		
	Note Technique DES	Page 1/116

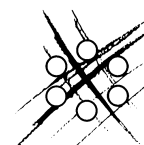
DIRECTION DES ENERGIES
INSTITUT DES SCIENCES APPLIQUEES ET DE LA
SIMULATION POUR LES ENERGIES BAS CARBONNE
DEPARTEMENT DE MODELISATION DES SYSTEMES ET STRUCTURES
SERVICE DE THERMOHYDRAULIQUE ET DE MECANIQUE DES FLUIDES

LBM_saclay : code HPC multi-architectures sur base LBM. Guide du
développeur.

Werner Verdier
Téo Boutin
Pierre Kestener
Alain Cartalade


DES/ISAS/DM2S/STMF/LMSF/NT/2022-70869/A

Commissariat à l'énergie atomique et aux énergies alternatives
 Centre de Saclay - DES/ISAS/DM2S/STMF/LMSF - BAT 454 - PC 47
 Tél. : 33 - 1 69 08 91 10 Fax : 33 - 1 69 08 96 96
 Courriel : katy.colafrancesco@cea.fr
 Établissement public à caractère industriel et commercial
 RCS PARIS B 775 685 019



Réf du formulaire : F1-DM2S/DIR/PR/003 classe L^AT_EX docDM2S version C

Document propriété du CEA - Reproduction et diffusion externe au CEA soumises à l'autorisation de l'émetteur

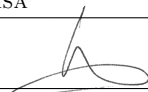

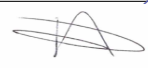
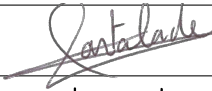

		Note Technique DES	Page 2/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		


NIVEAU DE CONFIDENTIALITÉ				
DO	DR	CCEA	CD	SD
X				

PARTENAIRES/CLIENTS	ACCORD	TYPE D'ACTION

RÉFÉRENCES INTERNES CEA			
DIRECTION D'OBJECTIFS	DOMAINE	PROJET	EOTP
DPE	SIMU	PICI2	A-PICI2-03-01-01
JALON	INTITULÉ DU JALON	DÉLAI CONTRACTUEL DE CONFIDENTIALITÉ	CAHIERS DE LABORATOIRE

SUIVI DES VERSIONS			
INDICE	DATE	NATURE DE L'ÉVOLUTION	PAGES & CHAPITRES MODIFIÉS
A	21/11/2022	Document initial	

	NOM	FONCTION	VISA	DATE
RÉDACTEUR	WERNER VERDIER	Doctorant		21/11/2022
RÉDACTEUR	TÉO BOUTIN	Doctorant		24/11/2022
RÉDACTEUR	PIERRE KESTENER	Ingénieur-chercheur		24/11/2022
RÉDACTEUR	ALAIN CARTALADE	Ingénieur-chercheur		21/11/2022
VÉRIFICATEUR	ALAIN GENTY	Ingénieur-chercheur		24/11/2022
VÉRIFICATEUR	RÉMI BARON	Ingénieur-chercheur, responsable de lot PICI2/HPC		
APPROBATEUR	NICOLAS DORVILLE	Chef de Service		
ÉMETTEUR	NICOLAS DORVILLE	Chef de Service		

		Note Technique DES	Page 3/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Mots Clefs


Lattice Boltzmann Methods, modèles à champ de phase, portabilité, HPC, GPUs, LBM_saclay, Kokkos

RESUME/CONCLUSIONS

LBM_saclay est une application écrite en C++ et initialement co-développée par deux directions du CEA : la DES et la DRF. Elle est destinée aux "calculs haute performance" (High Performance Computing – HPC) et ses noyaux de calculs reposent sur les "méthodes de Boltzmann sur réseaux" (Lattice Boltzmann Methods – LBM). La LBM est une méthode numérique alternative aux méthodes traditionnelles (telles que les Éléments Finis ou les Volumes Finis) qui permet de simuler très simplement une grande variété d'Equations aux Dérivées Partielles (EDPs) qui interviennent dans plusieurs domaines de la physique. Son principal domaine d'application est celui de la physique des fluides (monophasiques turbulents, diphasiques, etc ...) mais elle trouve aussi des applications en sciences des matériaux et en électro-magnétisme.

Grâce à la simplicité de ses deux étapes de collision et de déplacement, la méthode est très bien adaptée au HPC. Le principal atout du code LBM_saclay est qu'il peut être compilé et exécuté sur plusieurs plateformes HPC, telles que les architectures classiques multi-CPU (pour "Central Processing Units"), mais aussi les architectures multi-GPU (pour "Graphics Processing Units"). La portabilité du code sur les différentes architectures est assurée par l'utilisation de la bibliothèque Kokkos et l'exécutable est adapté à chaque architecture par le passage d'options au niveau de la configuration avec CMake. Deux niveaux de parallélisme sont présents dans LBM_saclay : le parallélisme à mémoire partagée, géré par Kokkos qui assure l'interface entre CUDA, OpenMP et pthreads, et le parallélisme à mémoire distribuée (décomposition de domaine), géré par MPI.

Cette documentation s'adresse aux étudiants ou aux ingénieurs qui souhaitent s'initier à la LBM et au HPC en réalisant des développements dans LBM_saclay. Pour cela, après un premier chapitre qui décrit l'installation du code sur plusieurs architectures, le document est séparé en deux parties. La première décrit en quatre chapitres les bases de LBM_saclay. La seconde partie est composée de deux tutoriels qui décrivent pas à pas les modifications à apporter au code source pour simuler un problème de décomposition spinodale et un autre de croissance cristalline. Ces deux tutoriels ont été refaits avec succès par plusieurs étudiants venus se former à la méthode et au code. Enfin, deux annexes détaillent quelques développements mathématiques relatifs à la LBM et viennent clore cette documentation. LBM_saclay est en constante évolution et cette documentation sera mise à jour et complétée petit à petit.

		Note Technique DES	Page 4/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

DIFFUSION INITIALE

(Diffusion par email si document en DO)

Diffusion interne CEA

DES/ISAS	Stéphane Gossé
DES/EC/DPE	Xavier Raepsaet
	Thomas Laporte
	François Sudreau
	Cécile Ferry
DES/ISAS/DM2S	Patrick Blanc-Tranchant
	Daniel Caruge
	Frédéric Damian
DES/ISAS/DM2S/STMF	Mathieu Niezgoda
	Nicolas Dorville
	Marion Le Flem
	Philippe Fillion
DES/ISAS/DM2S/STMF/LMSF	tout le laboratoire
DIF/DSSI/SANL/LANS	Pierre Kestener
DES/ISAS/DM2S/STMF/LATF	Aurélien Davailles
	Alain Genty
	Bruno Raverdy
	Sonia Benteboula
DES/ISAS/DM2S/STMF/LGLS	Erwan Adam
	Pierre Ledac
	Fabrice Gaudier
DES/ISAS/DM2S/STMF/LMES	Anouar Mekkas
	Marc Tajchman
DES/ISAS/DM2S/STMF/LIEFT	Gilles Bernard-Michel
DES/ISAS/DM2S/STMF/LMEC	Didier Schneider
	Samuel Kokh
	Christophe Le Potier
DES/ISAS/DM2S/SERMA/LLPR	Rémi Baron
DES/ISEC/DE2D/DIR	Sophie Schuller
DES/ISEC/DE2D/SEVT/LCLT	Frédéric Angeli
	Jean-Marc Delaye
DES/ISEC/DMRC/DIR	Sophie Charton
DES/ISEC/DMRC/SASP/LSPS	Emeric Brun
	Tojonirina Randriamanantena
DES/IRESNE/DTN	Vincent Faucher
DES/IRESNE/DTN/SMTA/LMAG	Laurent Saas
	Romain Le Tellier

Diffusion externe

"sans objet"

Diffusion résumé

DES/ISAS/DM2S/DIR
DES/ISAS/DM2S/STMF/DIR
DES/ISAS/DM2S/STMF/LMSF




		Note Technique DES	Page 5/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Table des matières


1	Installation de LBM_saclay	13
1.1	Récupération du code source	13
1.1.1	Par mail	13
1.1.2	Par le GitLab de la MDLS du CEA	13
1.2	Description des sous-répertoires	14
1.3	Compilation et exécution d'un cas test sur CPUs	15
1.3.1	Version CPUs en OpenMP	15
1.3.2	Version CPUs en OpenMP + MPI	16
1.4	Compilation et exécution d'un cas test sur GPUs	17
1.4.1	Version GPUs en CUDA	17
1.4.2	Version GPUs en CUDA + MPI	18
1.5	Description d'un jeu de données : fichiers .ini	18
I	Les bases de LBM_saclay	21
2	Les nouveaux types dans LBM_saclay	23
2.1	Rappels de la LBM « standard »	23
2.1.1	La fonction de distribution f_i	23
2.1.2	Définition des réseaux	24
2.1.3	Description de l'algorithme et liens avec les équations macroscopiques	24
2.2	Définitions des nouveaux types C++ dans LBM_saclay	26
2.2.1	Liste des nouveaux types	27
2.2.2	Exemples d'utilisation : le noyau de calcul CollideAndStream_NS_STD_Functor.h	28
3	Organisation de LBM_saclay	31
3.1	Le fichier pilote « LBMRun.h »	31
3.1.1	Gestion des noyaux de calcul dans LBMRun.h : insertion des fichiers de src/kernels	31
3.1.2	Allocation mémoire des tableaux dans LBMRun.h : la fonction LBMRun	32
3.1.3	Gestion des conditions initiales dans LBMRun.h	33
3.1.4	Itération en temps dans LBMRun.h : les fonctions run et update	34
3.1.5	Gestion des entrées-sorties dans LBMRun.h : la classe configMap	36
3.1.6	Gestion du parallélisme MPI dans LBMRun.h	36
3.2	Les instructions relatives à la bibliothèque Kokkos	37
3.2.1	La « décoration » KOKKOS_INLINE_FUNCTION	37
3.2.2	La méthode statique apply	37
3.2.3	Les fonctions void operator()	38
3.2.4	Les tags pour différencier les fonctions void operator()	38

