: `10.1109/99.660313` (cited on page 66).

[28] Gregory G Davidson, Thomas M Evans, Joshua J Jarrell, and Rachel N Slaybaugh. “Massively Parallel, Three-Dimensional Transport Solutions for the k-Eigenvalue Problem”. In: *International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering (M&C 2011)*. Brazil, May 2011 (cited on pages 2, 3, 11, 42, 102).

[29] Robert H Dennard, VL Rideout, E Bassous, and AR Leblanc. “Design of ion-implanted MOSFET’s with very small physical dimensions”. In: *Solid-State Circuits, IEEE Journal of* 9.5 (1974), pp. 256–268 (cited on page 17).

[30] James J Duderstadt and Louis J Hamilton. “Nuclear reactor analysis”. In: (1976) (cited on page 6).

[31] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. “Dark silicon and the end of multicore scaling”. In: *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE. 2011, pp. 365–376 (cited on page 17).

[32] Roger Espasa, Mateo Valero, and James E Smith. “Vector architectures: past, present and future”. In: *Proceedings of the 12th international conference on Supercomputing*. ACM. 1998, pp. 425–432 (cited on page 22).

[33] Pierre Estérie, Joel Falcou, Mathias Gaunard, and Jean-Thierry Lapresté. “Boost. simd: generic programming for portable simdization”. In: *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*. ACM. 2014, pp. 1–8 (cited on page 25).

[34] Thomas M Evans, Gregory G Davidson, Rachel N Slaybaugh, and K Clarno. “Three-dimensional full core power calculations for pressurized water reactors”. In: *Journal of Physics: Conference Series, SciDAC*. Vol. 68. 2010, pp. 367–379 (cited on page 3).

[35] Thomas M Evans, Wayne Joubert, Steven P Hamilton, Seth R Johnson, John A Turner, Gregory G Davidson, and Tara M Pandya. *Three-dimensional discrete ordinates reactor assembly calculations on GPUs*. Tech. rep. Oak Ridge National Laboratory (ORNL); Oak Ridge Leadership Computing Facility (OLCF); Consortium for Advanced Simulation of LWRs (CASL), 2015 (cited on page 106).

[36] Thomas M Evans, Alissa S Stafford, Rachel N Slaybaugh, and Kevin T Clarno. “Denovo: A new three-dimensional parallel discrete ordinates code in SCALE”. In: *Nuclear technology* 171.2 (2010), pp. 171–200 (cited on page 96).

[37] F Févotte. “PDSA: a Piecewise Diffusion Synthetic Acceleration Scheme”. In preparation for submission to Journal of Computational Physics (cited on pages 88, 97, 98).

[38] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. "Intel AVX: New frontiers in performance improvements and energy efficiency". In: *Intel white paper* (2008) (cited on page 22).

[39] James W Fischer and YY Azmy. "Comparison via parallel performance models of angular and spatial domain decompositions for solving neutral particle transport problems". In: *Progress in Nuclear Energy* 49.1 (2007), pp. 37–60 (cited on pages 3, 43).

[40] Michael J Flynn. "Some computer organizations and their effectiveness". In: *Computers, IEEE Transactions on* 100.9 (1972), pp. 948–960 (cited on page 22).

[41] John Fruehe. "Multicore processor technology". In: *Reprinted from Dell Power Solutions www. dell. com/powersolutions (Obtained from the Internet on Mar. 23, 2012)* (2005), pp. 67–72 (cited on page 2).

[42] G. Marleau, A. Hébert and R. Roy. *A User's Guide for DRAGON 3.05*. Tech. rep. IGE-174 Rev.6. Institut de Génie Nucléaire, École Polytechnique de Montréal, 2006 (cited on page 93).

[43] E.M. Gelbard. *Simplified spherical harmonics equations and their use in shielding problems*. Tech. rep. WAPD-T11-1182. Westinghouse Report, 1961 (cited on page 86).

[44] Chunye Gong, Jie Liu, Lihua Chi, Haowei Huang, Jingyue Fang, and Zhenghu Gong. "GPU Accelerated Simulations of 3D Deterministic Particle Transport Using Discrete Ordinates Method". In: *J. Comput. Phys.* 230.15 (July 2011), pp. 6010–6022. ISSN: 0021-9991. DOI: `10.1016/j.jcp.2011.04.010` (cited on page 106).