		Note Technique DES	Page 6/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		


3.3	Autres fonctions	39
3.3.1	Les énumérations : fichier <code>LBM_enums.h</code>	39
3.3.2	Gestion des champs macroscopiques : fichier <code>FieldManager.h</code>	40
4	Description des principaux noyaux de calculs	43
4.1	Introduction	43
4.2	Équation du transport par advection-diffusion	44
4.2.1	Rappel de l'équation continue	44
4.2.2	Algorithme LBM pour l'ADE avec $S_C = 0$	44
4.2.3	Algorithme LBM pour l'ADE avec $S_C \neq 0$	45
4.3	Équation du champ de phase	46
4.3.1	Rappel de l'équation continue	46
4.3.2	Algorithme LBM associé	46
4.3.3	Fichier <code>CollideAndStream_PhaseField_Functor.h</code>	47
4.3.3.1	Calcul du vecteur unitaire normal à l'interface \mathbf{n}	47
4.3.3.2	Utilisation dans la fonction à l'équilibre	48
4.4	Modèle de Navier-Stokes/Conservative Allen-Cahn	49
4.4.1	Rappel du modèle NS en « compressibilité artificielle »	49
4.4.2	Algorithme LBM avec prise en compte d'un terme force	50
4.4.3	Fichier <code>CollideAndStream_NS_CAC_Functor.h</code>	51
5	Parallélisme à mémoire distribuée et super-calculateurs	55
5.1	Décomposition de domaines	55
5.1.1	Informations dans le jeu de données et IO en HDF5	55
5.1.2	Communications MPI et cellules fantômes	56
5.2	Tests sur les super-calculateurs	58
5.2.1	Nœuds GPUs d'ORCUS	58
5.2.2	Calculateur JEAN-ZAY	60
5.2.3	Partition Topaze-A100	64
5.2.4	Perspectives : tests sur d'autres super-calculateurs	65
II	Développer son modèle dans LBM_saclay	67
6	Tutoriel sur l'équation de Cahn-Hilliard	71
6.1	Équation de Cahn-Hilliard	71
6.1.1	Rappel de l'équation continue	71
6.1.2	Rappel de l'algorithme LBM pour le Cahn-Hilliard	72
6.1.3	Quelques remarques	73
6.2	Mise en œuvre dans LBM_saclay	74
6.2.1	Nouveau noyau <code>CollideAndStream_CahnHilliard_Functor.h</code>	74
6.2.1.1	Nouvelle structure de paramètres <code>CH_params</code>	74
6.2.1.2	Pré-requis pour le calcul du potentiel chimique	75
6.2.1.3	Nouvelle fonction <code>compute_chemical_potential</code>	76
6.2.1.4	Modification de l'équilibre	77
6.2.2	Modifications à apporter dans <code>LBMRun.h</code>	78
6.2.2.1	Inclure le nouveau noyau de calcul	78
6.2.2.2	Faire l'appel de la nouvelle fonction	78
6.2.3	Modifications dans <code>LBM_enums.h</code> et <code>FieldManager.h</code>	79
6.2.4	Conditions initiales	80

		Note Technique DES	Page 7/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

6.2.4.1	La condition initiale <code>InitPhi_Shrf_Vortex.h</code>	80
6.2.4.2	Nouvelle condition initiale <code>InitPhi_Unif_Mixture.h</code>	80
6.3	Tests	82
6.3.1	Vérification des développements du Cahn-Hilliard	82
6.3.2	Simulation avec la condition initiale aléatoire	82
7	Tutoriel sur un modèle de croissance cristalline	85
7.1	Modèle à champ de phase pour la croissance cristalline	85
7.1.1	Rappel du modèle continu	85
7.1.2	Rappel de l'algorithme LBM	86
7.1.3	Quelques remarques	87
7.2	Mise en œuvre dans <code>LBM_saclay</code>	88
7.2.1	Nouveau noyau <code>CollideAndStream_KarmaRappel_Phi_Functor.h</code>	88
7.2.1.1	Nouvelle structure de paramètres <code>KR_params</code>	88
7.2.1.2	Dans la fonction <code>compute_gradient_phi</code>	89
7.2.1.3	Nouvelle fonction <code>compute_anisotropy_function_As_STD</code>	89
7.2.1.4	Nouvelle fonction <code>compute_anisotropy_vector_N</code>	90
7.2.1.5	Les fonctions <code>void operator()</code>	90
7.2.2	Nouveau noyau <code>CollideAndStream_KarmaRappel_Tpr_Functor.h</code>	92
7.2.3	Ajouts dans le fichier <code>LBMRun.h</code>	92
7.2.3.1	Ajouts pour le noyau « champ de phase »	92
7.2.3.2	Ajouts pour le noyau « température »	93
7.2.4	Ajouts dans les fichiers <code>LBMPParams.cpp</code> , <code>LBMPParams.h</code> et <code>FieldManager.h</code>	94
7.2.5	Conditions initiales	95
7.2.5.1	Condition initiale pour ϕ : fichier <code>InitPhi_Shrf_Vortex.h</code>	95
7.2.5.2	Condition initiale pour θ : nouveau fichier <code>UniformValue.h</code>	96
7.3	Tests	96
III	Conclusion – Perspectives	99
IV	Annexes	103
A	Développements de Chapman-Enskog pour les équations de Navier-Stokes	105
A.1	Introduction	105
A.1.1	Rappels de la méthode	105
A.1.2	Objectifs	106
A.2	Grandeurs conservées et résultats préliminaires sur la fonction à l'équilibre	106
A.3	Développement de Chapman-Enskog	107
A.3.1	Première partie : développement de Taylor et séparation des échelles	107
A.3.1.1	Moments des termes en ε	108
A.3.1.2	Moments des termes en ε^2	109
A.3.1.3	Regroupement des moments	109
A.3.2	Seconde partie : coefficient de viscosité et équation de conservation de la quantité de mouvement	109
A.3.2.1	Viscosité	109
A.3.2.2	Equation de conservation de la quantité de mouvement	111
A.4	Récapitulatif	111
A.5	Démonstration de l'égalité (A.27)	111

		Note Technique DES	Page 8/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

B	Changement de variable dans la LBE pour la prise en compte d'une source	113
B.1	Equivalence de l'Eq. 6.25 de [1] et de l'Eq. B.11	114
B.1.1	Démonstrations de l'équivalence	114
B.1.2	Lien avec la formulation qui introduit \bar{f}_i^{eq}	114

		Note Technique DES	Page 9/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Introduction

Spécificités de LBM_saclay


LBM_saclay est une application écrite en C++ qui est co-développée entre deux directions du CEA : la « Direction des EnergieS » (DES) et la « Direction de la Recherche Fondamentale » (DRF). Elle est destinée aux « calculs haute performance » (High Performance Computing – HPC) et ses noyaux de calculs reposent sur les « méthodes de Boltzmann sur réseaux » (Lattice Boltzmann Methods – LBM). La LBM est une méthode numérique alternative aux méthodes traditionnelles (telles que les Éléments Finis ou les Volumes Finis) qui permet de simuler très simplement une grande variété d'Equations aux Dérivées Partielles (EDPs) qui interviennent dans plusieurs domaines de la physique. Son principal domaine d'application est celui de la physique des fluides (monophasiques turbulents, diphasiques, etc, ...) mais elle trouve aussi des applications en sciences des matériaux et en électro-magnétisme. Grâce à la simplicité de ses deux étapes de collision et de déplacement, la méthode est très bien adaptée au HPC. La principale spécificité de LBM_saclay est qu'elle peut être compilée et exécutée sur plusieurs plateformes HPC, telles que les architectures multi-CPU (pour « Central Process Units ») et les architectures muti-GPU (pour « Graphics Process Units »). La portabilité du code sur les différentes architectures est assurée par l'utilisation de la bibliothèque Kokkos [2, 3]. Deux niveaux de parallélisme sont présents dans LBM_saclay : le parallélisme à mémoire partagée, géré par Kokkos qui assure l'interface entre CUDA, OpenMP et pThreads, et le parallélisme à mémoire distribuée (décomposition de domaine), géré par MPI.

L'utilisation de LBM_saclay est classique : toutes les données d'entrée sont indiquées dans un unique fichier au format ASCII. Ce « jeu de données » (jdd) regroupe en plusieurs sections, les informations relatives *i*) au maillage en 2D ou en 3D, *ii*) aux modèles physiques et ses paramètres, et *iii*) aux formats de sortie des résultats. Après exécution, les champs scalaires et vectoriels sont écrits dans plusieurs fichiers au format vti (en ASCII ou en binaire) qui peuvent être visualisés et post-traités avec Paraview [4]. Des modifications du modèle mathématique (ajout d'équations, modification des termes de couplage, etc ...) nécessitent de nouveaux développements en C++ et une recompilation du code.

Organisation du document

Cette documentation s'adresse aux étudiants ou aux ingénieurs qui souhaitent s'initier à la LBM et au HPC en réalisant des développements dans LBM_saclay. Pour cela, après un premier chapitre qui décrit l'installation du code sur plusieurs architectures, le document est séparé en trois parties. La première partie décrit en quatre chapitres les bases de LBM_saclay. Ensuite, la seconde partie se présente sous la forme de tutoriels qui décrivent en deux chapitres la façon de développer des modèles mathématiques. Enfin la dernière partie est composée d'annexes qui décrivent quelques développements mathématiques relatifs à la LBM.

Dans la partie I, le chapitre 2 rappelle les principaux concepts de la méthode afin d'introduire les nouveaux types C++ qui ont été définis dans le code en relation avec les notations mathématiques. Des exemples d'utilisation de ces nouveaux types permettront de détailler le noyau de calcul du « Navier-Stokes Standard » en LBM. Le chapitre 3 suivant décrit l'organisation générale du code en détaillant plus particulièrement le fichier pilote LBMRun.h et les appels aux noyaux. Enfin, le chapitre 4 décrit les principaux noyaux de calcul qui sont relatifs à l'équation

		Note Technique DES	Page 10/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

du transport par Advection-Diffusion avec et sans terme source ; à l'équation du champ de phase de Allen-Cahn ; et enfin aux équations de Navier-Stokes en version « compressibilité artificielle » avec terme force. Dans tous les chapitres, on rappelle les équations mathématiques continues des modèles physiques, leurs versions discrètes dans le formalisme LBM, et leurs mises en œuvre informatique dans LBM_saclay. Enfin le chapitre 5 est consacré au parallélisme à mémoire distribuée géré par MPI. Le chapitre décrit la transmission des informations d'un nœud de calcul à un autre qui peuvent se limiter aux fonctions de distribution pour les problèmes physiques simples. Pour ceux qui impliquent des couplages et des calculs de champs macroscopiques intermédiaires, il peut être utile de les transmettre aussi lors des communications MPI. On décrit aussi dans ce chapitre le format de sortie HDF5 des fichiers de résultats et la gestion des reprises des calculs lorsqu'ils sont exécutés sur un super-calculateur. Les commandes pour installer, compiler et exécuter LBM_saclay sur les partitions GPUs d'ORCUS, sur Jean-Zay et Topaze-A100, sont présentées dans la section 5.2.

Les modèles mathématiques qui sont décrits dans la première partie du document permettent de simuler plusieurs problèmes simples de transport et d'écoulement (avec ou sans suivi d'interface). Ensuite dans la partie II, les tutoriels permettent de simuler divers problèmes physiques tels que la décomposition spinodale (chapitre 6), la croissance cristalline (chapitre 7). Plusieurs autres tutoriels sont en cours de préparation, en particulier, les écoulements diphasiques isothermes avec une approche par loi d'état et les écoulements diphasiques anisothermes avec une approche par champ de phase (voir [5]).

Les tutoriels sont présentés dans un ordre de complexité croissante en termes de nombre d'équations et de couplage. Il est donc conseillé de les suivre dans l'ordre. De plus, les déclarations relatives à l'écriture d'un nouveau noyau et les instructions pour l'écriture d'une nouvelle condition initiale sont présentées dans le chapitre 6 et ne seront pas rappelées par la suite. Par ailleurs, on montre dans les chapitres 6 et 7 comment calculer certaines grandeurs physiques qui sont utilisées dans [5]. Par exemple, dans le modèle de cette référence, le potentiel chimique est décrit dans le chapitre 6 et le terme source $\partial_t \phi$ dans l'équation de la chaleur est décrit dans le chapitre 7.


Pour simuler des modèles physiques qui impliquent le couplage de plusieurs EDPs, la philosophie de LBM_saclay est d'introduire une fonction de distribution qui obéit à sa propre équation d'évolution en temps (équation discrète de Boltzmann) pour chaque EDP. L'ensemble des EDPs qui composent le modèle complet sont résolues successivement les unes après les autres. L'algorithme de base de la LBM est totalement explicite et on choisit de le conserver tel quel, même si des alternatives implicites de l'opérateur existent dans la littérature. De même, plusieurs opérateurs de collision existent : le Bhatnagar–Gross–Krook (BGK), le Two-Relaxation-Times (TRT) et le Multiple-Relaxation-Times (MRT). Dans cette documentation seule la collision BGK est utilisée.

Choix de la version « master 2020 »

LBM_saclay a pour principal objectif de développer et de simuler des modèles physiques qui impliquent le suivi d'interface entre plusieurs phases. Celles-ci peuvent être liquides, solides ou gazeuses et leurs modèles sont basés essentiellement sur la « théorie du champ de phase ». La version « Master 2020 » de LBM_saclay ne contient que peu de modèles physiques (décrits dans le chapitre 4). En particulier, les simulations du « film boiling » de l'article [5] ne peuvent pas être reproduites avec cette version du code sans réaliser les développements associés au modèle. Ces développements sont réalisés dans une autre version de R&D (https://gitlab.maisondelasimulation.fr/wverdier/LBM_saclay)

Le choix de cette version du code est délibéré car l'objectif de ce document est d'aider le développeur à s'approprier le code source, à partir d'une version allégée, et de réaliser rapidement ses propres développements. Dans cette version, toute l'ossature informatique est bien présente (gestion des entrées-sorties, appels à Kokkos, parallélisme MPI, noyaux de calculs, etc ...) mais les modèles physiques sont réduits au strict minimum. Plutôt que de décrire des noyaux de calcul existants, le choix a été fait de montrer comment les développer soi-même. Ainsi, le document indique les étapes importantes pour développer son modèle, écrire sa condition initiale et « câbler » le tout dans LBMRun.h.

Dans cette optique, les tutoriels de la seconde partie sont essentiels car ils donnent des exemples précis sur les modifications à apporter petit à petit pour développer et valider des modèles concrets. Ils sont réalisés en s'appuyant

		Note Technique DES	Page 11/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		


sur la version « master 2020 » et en s’inspirant de plusieurs versions de développement. Par exemple le tutoriel du chapitre 7 s’appuie sur la méthodologie publiée dans [6] et programmée dans un code **Fortran**.


D’autres développements sont en cours dans **LBM_saclay**, qui, une fois validés et publiés pourront s’ajouter à la liste des tutoriels de la seconde partie. Par exemple, le modèle à champ de phase à deux phases et trois composants développé et validé dans [7] pourra s’ajouter dans la liste des tutoriels. De même le modèle de dissolution [8] (https://gitlab.maisondelasimulation.fr/tboutin/LBM_saclay) d’un milieu poreux pourra lui aussi être proposé dans un tutoriel. La documentation sera mise à jour au fur et à mesure de l’avancement des travaux puis déposée sur le **gitlab**.

Périmètre et perspective de cette documentation

Même si des calculs formels sont développés dans les annexes de la partie IV, en particulier sur les « développements de Chapman-Enskog » (Annexe A) et la technique du changement de variable (Annexe B), cette documentation n’est pas un cours sur la théorie de la LBM. Plusieurs références pédagogiques existent parmi lesquelles on citera [1] pour une très bonne introduction et [9] pour une synthèse des principales approches diphasiques en LBM. Une synthèse de ces dernières peut également être trouvée dans l’article [10]. On pourra se référer à [1] pour *i*) comprendre les liens entre l’équation de Boltzmann continue et l’équation discrète de Boltzmann sur réseaux, et *ii*) une présentation des différentes formulations des termes forces en LBM.

L’origine des « modèles à champ de phase » qui sont présentés dans ce document n’est pas rappelée non plus. Ils ont pour origine la thermodynamique des systèmes hors équilibre. Pour les modèles en mécanique des fluides, on pourra se référer à [11] et pour ceux de la Science des Matériaux on pourra se référer au livre [12] ou à l’article de synthèse [13].

		Note Technique DES	Page 12/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

		Note Technique DES	Page 13/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Chapitre 1

Installation de LBM_saclay

Dans ce chapitre on décrit les commandes d'installation du code et d'exécution des cas tests sur les différentes architectures cibles. La section 1.1 décrit la manière de récupérer les fichiers sources du code à partir du **GitHub** de la Maison De La Simulation (MDLS). Après décompression de l'archive, la section 1.2 décrit les sous-répertoires qui sont présents. La création de l'exécutable sur les architectures CPUs est décrite dans la section 1.3 et celle sur les GPUs est décrite dans la section 1.4. Enfin la section 1.5 décrit un jeu de données d'un cas test. La compilation de **LBM_saclay** nécessite l'utilisation de la commande **cmake** qui doit préalablement être installée, au même titre qu'un compilateur **C++**.

Pour la compilation, une version de **cmake** > 3.3 doit être installée

Dans cette documentation, les développements et les tests ont été réalisés sur un ordinateur portable installé sous Linux/Ubuntu 18.04, avec la version 7.5.0 du compilateur **gcc** et la version 3.10.2 de **cmake**. Pour la visualisation et le post-traitement des résultats de simulation, la version de **paraview** est la 5.4.1 et celle de **gnuplot** est la 5.2.

1.1 Récupération du code source

L'archive **LBM_saclay-master-20-09-2020.tgz** peut être récupérée par mail ou bien par le **GitLab** de la MDLS. Les membres du CEA/DM2S ayant un compte sur le réseau **intra.cea.fr** peuvent également récupérer l'archive dans le répertoire « /tmpcatC/LBM_saclay/ »


1.1.1 Par mail

Envoyer une demande par mail à pierre.kestener@cea.fr ou alain.cartalade@cea.fr. Le fichier (d'extension **.tgz**), doit être sauvegardé, puis « détaré »/« dézippé » :

```
> tar -xvzf LBM_saclay-master-20-09-2020.tgz
```

1.1.2 Par le GitLab de la MDLS du CEA

L'archive contenant le code source peut aussi être récupéré à partir du **GitHub** de la MDLS du CEA/Saclay. On rappelle ci-dessous les trois commandes à exécuter : a) génération d'une clé **ssh**, b) récupération des sources de

		Note Technique DES	Page 14/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

LBM_saclay et de la bibliothèque Kokkos. c) Les commandes séparées de l'étape précédente peuvent être fusionnées en une seule et même commande.

a) Étape préliminaire : génération d'une clé ssh

Si c'est la toute première connexion de l'ordinateur local au **gitlab**, il est nécessaire de générer au préalable une nouvelle clé **ssh**. On suivra la procédure décrite sur la page du **gitlab** :

```
https://gitlab.maisondelasimulation.fr/users/sign_in
login : login du github (ex. alain.cartalade@cea.fr)
password : passwd du github de la MdLS
```

en cliquant sur l'onglet « **SSH Keys** » (panneau sur la gauche) du « **User Settings** » (Edit profile dans le menu déroulant en cliquant sur l'icône en haut à droite). Pour accéder au tutoriel, cliquer sur « **generate one** » puis se référer à la section « **RSA ssh keys** ». Pour générer la clé **ssh** avec la commande depuis une fenêtre **ORCUS**, (ou de **JEAN-ZAY**) il faut taper la commande :

```
> ssh-keygen -t rsa -b 2048
```

Ensuite dans le répertoire **.ssh** d'**ORCUS** (ou de **JEAN-ZAY**) il faut copier le contenu du fichier **id_rsa.pub** sur la page du **gitlab**.

b) Récupérer les sources

On utilise les commandes **git** pour télécharger les sources de **LBM_saclay**. Attention, pour que la commande fonctionne, il est nécessaire d'avoir une clé **ssh** spécifique à la machine locale. Pour télécharger les sources, taper la commande :

```
> git clone https://gitlab.maisondelasimulation.fr/pkestene/LBM_saclay.git
```

Pour récupérer la bibliothèque **Kokkos** taper les 2 commandes

```
> git submodule init
> git submodule update
```

c) Récupérer LBM_saclay + Kokkos

On ajoute l'option **--recursive** à la commande **git clone** pour la récupération du code source et des dépendances. Par exemple pour "cloner" le sous-projet de **wverdier** on tape la commande :


```
> git clone --recursive git@gitlab.maisondelasimulation.fr:wverdier/LBM_saclay.git
```

Pour cloner celui de **tboutin**, on tape la commande :

```
> git clone --recursive git@gitlab.maisondelasimulation.fr:tboutin/LBM_saclay.git
```

1.2 Description des sous-répertoires

Parmi les trois fichiers présents dans le dossier **LBM_saclay-master-20-09-2020**, le fichier **Readme.md** donne les instructions minimales pour compiler le code sur les différentes architectures et exécuter un cas test. Le fichier

		Note Technique DES	Page 15/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

LICENSE.CECILL.EN.utf8 indique la licence de LBM_saclay. Le dossier contient également sept sous-répertoires dont le contenu pour chacun d'eux est décrit ci-dessous.

cmake

doc

Le répertoire contient les documents relatifs à LBM_saclay. Il s'agit de cette documentation et des articles publiés dans les journaux sur lesquels se basent les tutoriels de la seconde partie.

external

Le sous-répertoire « **external** » contient les fichiers sources de la bibliothèque externe « Kokkos ». La bibliothèque est nécessaire à la compilation de LBM_saclay.

post-processing

Le sous-répertoire « **post-processing** » contient des scripts de post-traitements écrits en **python**. Après exécution, LBM_saclay génère des fichiers de sortie au format « **.vti** ». Chacun de ces fichiers peut se visualiser dans le logiciel Paraview [4]. Les scripts en python permettent de post-traiter ces fichiers **.vti** pour obtenir les courbes qui sont comparées aux solutions analytiques du cas test du double-Poiseuille.

reference

Le sous-répertoire « **reference** » contient une mini-maquette LBM écrite en **Fortran** qui a servi au développement initial de LBM_saclay. Cette mini-maquette, contenue dans le sous-répertoire « **Code-Pedago** », est une version très allégée de SILABE3D [14], code de R&D dédié aux méthodes LB. Ce répertoire n'est pas utile, ni pour les développements de LBM_saclay, ni pour l'exécution de ses cas tests.

settings

Le sous-répertoire « **settings** » contient plusieurs jeux de données (jdd) de cas tests pour LBM_saclay. Les fichiers de paramètres d'entrée ont pour extension « **.ini** ». Le sous-répertoire « **settings** » en contient deux autres : « **phase_field** » et « **transport** » qui contiennent les jdd des cas tests de validation de l'équation du champ de phase (voir section) et de l'équation du transport (voir section). Le jdd du cas test du double-poiseuille s'appelle « **test_lbm_d2q9_double_poiseuille.ini** ».

src

Le sous-répertoire « **src** » contient tous les fichiers sources de LBM_saclay. **src** contient des fichiers avec pour extension « **.h** » et « **.cpp** ». Il contient également cinq sous-répertoires « **bc_cond** » (conditions aux limites), « **init_cond** » (conditions initiales), « **kernels** » (étapes de déplacements et de collision de la LBM), « **model** » et « **utils** » (divers utilitaires e.g. pour MPI). Le contenu des fichiers sources sera détaillé dans le chapitre suivant.