[45] Richard L Graham, Galen M Shipman, Brian W Barrett, Ralph H Castain, George Bosilca, and Andrew Lumsdaine. "Open MPI: A high-performance, heterogeneous MPI". In: *Cluster Computing, 2006 IEEE International Conference on*. IEEE. 2006, pp. 1–9 (cited on page 44).

[46] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. "A high-performance, portable implementation of the MPI message passing interface standard". In: *Parallel computing* 22.6 (1996), pp. 789–828 (cited on page 42).

[47] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010 (cited on pages 25, 33, 68).

[48] Linley Gwennap. "Sandy Bridge spans generations". In: *Microprocessor Report* 9.27 (2010), pp. 10–01 (cited on page 23).

[49] A. Hébert. *Applied Reactor Physics*. Presses internationales Polytechnique, 2009. ISBN: 9782553014369 (cited on page 11).

[50] Alain Hébert. "High order diamond differencing schemes". In: *Annals of Nuclear Energy* 33.17–18 (2006), pp. 1479–1488. ISSN: 0306-4549. DOI: `10.1016/j.anucene.2006.10.003` (cited on pages 13, 87).

[51] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J Ramanujam, and P Sadayappan. "Data layout transformation for stencil computations on short-vector SIMD architectures". In: *Compiler Construction*. Springer. 2011, pp. 225–245 (cited on page 22).

[52] Michael A Heroux, Roscoe A Bartlett, Vicki E Howle, Robert J Hoekstra, Jonathan J Hu, Tamara G Kolda, Richard B Lehoucq, Kevin R Long, Roger P Pawlowski, Eric T Phipps, et al. "An overview of the Trilinos project". In: *ACM Transactions on Mathematical Software (TOMS)* 31.3 (2005), pp. 397–423 (cited on page 3).

[53] Mike Heroux, Rob Neely, and Sriram Swaminarayan. *ASC co-design proxy app strategy*. Tech. rep. Technical Report LA-UR-13 (cited on page 3).

[54] Roger W. Hockney. "The communication challenge for MPP: Intel Paragon and Meiko CS-2". In: *Parallel Computing* 20.3 (1994), pp. 389–398. ISSN: 0167-8191. DOI: `http://dx.doi.org/10.1016/S0167-8191(06)80021-9` (cited on page 49).

[55] Adolfy Hoisie, Olaf Lubeck, and Harvey Wasserman. "Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications". In: *International Journal of High Performance Computing Applications* 14.4 (2000), pp. 330–346 (cited on pages 3, 43).

[56] Philippe Humbert. "Parallelization of PANDA discrete ordinates code using spatial decomposition". In: *Proceedings of PHYSOR-2006, ANS Topical Meeting on Reactor Physics*. 2006, pp. 10–14 (cited on page 3).

[57] E Jamelot, J Dubois, JJ Lautard, C Calvin, and AM Baudron. "High performance 3D neutron transport on petascale and hybrid architectures within APOLLO3 code". In: *International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering*. Vol. 18. 2011 (cited on page 106).

[58] Erell Jamelot and Patrick Ciarlet Jr. "Fast non-overlapping Schwarz domain decomposition methods for solving the neutron diffusion equation". In: *Journal of Computational Physics* 241 (2013), pp. 445–463. ISSN: 0021-9991. DOI: `http://dx.doi.org/10.1016/j.jcp.2013.01.026` (cited on pages 88, 106).

[59] Sverre Jarp, Alfio Lazzaro, A Nowak, and J Leduc. *Evaluation of the Intel Sandy Bridge-EP server processor*. Tech. rep. 2012 (cited on page 36).

[60] Adam Jundt, Ananta Tiwari, William A Ward Jr, Roy Campbell, and Laura Carrington. "Optimizing codes on the Xeon Phi: a case-study with LAMMPS". In: *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*. ACM. 2015, p. 28 (cited on page 3).

[61] L. V. Kalé and S. Krishnan. "CHARM++: A Portable Concurrent Object Oriented System Based On C++". In: *OOPSLA*. 1993, pp. 91–108 (cited on page 67).

[62] Darshan Kaushik, A Wollaber, Brian Smith, Avivas Siegel, Won Sik Yang, et al. "Enabling high-fidelity neutron transport simulations on petascale architectures". In: *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*. IEEE. 2009, pp. 1–12 (cited on page 2).