1.3 Compilation et exécution d'un cas test sur CPUs

Les commandes qui suivent permettent de créer l'exécutable « LBM_saclay » adapté aux architectures multi-CPU. L'exécutable est créé en deux étapes en utilisant deux commandes : la première commande « **cmake** » permet de créer le fichier « **Makefile** » qui contient les instructions de compilation avec les options appropriées au type de parallélisme disponible sur le calculateur. La seconde commande « **make** » permet de compiler le code en utilisant les options de compilation du **Makefile** généré par la première commande. Deux versions de compilation sont décrites ci-dessous. La première n'utilise que le parallélisme en mémoire partagée **OpenMP** (section 1.3.1) et la seconde utilise le parallélisme en mémoire partagée et distribuée **OpenMP + MPI** (section 1.3.2).

1.3.1 Version CPUs en OpenMP

Création du répertoire build_openmp

Le fichier exécutable sera créé dans le répertoire « **build_openmp** » :

		Note Technique DES	Page 16/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.			

```
> mkdir build_openmp
> cd build_openmp
```

Création du Makefile

Le Makefile est créé par la commande `cmake` à laquelle on ajoute l'option pour utiliser `OPENMP` :

```
> cmake -DKokkos_ENABLE_OPENMP=ON ..
```

En plus de la création du Makefile, la commande recopie aussi les fichiers sources de `LBM_saclay` et la bibliothèque « Kokkos » dans le répertoire « `build_openmp` » (respectivement dans les sous-répertoires « `src` » et « `external` »).

Compilation

Ensuite taper la commande suivante pour compiler le code :

```
> make
```

Le fichier exécutable `LBM_saclay` est créé dans « `build_openmp/src` »

Exécution d'un cas test

Les jeux de données des cas tests sont regroupés dans le répertoire `settings`. Les fichiers ont pour extension `.ini`. Si on souhaite exécuter un cas test de transport, on crée un répertoire dédié qui regroupe les fichiers de sortie du code :

```
> mkdir tests_transport
> cd tests_transport
```

Pour lancer l'exécution il faut taper la commande :

```
> ../../src/LBM_saclay ../../settings/transport/test_lbm_d2q9_transport.ini
```

1.3.2 Version CPUs en OpenMP + MPI

La bibliothèque MPI doit être installée sur le calculateur.

Création répertoire `build_openmp_mpi`


Créer le répertoire :

```
> mkdir build_openmp_mpi
> cd build_openmp_mpi
```

Création du Makefile

Pour utilisation avec MPI, activer l'option « `-DUSE_MPI=ON` » de la commande `cmake` :

```
> cmake -DKokkos_ENABLE_OPENMP=ON -DUSE_MPI=ON ..
```


			Note Technique DES	Page 17/116
	<u>Réf</u> : STMF/LMSF/NT/2022-70869			
			<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.			

Compilation

Ensuite taper la commande suivante pour compiler le code :

```
> make
```

Le fichier exécutable LBM_saclay est créé dans le sous-répertoire « src »

Exécution d'un cas test

La commande ci-dessous permet d'exécuter le code en utilisant quatre tâches MPI, chacune utilisant deux threads OpenMP :

```
> cd src
> mpirun -x OMP_NUM_THREADS=2 -np 4 ./LBM_saclay ../../settings/transport/test_lbm_d2q5_transport_mpi.ini
```

1.4 Compilation et exécution d'un cas test sur GPUs

Les commandes qui suivent permettent de créer l'exécutable « LBM_saclay » adapté aux architectures multi-GPUs. Comme dans la section 1.3, l'exécutable est créé en utilisant trois commandes successives : la première « cmake » permet de créer le fichier « Makefile » qui contient les instructions de compilation avec les options appropriées au type de parallélisme disponible sur le calculateur. La seconde commande « export » permet d'exporter le chemin conduisant au « nvcc_wrapper » de la configuration du calculateur. Enfin « make » permet de compiler le code en utilisant les options de compilation du Makefile généré par la première commande. Comme dans la section 1.3, deux versions de compilation sont décrites ci-dessous. La première n'utilise que le CUDA (section 1.4.1) et la seconde utilise le CUDA et le parallélisme en mémoire distribuée CUDA + MPI (section 1.4.2).

1.4.1 Version GPUs en CUDA

Le pilote Nvidia/CUDA doit être installé sur le calculateur et tous les utilitaires associés.

Compilation

Comme précédemment, on crée un répertoire dédié :


```
> mkdir build_cuda
> cd build_cuda
```

Ensuite il faut ajouter le chemin qui conduit au nvcc_wrapper qui est situé dans les sources de Kokkos de la configuration (très certainement \$(LBM_SACLAY_TOPDIR)/external/kokkos/bin/nvcc_wrapper) :

```
> export CXX=/complete/path/to/kokkos/bin/nvcc_wrapper
```

Activer l'option CUDA dans l'interface cmake en mettant l'option Kokkos_ENABLE_CUDA, puis sélectionner une architecture CUDA, par exemple pour les cartes Nvidia K80, activer l'option Kokkos_ARCH à Kepler37 ; cette option est à adapter au type de carte disponible sur votre machine.

```
> cmake -DKokkos_ENABLE_OPENMP=ON -DKokkos_ENABLE_CUDA=ON -DKokkos_ENABLE_CUDA_LAMBDA=ON
-DKokkos_ARCH_KEPLER37=ON ..
```

		Note Technique DES	Page 18/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Sur les calculateurs ORCUS et JEAN-ZAY qui possèdent des V100, il faut mettre l'option `-DKokkos_ARCH_VOLTA70=ON` (voir chapitre 5). Pour créer l'exécutable « LBM_saclay » il faut ensuite taper la commande :

```
> make -j 8
```

Exécution d'un cas test

```
> cd src
> ./LBM_saclay ../../settings/transport/test_lbm_d2q9_transport.ini
```

1.4.2 Version GPUs en CUDA + MPI

Compilation

Un exemple de compilation en version CUDA + MPI est donné sur le calculateur de la Maison De La Simulation (adresse IP 132.167.204.83) qui possède huit nœuds GPUs.

```
> module load cuda/3.1.2_cuda8.0
> export OMPI_CXX=/path/to/nvcc_wrapper
> export CXX=/opt/local/openmpi-3.1.2_cuda8.0/bin/mpicxx
> mkdir build_cuda_mpi
> cd build_cuda_mpi
> cmake -DKokkos_ENABLE_OPENMP=ON -DKokkos_ENABLE_CUDA=ON -DKokkos_ENABLE_CUDA_LAMBDA=ON
-DKokkos_ARCH_KEPLER37=ON -DUSE_MPI=ON -DUSE_MPI_CUDA_AWARE_ENFORCED=ON ..
> make -j 8
```


Exécution d'un cas test

Exemple d'utilisation sur un domaine décomposé en 2×2 :

```
> mpirun -np 4 ./LBM_saclay ../../settings/test_lbm_d2q9_double_poiseuille.ini
--kokkos-ndevices=4
```

1.5 Description d'un jeu de données : fichiers .ini

Dans cette section on décrit un jeu de données (jdd) d'entrée nécessaire pour l'exécution du code. Les noms des fichiers de jdd ont une extension `.ini` et se trouvent dans le répertoire `settings`. Ces fichiers se présentent sous la forme de plusieurs sections déclarés par `[name:section]` suivis par les paramètres de ces sections et leurs valeurs. Par exemple dans le répertoire `settings/phase_field`, on trouve le jdd pour le cas test qui simule la déformation d'une interface dans un champ de vitesse imposé `test_lbm_d2q9_shrf_vortex.ini`. Les deux premières sections de ce fichiers se présentent comme suit :

		Note Technique DES	Page 19/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

```
[run]
lbm_name=D2Q9
tEnd=10.0
nStepmax=80000
nOutput=500
nPlot=100
dt=1e-4
```

```
[mesh]
nx=100
ny=100
xmin=0.0
xmax=1.0
ymin=0.0
ymax=1.0
boundary_type_xmin=3
boundary_type_xmax=3
boundary_type_ymin=3
boundary_type_ymax=3
```

La section `[run]` indique les paramètres liés au pas de temps du calcul (`dt`) ; au temps final de la simulation (`tEnd`), ou alternativement au nombre d'itérations maximales (`nStepmax`) ; au pas d'écriture des fichiers de sortie (`nOutput`) ; et au pas de temps d'écriture des informations à l'écran (`nPlot`).

La section `[mesh]` regroupe tous les paramètres liés au maillage et aux types de conditions aux limites. Les coordonnées des points du domaine sont `xmin`, `xmax`, `ymin` et `ymax` et le nombre de nœuds sont `nx` et `ny`. Pour chaque face `xmin`, `xmax`, `ymin`, et `ymax`, les types conditions aux limites pour sont précédés par la chaîne `boundary_type_`. Les deux options suivantes sont possibles : flux nul (1) et périodique (3). Signalons que dans les versions en cours de développement la condition de Dirichlet est aussi implémentée et validée.

Dans les informations relatives aux modèle mathématique à simuler sont regroupées dans deux sections suivantes `[lbm]` et `[phase_field]`.


```
[lbm]
gamma0=1.666
problem=SHRF_VORTEX
phase_field_enabled=yes
phase_field_model=Allen-Cahn
fluid_enabled=no
heat_transfert_enabled=no
transport_enabled=no
```

```
[phase_field]
# possible values are : unif, zalesak, tanh, shrf,
# layr, rd10, rd50, ...
# to be defined
init=SHRF_VORTEX
collision=BGK
coupling_with_fluid=no
mobility=5e-4
```

Dans `[lbm]`, on indique quel module physique on souhaite simuler avec les options `phase_field_enabled`, `fluid_enabled`, `heat_transfert_enabled`, et `transport_enabled`. qui peuvent prendre les valeurs `yes` ou `no`. Dans l'exemple, comme l'option `phase_field_enabled` prend la valeur `yes`, on indique ensuite le type d'équation à champ de phase à simuler : `phase_field_model=Allen-Cahn`.

Dans la section `[phase_field]` on indique ensuite les options de résolutions pour cette équation à champ de phase : le type d'initialisation (`init=SHRF_VORTEX`), le type de collision utilisée pour son équation LBE (`collision=BGK`) et une option pour le couplage avec les fluides (`coupling_with_fluid=no`). Enfin on indique la valeur du paramètre de mobilité (`mobility=5e-4`). On remarque qu'il est possible d'insérer des commentaires dans le jdd. Pour cela, il faut faire précéder les lignes de commentaires par un `#`.

Enfin les deux dernières sections sont relatives aux paramètres de la condition initiale `[shrf_vortex]` et aux paramètres de gestion des fichiers des résultats `[output]`.


		Note Technique DES	Page 20/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

```
[shrf_vortex]
# shrf vortex center coordinate, radius and spread
(border thickness)
x0=0.5
y0=0.3
z0=0.5
r0=0.2
spread=0.01
```

```
[output]
write_variables=phi,vx,vy
outputPrefix=LBM_D2Q9_shrf_vortex
outputVtkAscii=yes
```


Dans `[shrf_vortex]`, la condition initiale est celle d'une sphère diffuse de coordonnées `x0`, `y0` et `z0`, de rayon `r0` et dont l'interface diffuse de la valeur `spread`. Il s'agit en fait d'une condition initiale en tangente hyperbolique (Eqs (6.3a) et (6.3b)).

Dans `[output]`, on indique les champs qui seront écrits dans le fichier de sortie : ici `phi`, et les deux composantes de la vitesse `vx` et `vy`. Le préfixe des fichiers de sorties est `LBM_D2Q9_shrf_vortex`, et l'écriture sera fait au format ASCII.

		Note Technique DES	Page 21/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Première partie

Les bases de LBM_saclay

		Note Technique DES	Page 23/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Chapitre 2

Les nouveaux types dans LBM_saclay

Ce chapitre décrit les bases de la mini-application LBM_saclay. Tout d'abord l'algorithme LBM standard est rappelé dans la section 2.1. Le but est de rappeler les principales notations mathématiques afin d'introduire les types C++ associés aux notations mathématiques, qui ont été définis pour faciliter sa programmation. Les nouveaux types, décrits dans la section 2.2, facilitent la programmation dans LBM_saclay en établissant un lien immédiat avec les équations discrètes. Ces derniers sont illustrés en décrivant les principales étapes de l'algorithme qui sont regroupées dans le noyau de calcul `CollideAndStream_NS_STD_Functor.h`.

2.1 Rappels de la LBM « standard »

Dans cette section on rappelle l'algorithme LBM « STanDard » (STD) de base afin d'introduire les principaux concepts de la méthode : la fonction de distribution f_i (2.1.1), les réseaux (section 2.1.2) et les liens avec les équations macroscopiques qui sont simulés par la méthode (section 2.1.3).

2.1.1 La fonction de distribution f_i


Le premier élément fondamental de la méthode est l'utilisation d'une « fonction de distribution » f_i qui peut être vue comme un intermédiaire de calcul pour obtenir les densités ρ et les quantités de mouvement $\rho \mathbf{u}$ (où \mathbf{u} est la vitesse) à l'aide des relations (2.1a) et (2.1b) :

$$\rho = \sum_i f_i \quad (2.1a)$$

$$\rho \mathbf{u} = \sum_i f_i \mathbf{c}_i \quad (2.1b)$$

où l'Eq. (2.1a) est appelée le moment d'ordre zéro (en \mathbf{c}_i) et l'Eq. (2.1b) est appelée le moment d'ordre un (en \mathbf{c}_i). Les vecteurs \mathbf{c}_i sont définis par $\mathbf{e}_i \delta x / \delta t$ où δx est le pas de discrétisation en espace, δt est le pas de temps et les \mathbf{e}_i sont les vecteurs de déplacement des f_i sur un réseau défini par l'utilisateur pour les simulations. L'indice i différencie les différentes directions de déplacements et varie de 0 à N_{pop} où N_{pop} est un entier qui dépend du réseau. Des exemples de réseaux en 2D et en 3D sont indiqués dans la section 2.1.2.

La fonction de distribution f_i est une fonction qui dépend de la position $\mathbf{x} = (x, y, z)^T$, du temps t et de l'indice des populations i . Il s'agit d'un tableau de réels qui dépend des trois indices de position et de i . Cet intermédiaire

		Note Technique DES	Page 24/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

de calcul obéit à l'équation d'évolution (2.2) :

$$f_i(\mathbf{x} + \mathbf{c}_i \delta t, t + \delta t) = f_i(\mathbf{x}, t) - \frac{1}{\tau_f} [f_i(\mathbf{x}, t) - f_i^{eq}(\mathbf{x}, t)] \quad (2.2)$$

qui est une équation totalement explicite : le membre de droite ne dépend que de t alors que le membre de gauche est défini à $t + \delta t$. Le second terme du membre de droite de l'équation (2.2) est l'opérateur de collision et fait intervenir une fonction de distribution à l'équilibre f_i^{eq} définie par :

$$f_i^{eq}(\mathbf{x}, t) = w_i \rho \left[1 + \frac{\mathbf{c}_i \cdot \mathbf{u}}{c_s^2} + \frac{(\mathbf{c}_i \cdot \mathbf{u})^2}{2c_s^4} - \frac{\mathbf{u} \cdot \mathbf{u}}{2c_s^2} \right] \quad (2.3)$$

où les w_i sont des poids qui sont associés aux différentes directions de déplacement du réseau considéré. Pour le calcul de la fonction de distribution à l'équilibre Eq. (2.3), on calcule une fonction intermédiaire $\Gamma(\mathbf{x}, t)$ qui est quelque fois utilisée dans le calcul de la force microscopique F_i (voir chapitre suivant 4) :

$$\Gamma_i(\mathbf{x}, t) = w_i \left[1 + \frac{\mathbf{c}_i \cdot \mathbf{u}}{c_s^2} + \frac{(\mathbf{c}_i \cdot \mathbf{u})^2}{2c_s^4} - \frac{\mathbf{u} \cdot \mathbf{u}}{2c_s^2} \right] \quad (2.4)$$

La fonction de distribution f_i^{eq} est alors obtenue en effectuant le produit avec la densité :

$$f_i^{eq}(\mathbf{x}, t) = \rho(\mathbf{x}, t) \Gamma_i(\mathbf{x}, t) \quad (2.5)$$

Dans l'Eq. (2.2), la relaxation de f_i vers son équilibre f_i^{eq} se fait proportionnellement à l'inverse du taux de collision τ_f , qui est relié à la viscosité cinématique ν du fluide par la relation :

$$\nu = \frac{1}{3} \left(\tau_f - \frac{1}{2} \right) \frac{\delta x^2}{\delta t} \quad (2.6)$$

2.1.2 Définition des réseaux

Le deuxième élément fondamental de la méthode est le choix du réseau sur lequel vont se déplacer les fonctions de distribution f_i . Les réseaux sont définis par un ensemble de vecteurs de déplacement $\mathbf{e}_i = (e_{ix}, e_{iy}, e_{iz})^T$ (où i varie de 0 à N_{pop}). L'ensemble de ces vecteurs définissent le réseau sur lequel se déplacent les fonctions de distribution f_i . Dans LBM_saclay les réseaux populaires sont déjà définis. Il s'agit des réseaux D2Q5 et D2Q9 en 2D et D3Q7, D3Q15 et D3Q19 en 3D. Ils sont schématisés sur la figure 2.1 pour les réseaux en 2D et sur la figure 2.2 pour ceux en 3D. Les vecteurs \mathbf{e}_i sont définis dans la table 2.1 pour les réseaux D2Q5 et D2Q9 dans la table 2.2 pour les réseaux 3D. Comme on le constate ce sont des vecteurs d'entiers dont chaque composante peut prendre les valeurs -1, 0 et +1.

Dans l'Eq. (2.2), les vecteurs de déplacements \mathbf{e}_i interviennent dans la « vitesse du réseau » $\mathbf{c}_i = \mathbf{e}_i \delta x / \delta t$ où les δx et δt sont les pas de discrétisation en espace et en temps respectivement. On note que $\mathbf{x} + \mathbf{c}_i \delta t = \mathbf{x} + \mathbf{e}_i \delta x$ et que $\mathbf{c}_i \delta t = \mathbf{e}_i \delta x$.

2.1.3 Description de l'algorithme et liens avec les équations macroscopiques

L'algorithme LBM est le suivant : à l'instant t , les densités et vitesses sont mises à jour à l'aide des relations Eqs. (2.1a) et (2.1b). Ces densités et vitesses sont utilisées pour mettre à jour la nouvelle fonction à l'équilibre Eq. (2.3). La fonction à l'équilibre est elle-même utilisée pour calculer la collision (second terme du membre de droite de l'Eq. (2.2)). Ensuite on réalise l'étape de déplacement de la fonction de distribution f_i (membre de gauche de l'Eq. (2.2)). Toutes ces étapes sont itérées dans une boucle en temps. Dans LBM_saclay, les étapes de collision et de déplacement sont regroupées en une seule étape en introduisant un tableau supplémentaire temporaire qui stocke les fonctions de distribution au temps $t + \delta t$ (voir section 2.2.1).

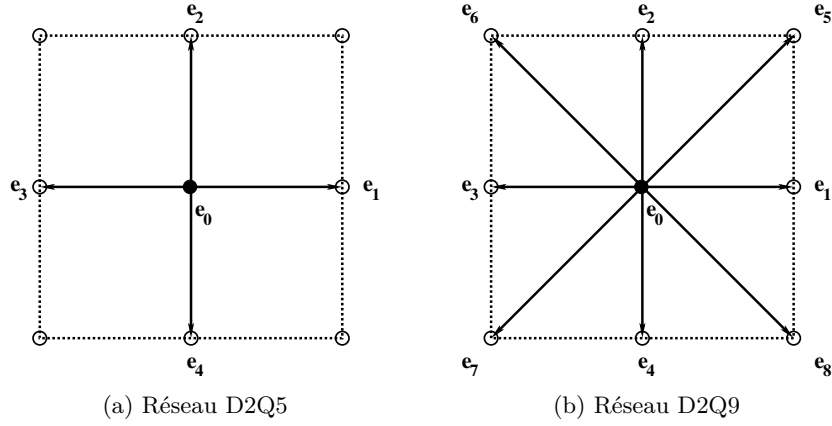


FIGURE 2.1 – Réseaux en 2D.

Définition des vecteurs e_i pour le D2Q5

$$e_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad e_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad e_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad e_3 = \begin{pmatrix} -1 \\ 0 \end{pmatrix} \quad e_4 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

Vecteurs supplémentaires pour le D2Q9

$$e_5 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad e_6 = \begin{pmatrix} -1 \\ 1 \end{pmatrix} \quad e_7 = \begin{pmatrix} -1 \\ -1 \end{pmatrix} \quad e_8 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

TABLE 2.1 – Définition des vecteurs de déplacement e_i pour les réseaux en 2D.

Définition des vecteurs e_i pour le réseau D3Q7

$$e_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \quad e_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \quad e_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad e_3 = \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix} \quad e_4 = \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix} \quad e_5 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad e_6 = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$

Vecteurs e_i supplémentaires pour le réseau D3Q15

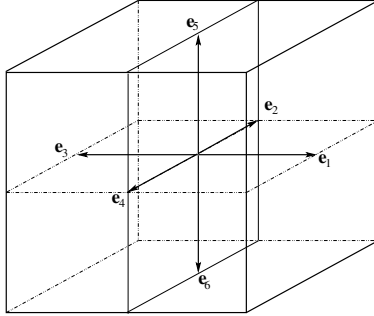
$$e_7 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \quad e_8 = \begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix} \quad e_9 = \begin{pmatrix} -1 \\ -1 \\ 1 \end{pmatrix} \quad e_{10} = \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix} \\ e_{11} = \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix} \quad e_{12} = \begin{pmatrix} -1 \\ 1 \\ -1 \end{pmatrix} \quad e_{13} = \begin{pmatrix} -1 \\ -1 \\ -1 \end{pmatrix} \quad e_{14} = \begin{pmatrix} 1 \\ -1 \\ -1 \end{pmatrix}$$

Vecteurs e_i supplémentaires pour le réseau D3Q19

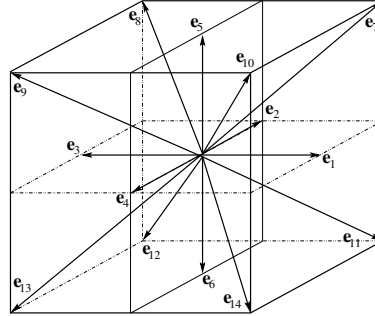
$$e_7 = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \quad e_8 = \begin{pmatrix} -1 \\ 1 \\ 0 \end{pmatrix} \quad e_9 = \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix} \quad e_{10} = \begin{pmatrix} -1 \\ -1 \\ 0 \end{pmatrix} \quad e_{11} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \quad e_{12} = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix} \\ e_{13} = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix} \quad e_{14} = \begin{pmatrix} -1 \\ 0 \\ -1 \end{pmatrix} \quad e_{15} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \quad e_{16} = \begin{pmatrix} 0 \\ -1 \\ 1 \end{pmatrix} \quad e_{17} = \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix} \quad e_{18} = \begin{pmatrix} 0 \\ -1 \\ -1 \end{pmatrix}$$

TABLE 2.2 – Définition des vecteurs e_i pour les réseaux D3Q7, D3Q15 et D3Q19.