[63] Dinesh Kaushik, Micheal Smith, Allan Wollaber, Barry Smith, Andrew Siegel, and Won Sik Yang. "Enabling high-fidelity neutron transport simulations on petascale architectures". In: *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*. IEEE. 2009, pp. 1–12 (cited on pages 3, 42).

[64] Darren J Kerbyson, Adolfy Hoisie, and Shawn D Pautz. "Performance modeling of deterministic transport computations". In: *Performance Analysis and Grid Computing*. Springer, 2004, pp. 21–39 (cited on pages 3, 43).

[65] Wilfried Kirschenmann. "Towards sustainable intensive computing kernels". Theses. Université de Lorraine, Oct. 2012 (cited on page 25).

[66] Keisuke Kobayashi, Naoki Sugimura, and Yasunobu Nagaya. *3-D radiation transport benchmark problems and results for simple geometries with void regions*. Nuclear Energy Agency, 2000 (cited on pages 91, 92, 95, 96).

[67] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R Bishop. "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units". In: *SIAM Journal on Scientific Computing* 36.5 (2014), pp. C401–C423 (cited on page 22).

[68] AJ Kunen, TS Bailey, and PN Brown. "KRIPKE-A MASSIVELY PARALLEL TRANSPORT MINI-APP". In: (2015) (cited on page 4).

[69] Edward W Larsen and Jim E Morel. "Advances in discrete-ordinates methodology". In: *Nuclear Computational Science*. Springer, 2010, pp. 1–84 (cited on pages 86, 87).

[70] E.W. Larsen. "Unconditionally stable diffusion synthetic acceleration methods for the slab geometry discrete ordinates equations". In: *Nuclear Science and Engineering* (1982), pp. 47–63 (cited on page 87).

[71] Alexander R Larzelere et al. "Creating simulation capabilities". In: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 27–35 (cited on page 42).

[72] Kaye D Lathrop. "Ray effects in discrete ordinates equations". In: *Nuclear Science and Engineering* 32.3 (1968), pp. 357–369 (cited on page 13).

[73] JJ Lautard, D Schneider, and A Baudron. "Mixed dual methods for neutronic reactor core calculations in the CRONOS system". In: *Proc. Int. Conf. Mathematics and Computation, Reactor Physics and Environmental Analysis of Nuclear Systems*. 1999, pp. 27–30 (cited on page 86).

[74] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU". In: *ACM SIGARCH Computer Architecture News*. Vol. 38. 3. ACM. 2010, pp. 451–460 (cited on page 22).

[75] Randall J LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*. Vol. 98. Siam, 2007 (cited on page 12).

[76] E. E. Lewis and W. F. Miller, Jr. *Computational Methods of Neutron Transport*. New York: John Wiley and Sons, Inc, 1984 (cited on pages 6, 9–11).

[77] Saeed Maleki, Yaoqing Gao, Mara J Garzaran, Tommy Wong, David Padua, et al. "An evaluation of vectorizing compilers". In: *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE. 2011, pp. 372–382 (cited on page 25).

[78] Maxime Martinasso and Jean-François Méhaut. "A Contention-Aware Performance Model for HPC-Based Networks: A Case Study of the InfiniBand Network". English. In: *Euro-Par 2011 Parallel Processing*. Ed. by Emmanuel Jeannot, Raymond Namyst, and Jean Roman. Vol. 6852. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 91–102. ISBN: 978-3-642-23399-9. DOI: `10.1007/978-3-642-23400-2_10` (cited on page 49).

[79] John D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Tech. rep. A continually updated technical report. Charlottesville, Virginia: University of Virginia, 1991-2007 (cited on page 24).

[80] John D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), pp. 19–25 (cited on page 24).

[81] Sally A McKee. “Reflections on the memory wall”. In: *Proceedings of the 1st conference on Computing frontiers*. ACM. 2004, p. 162 (cited on page 2).

[82] Maxime Martinasso—Jean-François Méhaut. “Model of concurrent MPI communications over SMP clusters”. In: (2006) (cited on page 49).

[83] J Mielikainen, B Huang, and AH-L Huang. “Intel Xeon Phi accelerated Weather Research and Forecasting (WRF) Goddard microphysics scheme”. In: *Geoscientific Model Development Discussions* 7.6 (2014), pp. 8941–8973 (cited on page 3).