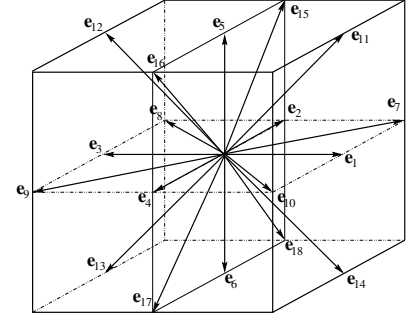
		Note Technique DES	Page 26/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		



(a) Réseau D3Q7



(b) Réseau D3Q15



(c) Réseau D3Q19

FIGURE 2.2 – Réseaux 3D

La viscosité cinématique ν est lue dans le fichier d'entrée, puis le taux de collision τ_f est déduit en inversant l'Eq. (2.6). Si cette viscosité cinématique est indépendante de la position et du temps cela peut se faire à l'initialisation du code, sinon cette étape devra être insérée dans la boucle en temps.

Les liens entre cet algorithme et les équations macroscopiques sont établis avec un développement de Chapman-Enskog. Les principales étapes de la démonstration sont présentées en annexe A. Les développements montrent que cet algorithme permet de simuler les équations de Navier-Stokes suivantes :

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \quad (2.7a)$$


$$\frac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot (\rho \mathbf{u} \mathbf{u}) = -\nabla p + \nabla \cdot [\eta (\nabla \mathbf{u} + \nabla \mathbf{u}^T)] \quad (2.7b)$$

Dans les équations (2.7a) et (2.7b), $\rho \equiv \rho(\mathbf{x}, t)$ est la densité, $\mathbf{u} \equiv \mathbf{u}(\mathbf{x}, t)$ est le vecteur vitesse et η est la viscosité dynamique du fluide. La pression $p(\mathbf{x}, t)$ est donnée par la loi d'état des gaz parfaits $p = \rho c_s^2$ où $c_s^2 = (1/3)\delta x^2/\delta t^2$ où δx et δt sont respectivement les pas de discrétisation en espace et en temps.

2.2 Définitions des nouveaux types C++ dans LBM_saclay

Du rappel de la section précédente, on différencie d'ores et déjà 1) les champs physiques macroscopiques et 2) les variables et les fonctions qui sont indicées par i et qui sont représentatives du réseau et de la LBM. Les champs physiques macroscopiques sont la densité $\rho(\mathbf{x}, t)$, la vitesse $\mathbf{u}(\mathbf{x}, t) = (u_x, u_y, u_z)^T$, la pression $p(\mathbf{x}, t)$ et la viscosité cinématique ν . Les grandeurs indicées par i sont la fonction de distribution f_i , la fonction de distribution à l'équilibre f_i^{eq} , les poids w_i et les directions de déplacements \mathbf{e}_i . Le coefficient c_s qui apparaît aux dénominateurs de f_i^{eq} est une constante du réseau définie par $c_s = (1/\sqrt{3})c$ où $c = \delta x/\delta t$. Le taux de relaxation τ_f est lui aussi une variable microscopique qui est relié à la viscosité cinématique macroscopique par la relation Eq. (2.6). Seule la fonction à l'équilibre f_i^{eq} Eq. (2.3) mélange dans son membre de droite des champs macroscopiques (ρ et \mathbf{u}) et des variables indicées par i (w_i et \mathbf{e}_i).

Dans LBM_saclay, en plus des types classiques au langage C++, plusieurs nouveaux types sont définis pour établir plus facilement les liens entre la méthode numérique, les instructions en C++ et les appels spécifiques à la bibliothèque Kokkos. On décrit tout d'abord dans la section) les types qui ont été définis. Ensuite dans la section 2.2.2 on donne des exemples d'utilisation de ces nouveaux types en décrivant le noyau de calcul `CollideAndStream_NS_STD_Functor.h`. Dans le chapitre suivant, on décrira l'organisation générale du code en détaillant le fichier `LBMRun.h` contenant la classe qui pilote toutes les instructions pour la réalisation d'un calcul.

			Note Technique DES	Page 27/116
	<u>Réf</u> : STMF/LMSF/NT/2022-70869			
			<u>Date</u> : 21/11/2022	<u>Indice</u> : A
LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.				

2.2.1 Liste des nouveaux types

En plus des types « entier » (`int`) et « booléen » (`bool`), plusieurs types C++ sont définis dans `LBM_saclay`. Le premier d'entre eux est le type « réel » (`real_t`) et les autres sont spécifiques à la LBM elle-même : `LBMArray`, `FArray`, `LBMParams`, `LBM_speeds` et `FState`. Ils sont décrits ci-dessous.

Le type « réel » dans `LBM_saclay`

`real_t` est un type qui définit un réel en `double` ou bien en `float` selon l'architecture sur laquelle sera compilée et exécutée la mini-app. Le type `real_t` est défini dans le fichier `real_type.h`. Il est utilisé dans tous les fichiers sources.

Les types spécifiques à la LBM dans `LBM_saclay`

`LBMArray` est un type qui définit un champ physique macroscopique. Il est défini dans le fichier `kokkos_shared.h`. Les champs physiques macroscopiques sont la densité ρ , la pression p , les composantes des vitesses u_x , u_y et u_z . Pour les simulations en 2D il s'agit d'un tableau de réels à trois indices `LBMArray2d`, dont les deux premiers i et j correspondent aux positions x et y , et le troisième différencie les champs (ρ , p , u_x , u_y , u_z). Pour les simulations en 3D, il s'agit d'un tableau de réels `LBMArray3d`, à quatre indices (i , j , k et numéro du champ). Ces deux types sont définis dans le fichier `kokkos_shared.h` par l'instruction (e.g. `LBMArray3d`)

```
using LBMArray3d = Kokkos::View<real_t****, Device>;
```


Le choix des structures de données à 3 ou 4 indices est fonction de la dimension spatiale `ndim` indiquée dans le fichier d'entrées `.ini`. Une fois cette indication connue, le type `LBMArray2d` ou `LBMArray3d` est attribué à `LBMArray` dans le fichier pilote `LBMRun.h` (voir section 3.1) par l'instruction :

```
using LBMArray = typename std::conditional<dim==2,
                                           LBMArray2d,
                                           LBMArray3d>::type;
```

`FArray` est un type qui est associé à une fonction de distribution f_i , il est défini dans le fichier `kokkos_shared.h`. Comme pour `LBMArray`, pour les simulations en 2D, il s'agit d'un tableau de réels `FArray2d` à trois indices où les deux premiers i et j correspondent aux positions x et y , tandis que le troisième $ipop$ correspond au numéro de la population i . Pour les simulations en 3D, il s'agit d'un tableau de réels `FArray3d`, à quatre indices (x , y , z et i). Le choix des tableaux à 3 ou 4 indices est fonction de la dimension spatiale indiquée dans le fichier d'entrées `.ini`. Une fois cette indication connue, le type `FArray2d` ou `FArray3d` est attribué à `FArray` dans le fichier pilote `LBMRun.h` (voir section 3.1) par l'instruction :

```
using FArray = typename std::conditional<dim==2,
                                           FArray2d,
                                           FArray3d>::type;
```

`LBMArrayConst` est de même type que `LBMArray`, mais il se différencie de celui-ci par le fait que les réels sont constants. Comme pour `LBMArray`, il est défini dans le fichier `kokkos_shared.h` et selon la dimension spatiale demandée dans le fichier d'entrée il est associé soit à `LBMArrayConst2d` ou bien à `LBMArrayConst3d`. Pour ce dernier, il est défini dans le fichier `kokkos_shared.h` par la commande de la bibliothèque Kokkos :

		Note Technique DES	Page 28/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

```
using LBMArrayConst3d= Kokkos::View<const real_t****, Device>;
```

LBMArrayConst2d et LBMArrayConst3d permettent de définir respectivement FArrayConst2d et FArrayConst3d qui possèdent le même type.

LBMParams est un type qui définit les paramètres numériques d'un calcul, il est défini dans le fichier **LBMParams.h**. Il s'agit d'une structure de données (déclaré par **struct**) qui contient entre autres (liste non exhaustive) le nombre total de nœuds (en comptant les nœuds fantômes) N_x (**isize**), N_y (**jsize**) et N_z (**ksize**); les coordonnées des dimensions dans chaque dimension x_{min} (**xmin**), x_{max} (**xmax**), y_{min} (**ymin**), y_{max} (**ymax**), z_{min} (**zmin**) et z_{max} (**zmax**); les paramètres de gestion des fichiers de sortie, etc ... Il contient également des booléens d'options pour la réalisation d'un calcul fluide (**lbm_fluid_enabled**), de transport (**lbm_transport_enabled**), de champ de phase (**lbm_phase_field_enabled**) et les pas de discrétisation en espace δx (**dx**) et δt (**dt**).

LBM_speeds est le type qui définit les vecteurs de déplacement e_i de la méthode LB. Le type est défini dans le fichier **LBM_Base_Functor.h**. Les différents vecteurs de déplacement e_i associés aux réseaux en 2D et 3D définis dans les tables 2.1 et 2.2 ainsi que les poids w_i qui apparaissent dans les fonctions à l'équilibre f_i^{eq} sont définis dans le fichier **LBM_Base_Functor.cpp**. Les vitesses du réseau c_i qui apparaissent dans les Eqs. (2.2), (2.3) et (2.1b) sont obtenues en multipliant par $c = \delta x / \delta t$.

FState est un type de fonction de distribution mais défini localement, c'est-à-dire qu'il ne dépend plus de la position mais uniquement de l'indice i . Le type **FState** est défini dans chaque noyau de calcul dont le fichier se trouve dans le répertoire **kernels**. Par exemple dans le fichier **CollideAndStream_NS_STD_Functor.h**, il est défini par la ligne :

```
using FState = typename Kokkos::Array<real_t, npop>;
```


LBMArrayHost est un type qui est introduit pour la copie du tableau de type **LBMArray** (défini ci-dessus) sur les CPUs lorsque le calcul principal est réalisé sur des GPUs. La copie est réalisée au moment de l'écriture par l'instruction « **Kokkos::deep_copy(data_h, data);** » dans le fichier **saveVTK.cpp** où **data_h** et **data** ont respectivement le type **LBMArrayHost** et **LBMArray**. C'est le tableau **data_h** qui est ensuite écrit sur le disque.

2.2.2 Exemples d'utilisation : le noyau de calcul **CollideAndStream_NS_STD_Functor.h**

L'algorithme LBM qui est décrit par les Eqs. (2.1a)–(2.6) est programmé dans un noyau de calcul qui est situé dans le sous-répertoire « **src/kernels** ». Il s'agit du fichier « **CollideAndStream_NS_STD_Functor.h** » qui contient une classe C++ nommée « **CollideAndStream_ns_std_Functor** ». À la fin de ce fichier, les noms de variables **params**, **f1**, **w** et **E** sont respectivement définis par les types **LBMParams**, **FArray**, **LBM_Weights** et **LBM_speeds** décrits ci-dessus. Ces nouvelles notations permettent de faire le lien rapidement entre les notations mathématiques de l'algorithme et les noms de variables informatiques, à savoir, **E** pour e_i et **w** pour w_i . Les variables **f0** et **f1** sont des tableaux de réels à trois ou quatre dimensions selon l'option choisie dans le jeu de données. Dans la classe **CollideAndStream_ns_std_Functor**, les opérations sont réalisées localement, c'est-à-dire pour une position fixée par les indices de positions **i**, **j** et **k**. Dans ce même fichier, les variables **f**, **feq** et **gamma** ont le type **FState**.

f0 est un exemple de type **FArrayConst** et il est utilisé pour la fonction de distribution $f_i(\mathbf{x}, t)$ qui apparaît dans le membre de droite de l'Eq. (2.2). Dans la version 3D de la classe, on manipule ce tableau en utilisant ses quatre indices qui le définissent **f0(i,j,k,ipop)** où **i**, **j**, **k** sont les trois indices de position x , y et z et **ipop** est l'indice de la direction de déplacement i .

f, **feq**, **gamma** sont des exemples de type **FState**. Ils sont utilisés respectivement pour f_i , f_i^{eq} et Γ_i . Comme ces trois variables sont définies *localement* (i.e. pour une position fixée **i**, **j**, **k**), on ne manipule qu'un

		Note Technique DES	Page 29/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

seul indice, celui du numéro de la population `ipop` qui représente l'indice de la population i . Dans le fichier `CollideAndStream_NS_STD_Functor.h` il est utilisé comme un tableau de travail intermédiaire pour effectuer les opérations élémentaires de l'algorithme localement à la fonction. Par exemple dans la boucle ci-dessous (issue du noyau 3D) on transmet les informations locales de `f0` dans `f` avant de réaliser les opérations avec `f` :

```
for (int ipop=0; ipop<npop; ++ipop) {
    f[ipop] = f0( i, j, k, ipop );
}
```

puis on utilise `f` pour calculer le moments d'ordre zéro par la boucle :

```
for (int ipop=0; ipop<npop; ++ipop) {
    rho += f[ipop];
    ...
}
```

où ... indique les instructions pour la mise à jour du moment d'ordre 1. Celles-ci sont décrites ci-dessous avec la définition de `E`. La variable macroscopique ρ ainsi obtenue est ensuite utilisée dans le calcul de la fonction à l'équilibre.

`params` est un exemple type `LBMParam`. Il s'agit d'un tableau qui regroupe les paramètres communs à l'ensemble des classes. Lorsqu'on a besoin d'utiliser un de ses paramètres, par exemple δx , on définit localement une constante réelle `dx` par l'instruction :

```
const real_t dx = params.dx;
```


Il en est de même pour tous les autres paramètres relatifs à la taille du maillage, par exemple la variable `isize` est préalablement définie par l'instruction `const int isize = params.isize;`)

`E` est un exemple de type `LBM_speeds`. `E` est utilisé pour l'ensemble des vecteurs e_i qui définissent un réseau. Dans `CollideAndStream_ns_std_Functor`, `E` est initialisé par l'instruction

```
this->init_speeds(E);
```

où la fonction `init_speeds` est définie dans le fichier `LBM_Base_Functor.cpp`. Les vecteurs e_i sont des vecteurs à deux (en 2D) ou trois composantes (en 3D). On les manipule à l'aide de deux indices : le premier `ipop` pour le numéro de la population et le second `IX`, `IY` ou `IZ` pour le numéro de sa composante. Les vecteurs e_i sont utilisés pour mettre à jour chaque composante du moment d'ordre un (la quantité de mouvement) de la fonction de distribution (Eq. (2.1b)) :

```
for (int ipop=0; ipop<npop; ++ipop) {
    ...
    rhou += dx/dt*E[ipop][IX]*f[ipop];
    rhov += dx/dt*E[ipop][IY]*f[ipop];
    rhow += dx/dt*E[ipop][IZ]*f[ipop];
}
```

		Note Technique DES	Page 30/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

où « ... » représente l'instruction pour le calcul du moment d'ordre zéro décrit ci-dessus. Les composantes e_{ix} , e_{iy} et e_{iz} du vecteur \mathbf{e}_i sont donc explicitement associées aux notations $\mathbf{E}[\text{ipop}][\text{IX}]$, $\mathbf{E}[\text{ipop}][\text{IY}]$ et $\mathbf{E}[\text{ipop}][\text{IZ}]$ et chacune des composantes est préalablement multipliée par \mathbf{dx}/\mathbf{dt} pour obtenir $\mathbf{c}_i = c\mathbf{e}_i$ avec $c = \delta x/\delta t$. Le vecteur \mathbf{e}_i est également utilisé dans le produit scalaire $\mathbf{e}_i \cdot \mathbf{u}$ de la fonction à l'équilibre f_i^{eq} (Eq. (2.3)). Ce produit scalaire **scal** est calculé par l'instruction :

```
real_t scal = dx/dt*(E[ipop][IX]*vx + E[ipop][IY]*vy + E[ipop][IZ]*vz);
```

Le vecteur \mathbf{e}_i apparaît aussi dans le membre de gauche de l'équation discrète de Boltzmann (Eq. (2.2)). On le retrouve dans la mise à jour de **f1** qui est décrite ci-dessous.

w est un exemple de type **LBM_Weights**. Pour chaque réseau, les poids w_i prennent des valeurs différentes en fonction du problème physique simulé. Ils sont initialisés dans le fichier **CollideAndStream_ns_std_Functor** par l'instruction

```
this->init_weights(w);
```

où la fonction **init_weights** est définie dans le fichier **LBM_Base_Functor.cpp**. Les poids w_i sont utilisés dans la définition de la fonction Γ_i (Eq. (2.4)) et la fonction d'équilibre (Eq. (2.5)) par les instructions :

```
gamma[ipop] = w[ipop] * (1.0 + w1*scal + w2*scal*scal - w3*vv);
feq[ipop] = rho * gamma[ipop];
```


où **scal** a déjà été défini et **vv** est le produit scalaire de la vitesse $\mathbf{u} \cdot \mathbf{u}$. Les dénominateurs c_s^2 , $2c_s^4$ et $2c_s^2$ sont notés **w1**, **w2** et **w3** et ont préalablement été définis en début de fichier par les instructions :

```
const real_t c = params.dx / params.dt;
const real_t w1 = 3.0 / (c*c);
const real_t w2 = 9.0 / 2 / (c*c*c*c);
const real_t w3 = 3.0 / 2 / (c*c);
```

f1 est un exemple de type **FArray** et il est utilisé pour la fonction de distribution $f_i(\mathbf{x} + \mathbf{c}_i\delta t, t + \delta t)$ (membre de gauche de l'Eq. (2.2)). Le type est similaire à celui de **FArrayConst** sauf que ses valeurs sont modifiées à l'intérieur de la classe.

```
f1(i+E[ipop][IX],
  j+E[ipop][IY],
  k+E[ipop][IZ],ipop) =
  f[ipop] - tau_inv * ( f[ipop] - feq[ipop] );
```

Dans cette ligne, le membre à droite du signe égal est l'étape de collision et le membre à gauche est l'étape de déplacement de la fonction de distribution. Les étapes de collision/déplacement sont « fusionnées » en une seule instruction. Comme l'algorithme est totalement explicite, la dépendance en temps se traduit par l'allocation en mémoire de deux tableaux : **f0** et **f1**.

		Note Technique DES	Page 31/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Chapitre 3

Organisation de LBM_saclay


LBM_saclay est constitué de plusieurs fichiers sources qui sont essentiellement 1) les noyaux de calculs LBM et 2) les fonctions nécessaires à l'exécution d'un calcul. Le noyau de calcul « LBM STanDard » a déjà été décrit dans le chapitre 2 par le biais d'exemples d'utilisation des nouveaux types C++ qui ont été définis (section 2.2.2). La description des autres noyaux de calcul fera l'objet du chapitre 4 qui présentera successivement les algorithmes pour la simulation de problèmes de « transport », de « champ de phase » et de problèmes couplés « fluide/champ de phase ». Dans ce chapitre, on décrit l'organisation du code en détaillant plus particulièrement le fichier pilote `LBMRun.h` (section 3.1). Celui-ci est situé dans le répertoire `src` et contient la classe `LBMRun` qui pilote l'ensemble des tâches nécessaires à la réalisation d'un calcul, c'est-à-dire l'allocation de la mémoire, la gestion des conditions initiales, les itérations en temps, la gestion des entrées-sorties etc ... La section 3.2 indique quelques instructions pour faire les appels à `Kokkos`. On rappelle que c'est cette bibliothèque qui assure l'interface entre 1) les appels `OpenMP` pour le parallélisme à mémoire partagée sur CPU multi-threads et 2) les appels `CUDA` pour l'exécution sur les cartes graphiques. Enfin la section 3.3 décrit le rôle des autres fonctions telle que `LBM_enums.h`, `LBMPParams.h`, `LBMPParams.cpp` et `FieldManager.h`. Des modifications sont à apporter à ces fonctions auxiliaires lorsqu'on cherche à développer son propre modèle (voir chapitre 6).

3.1 Le fichier pilote « LBMRun.h »

Plusieurs fonctions sont déclarées dans la classe `LBMRun` puis définies à l'extérieur (par exemple `run`, `update`, `saveData`, `make_boundary`, `make_boundaries_no_mpi`, `make_boundaries_mpi`, etc ...). D'autres fonctions sont déclarées et définies à l'intérieur de la classe elle-même (par exemple `swap_distribution`, et les fonctions de mise à jour `update_transport`, `update_phase_field`, `update_navier_stokes`, etc ...). Le fichier `LBMRun.h` inclut en premier tous les noyaux de calcul qui sont définis dans le sous-répertoire `src/kernels` (section 3.1.1). On y retrouve aussi l'allocation mémoire des tableaux (section 3.1.2), la gestion des conditions initiales (section 3.1.3), l'incrément du pas de temps (section 3.1.4), la gestion des entrées-sorties (section 3.1.5) et la gestion du parallélisme à mémoire distribuée MPI (section 3.1.6).

3.1.1 Gestion des noyaux de calcul dans LBMRun.h : insertion des fichiers de `src/kernels`

On retrouve dans les premières lignes du fichier `LBMRun.h` les instructions `#include "kernels/xxxx.h"` où `xxxx.h` sont les noms des fichiers qui contiennent les classes qui résolvent un problème physique donné (les noyaux de calcul). Par exemple le fichier qui contient les instructions du modèle de Navier-Stokes StanDard est le fichier `CollideAndStream_NS_STD_Functor.h` qui contient la classe « `class CollideAndStream_ns_std_Functor` ».

		Note Technique DES	Page 32/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Dans **LBMRun.h**, ce fichier est inclus par la commande :

```
//#include "kernels/CollideAndStream_NS_STD_Functor.h"
#include "kernels/CollideAndStream_NS_CAC_Functor.h"
```

Dans la version du 20/09/2020 de **LBM_saclay**, ce noyau n'est pas utilisé et cette ligne est mise en commentaire. Par contre le noyau de calcul relatif au modèle couplé Navier-Stokes/Conservative Allen-Cahn (décrit dans le chapitre suivant) est pris en compte.