[84] Nathalie Möller, Eric Petit, Loıc Thébault, and Quang Dinh. “A Case Study on Using a Proto-Application as a Proxy for Code Modernization”. In: *Procedia Computer Science* 51 (2015), pp. 1433–1442 (cited on page 3).

[85] Sally Falk Moore. “Law and social change: the semi-autonomous social field as an appropriate subject of study”. In: *Law and Society Review* (1973), pp. 719–746 (cited on page 17).

[86] Said F Mughabghab. *Neutron Cross Sections: Neutron Resonance Parameters and Thermal Cross Sections Part B: Z= 61-100*. Vol. 1. Academic press, 2012 (cited on page 11).

[87] Olga Mula. “Some contributions towards the parallel simulation of time dependent neutron transport and the integration of observed data in real time”. PhD thesis. Université Pierre et Marie Curie-Paris VI, 2014 (cited on page 107).

[88] Onur Mutlu. “Memory scaling: A systems architecture perspective”. In: *Memory Workshop (IMW), 2013 5th IEEE International*. IEEE. 2013, pp. 21–25 (cited on page 44).

[89] Jean-Claude Nédélec. “A new family of mixed finite elements in R3”. In: *Numerische Mathematik* 50.1 (1986), pp. 57–81 (cited on page 87).

[90] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. “Scalable Parallel Programming with CUDA”. In: *Queue* 6.2 (Mar. 2008), pp. 40–53. ISSN: 1542-7730. DOI: `10.1145/1365490.1365500` (cited on page 66).

[91] Matt Pharr and William R Mark. “ispc: A SPMD compiler for high-performance CPU programming”. In: *Innovative Parallel Computing (InPar), 2012*. IEEE. 2012, pp. 1–13 (cited on page 25).

[92] Chuck Pheatt. “Intel® threading building blocks”. In: *Journal of Computing Sciences in Colleges* 23.4 (2008), pp. 298–298 (cited on page 26).

[93] Katia Pinchedez. “Calcul parallèle pour les équations de diffusion et de transport homogènes en neutronique”. PhD thesis. PhD thesis, Université de Paris 6, 1999 (cited on page 106).

[94] Laurent Plagne and Angélique Ponçot. “Generic Programming for Deterministic Neutron Transport Codes”. In: *Mathematics & Computation, Supercomputing, Reactor Physics and Nuclear and Biological Applications*. Palais des Papes, Avignon, France, Sept. 2005 (cited on page 87).

[95] Steven J Plimpton, Bruce Hendrickson, Shawn P Burns, William McLendon, and Lawrence Rauchwerger. “Parallel Sn sweeps on unstructured grids: Algorithms for prioritization, grid partitioning, and cycle detection”. In: *Nuclear science and engineering* 150.3 (2005), pp. 267–283 (cited on page 43).

[96] Angélique Ponçot and Laurent Plagne. *Note de principe du solveur SN DOMINO*. Tech. rep. H-I23-2013-00799-FR. EDF R&D, 2014 (cited on page 96).

[97] Hari Radhakrishnan, Damian WI Rouson, Karla Morris, Sameer Shende, and Stavros C Kassinos. “Using Coarrays to Parallelize Legacy Fortran Applications: Strategy and Case Study”. In: *Scientific Programming* 501 (2015), p. 904983 (cited on page 3).

[98] Pierre-Arnaud Raviart and Jean-Marie Thomas. “A mixed finite element method for 2-nd order elliptic problems”. In: *Mathematical aspects of finite element methods*. Springer, 1977, pp. 292–315 (cited on page 87).

[99] James Reinders. *Intel Threading Building Blocks*. First. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 2007. ISBN: 9780596514808 (cited on page 66).

[100] James Reinders and James Jeffers. *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*. Morgan Kaufmann, 2014 (cited on page 3).

[101] Paul Reuss. *Précis de neutronique*. INSTN, 2003 (cited on page 1).

[102] A. Robison, M. Voss, and A. Kukanov. “Optimization via reflection on work stealing in TBB”. In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE. 2008, pp. 1–8 (cited on page 67).

[103] Arch D Robison. “Cilk plus: Language support for thread and vector parallelism”. In: *Talk at HP-CAST* 18 (2012) (cited on page 25).

[104] Youcef Saad. “Chebyshev acceleration techniques for solving nonsymmetric eigenvalue problems”. In: *Mathematics of Computation* 42.166 (1984), pp. 567–588 (cited on page 10).

[105] Francesc Salvat, José M Fernández-Varea, and Josep Sempau. “PENELOPE-2006: A code system for Monte Carlo simulation of electron and photon transport”. In: *Workshop Proceedings*. Vol. 4. 2006, p. 7 (cited on page 2).

[106] Richard Sanchez. “Prospects in deterministic three-dimensional whole-core transport calculations”. In: *Nuclear Engineering and Technology* 44.2 (2012), pp. 113–150 (cited on pages 1, 2).

[107] Brian K. Schmidt and Vaidy S. Sunderam. “Empirical analysis of overheads in cluster environments”. In: *Concurrency: Practice and Experience* 6.1 (1994), pp. 1–32. ISSN: 1096-9128. DOI: `10.1002/cpe.4330060102` (cited on page 49).

[108] Frédérique Silber-Chaussumier. “Recouvrement des communications et des calculs, du matériel au logiciel”. PhD thesis. Lyon, Ecole normale supérieure, 2002 (cited on pages 3, 43).

[109] Quinn O Snell, Armin R Mikler, and John L Gustafson. “Netpipe: A network protocol independent performance evaluator”. In: *IASTED International Conference on Intelligent Information Management and Systems*. Vol. 6. Washington, DC, USA. 1996 (cited on pages 49, 55).

[110] John E. Stone, David Gohara, and Guochun Shi. “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”. In: *IEEE Des. Test* 12.3 (May 2010), pp. 66–73. ISSN: 0740-7475. DOI: `10.1109/MCSE.2010.69` (cited on page 66).

[111] Toshikazu Takeda and Hideaki Ikeda. “3-D neutron transport benchmarks”. In: *Journal of Nuclear Science and Technology* 28.7 (1991), pp. 656–669 (cited on pages 91, 92, 96, 97).

[112] Peiyi Tang. “Measuring the overhead of Intel C++ Concurrent Collections over Threading Building Blocks for Gauss–Jordan elimination”. In: *Concurrency and Computation: Practice and Experience* 24.18 (2012), pp. 2282–2301 (cited on page 67).

[113] Robin AJ Taylor, Jaehak Jeong, Michael White, and Jeffrey G Arnold. “Code modernization and modularization of APEX and SWAT watershed simulation models”. In: *International Journal of Agricultural and Biological Engineering* 8.3 (2015) (cited on page 3).

[114] A Vladimirov. “Arithmetics on Intel’s Sandy Bridge and Westmere CPUs: not all FLOPS are created equal”. In: *Colfax International* (2012) (cited on pages 23, 36).

[115] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. “Intel Math Kernel Library”. In: *High-Performance Computing on the Intel® Xeon Phi*^{TM}. Springer, 2014, pp. 167–188 (cited on page 25).

[116] James S Warsa, Todd A Wareing, Jim E Morel, John M McGhee, and Richard B Lehoucq. “Krylov subspace iterations for deterministic k-eigenvalue calculations”. In: *Nuclear Science and Engineering* 147.1 (2004), pp. 26–42 (cited on page 2).

[117] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: an insightful visual performance model for multicore architectures”. In: *Communications of the ACM* 52.4 (2009), pp. 65–76 (cited on pages 35, 36).

[118] Akio Yamamoto. “A quantitative comparison of loading pattern optimization methods for in-core fuel management of PWR”. In: *Journal of nuclear science and technology* 34.4 (1997), pp. 339–347 (cited on page 2).

[119] Jie Yan, Guang-Ming Tan, and Ning-Hui Sun. “Optimizing parallel Sn sweeps on unstructured grids for multi-core clusters”. In: *Journal of Computer Science and Technology* 28.4 (2013), pp. 657–670 (cited on page 44).

[120] A. Yarkhan. “Dynamic Task Execution on Shared and Distributed Memory Architectures”. PhD thesis. University of Tennessee, 2012 (cited on page 67).

[121] Musa Yavuz and Edward W Larsen. “Iterative Methods for Solving x-y Geometry SN Problems on Parallel Architecture Computers”. In: *Nuclear science and engineering* 112.1 (1992), pp. 32–42 (cited on page 88).