3.1.2 Allocation mémoire des tableaux dans **LBMRun.h** : la fonction **LBMRun**

Dans le fichier **LBMRun.h**, on y trouve la définition de la classe **LBMRun** (déclarée par `class LBMRun`) et la fonction **LBMRun** (déclarée par `LBMRun<dim,npop>::LBMRun(...)`). Celle-ci regroupe toutes les instructions dédiées à l'initialisation d'un calcul et pour laquelle deux de ses étapes sont : 1) l'allocation en mémoire des tableaux et 2) la gestion des conditions initiales.

Dans **LBMRun**, qui est déclaré par


```
LBMRun<dim,npop>::LBMRun(LBMParams& params, ConfigMap& configMap) :
    params(params),
    configMap(configMap)
{
}
```

les instructions pour l'allocation en mémoire des tableaux en 3D sont :

```
...
} else if (dim==3) {
    data = LBMArray("lbm_data", isize, jsize, ksize, nbVar);
    data_h = Kokkos::create_mirror_view(data);
    f0 = FArray("f0" , isize, jsize, ksize, NPOP);
    if (params.lbm_phase_field_enabled)
        g0 = FArray("g0" , isize, jsize, ksize, NPOP);
    if (params.lbm_transport_enabled)
        h0 = FArray("h0" , isize, jsize, ksize, NPOP);
    f_tmp = FArray("f_tmp", isize, jsize, ksize, NPOP);
}
```

Au minimum les deux tableaux **f0** et **f_tmp** de type **FArray** sont alloués en mémoire. Si les options pour la résolution des problèmes de type « champ de phase » (**lbm_phase_field_enabled**) et de « transport » (**lbm_transport_enabled**) sont indiquées dans le fichier d'entrées alors des tableaux supplémentaires (toujours de type **FArray**) sont aussi alloués pour les fonctions de distribution supplémentaires g_i (**g0**) et h_i (**h0**). L'utilisation des fonctions g_i et h_i sera décrite dans le chapitre suivant. La mémoire est allouée en 2D ou en 3D selon la dimension spatiale indiquée dans le fichier d'entrée. Dans l'algorithme, **data** est le tableau de type **LBMArray** qui contient l'ensemble des champs physiques macroscopiques. La fonction **init_condition(data)** est ensuite appelée

```
...
init_condition(data);
...
```


		Note Technique DES	Page 33/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

3.1.3 Gestion des conditions initiales dans LBMRun.h

La fonction `init_condition` gère les conditions initiales selon les options du problème physique demandée : `init_condition_fluid`, `init_condition_phase_field` et `init_condition_transport`. Ces trois fonctions sont définies dans le même fichier `LBMRun.h`. Elles sont écrites de manière à définir le champ macroscopique (respectivement u , ϕ ou C selon le problème), puis d'utiliser ce champ pour calculer la fonction d'équilibre f_i^{eq} , ou g_i^{eq} ou h_i^{eq} .

Dans `init_condition`, on trouve les conditions pour exécuter chaque condition initiale :

```
template<int dim, int npop>
void LBMRun<dim, npop>::init_condition(LBMArray ldata)
{
    if (params.lbm_fluid_enabled)
        init_condition_fluid(ldata);
    if (params.lbm_phase_field_enabled)
        init_condition_phase_field(ldata);
    if (params.lbm_transport_enabled)
        init_condition_transport(ldata);
} // LBMRun<dim, npop>::init_condition
```

Par exemple pour l'équation du champ de phase, on trouve les conditions pour initialiser le disque de Zalesak ou le vortex dans la fonction `init_condition_phase_field` :


Dans `init_condition_phase_field`, on retrouve les conditions pour chaque condition initiale :

```
void LBMRun<dim, npop>::init_condition_phase_field(LBMArray ldata)
{
    ...
    if (params.phase_field_init == ZALESAK) {
        Zalesak_Functor<dim, npop>::apply(configMap, params, fm, ldata);
    } else if (params.phase_field_init == SHRF_VORTEX) {
        Shrf_Vortex_Functor<dim, npop>::apply(configMap, params, fm, ldata,
        initVelocityOnly, iStep);
    }
}
```

Les classes `Zalesak_Functor` et `Shrf_Vortex_Functor` sont écrites dans deux fichiers `.h` qui se trouvent dans le sous-répertoire `src/init_cond/phase_field`. Il s'agit de `InitPhi_Zalesak.h` et `InitPhi_Shrf_Vortex.h`. Pour cette dernière initialisation, on donne quelques précisions sur les équations qui sont programmées dans le chapitre 6 qui se concentre sur l'équation de Cahn-Hilliard. Les appels et l'`#include` à faire pour tenir compte de ces fichiers sont indiqués dans la section 6.2.4.

Ensuite, la fonction `apply` est appelée avec le tag `TAG_COMPUTE_FEQ` pour calculer la fonction d'équilibre g_i^{eq} :

```
CollideAndStream_phase_field_Functor<dim, npop>::apply(configMap, params,
    fm, ldata,
    g0, f_tmp,
    CollideAndStream_phase_field_Functor<dim, npop>::TAG_COMPUTE_FEQ);
}
```

		Note Technique DES	Page 34/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

3.1.4 Itération en temps dans LBMRun.h : les fonctions run et update

Le fichier LBMRun.h gère les combinaisons de résolution des noyaux de calcul et effectue le pas de temps de calcul de chaque noyau. On retrouve dans ce fichier la fonction `run` :

Dans `run`, qui est déclarée par :

```
template<int dim, int npop>
void LBMRun<dim, npop>::run()
{
    ...
} // LBMRun<dim, npop>::run
```

on trouve l'incrémentation en temps par la commande `while` :

```
while (m_t < params.tEnd && nStep < params.nStepmax) {
    ...
    // perform one step integration
    update(data, nStep);
    // increase time
    nStep++;
    m_t += m_dt;
} // end solver loop
```

où `nStep` est le numéro de l'itération en temps et où « ... » représente toutes les instructions intermédiaires qui sont réalisées dans le `while`. La fonction `run` est elle-même appelée par la commande « `lbm->run()`; » écrite dans le fichier `main.cpp`. Ce fichier `main.cpp` n'a pas pour vocation d'être modifié.


Le calcul lui-même est réalisé dans la fonction `update`

Dans `update`, qui est déclarée par :

```
template<int dim, int npop>
void LBMRun<dim, npop>::update(LBMArray ldata, int nStep)
{
    ...
} // LBMRun::update
```

regroupe des fonctions dont les noms commencent par la chaîne de caractères « `update_` » tels que : « `update_transport` », « `update_phase_field` » et « `update_navier_stokes` ».

Ces fonctions sont alors exécutées si l'option qui leur est associée dans le fichier d'entrée est demandée. Cette fonction intermédiaire permet de gérer la résolution des combinaisons possibles des EDPs en fonction du problème physique résolu. Par exemple la simulation d'un problème de Poiseuille ne nécessite de résoudre que le problème de Navier-Stokes (`update_navier_stokes`), alors que la simulation du problème d'ébullition en film nécessite de résoudre le problème de Navier-Stokes couplé à l'équation du champ de phase (`update_phase_field`) et à l'équation de la chaleur (`update_thermics`). On retrouve donc les instructions suivantes dans la fonction `update` définie dans la classe `LBMRun` :

		Note Technique DES	Page 35/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

```
{
  if (params.lbm_fluid_enabled)
    update_navier_stokes(f0, f_tmp);
  if (params.lbm_phase_field_enabled)
    update_phase_field(g0, f_tmp, nStep);
  if (params.lbm_transport_enabled)
    update_transport(h0, f_tmp);
}
```

Dans `update_transport`, on peut voir qu'elle exécute cinq étapes :

1. `make_boundaries` (tient compte des nœuds fantômes dans la fonction de distribution)
2. `CollideAndStream_mdr_transport_Functor` (appel au noyau de calcul)
3. `make_boundaries` (met à jour les conditions aux limites après l'étape de déplacement)
4. `Compute_density_Functor` (calcul du moment d'ordre zéro)
5. `swap_distribution` (met à jour la nouvelle fonction `f1` dans `f0`)

Dans `update_phase_field` on retrouve ces mêmes cinq étapes auxquelles est ajoutée une étape de prise en compte d'une vitesse connue analytiquement (pour le cas test de la déformation dans un vortex).


Dans `update_navier_stokes`, qui est déclarée par :

```
void update_navier_stokes(FArray& f_in, FArray& f_out)
{
  ...
}
```

en plus de ces mêmes cinq étapes, on y retrouve la gestion des options du modèle d'écoulement choisi (écoulements seuls ou couplés) et la différenciation entre les modèles de Navier-Stokes en versions « standard » (`NS_STD`) ou en « compressibilité artificielle » avec les termes de couplages avec le modèle CAC (`NS_CAC`) :

```
{
  //if (flow_opt == "NS_STD")
  // CollideAndStream_NS_STD_Functor<dim, npop>::apply(params, f_in, f_out);
  ...
  if (flow_opt == "NS_CAC") {
    ...
    CollideAndStream_NS_CAC_Functor<dim, npop>::apply(params, f_in, f_out, data, fm,
                                                       force_pre, force_phi, cacParams, tag1);
  }
  swap_distribution(f_in, f_out);
}
```

Les étapes 2 et 4 sont deux appels à la même fonction `apply` de `CollideAndStream_NS_CAC_Functor` mais avec deux tags différents. Ce tag est fixé à `tag1` pour l'étape 2 (collision et déplacement) et `tag2` pour l'étape 4 (pour la mise à jour des moments d'ordre zéro et un). L'exécution des instructions des noyaux est réalisée par l'intermédiaire de la fonction `apply` qui est définie dans chaque noyau de calcul. Elle sera décrite dans le chapitre suivant consacré aux noyaux de calcul.

			Note Technique DES	Page 36/116
	Réf : STMF/LMSF/NT/2022-70869			
			Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.			

3.1.5 Gestion des entrées-sorties dans LBMRun.h : la classe configMap

Les données d'entrées de LBM_saclay sont lues dans un fichier au format ASCII avec l'extension `.ini`. Les instructions pour la lecture du fichier `.ini` proprement dit est effectué par la fonction `ini_parse` contenue dans le fichier `ini.cpp` (dans le répertoire `./src/utils/config/inih`). Des explications complémentaires de cette fonction peuvent être trouvées dans les commentaires du fichier `ini.h`.

Dans LBMRun.h, l'utilisation des données lues dans le fichier d'entrées se fait à l'aide d'une nouvelle classe, `configMap`, qui permet de gérer toutes les entrées-sorties du code. Cette classe est définie dans le fichier `ConfigMap.h` qui se trouve dans le répertoire `./src/utils/config`. Plusieurs fonctions sont associées à cette classe parmi lesquelles on trouve `getBool` et `getFloat` qui sont définies dans le fichier `ConfigMap.cpp`. Les fonctions `getInteger` et `getString` peuvent aussi être utilisées et elles sont définies dans le fichier `INIReader.cpp`. La classe `INIReader` est quant à elle définie dans le fichier `INIReader.h`.

Dans `update_navier_stokes` du fichier LBMRun.h, l'option `flow_opt` est récupérée via le `configMap` par la fonction `getString` :

```
const std::string flow_opt = configMap.getString("lbm", "flow_opt", "NS_STD");
```

où le premier argument "lbm" désigne la section [lbm] dans le fichier `.ini` et "flow_opt" la chaîne de caractères qui y est indiquée. Le dernier argument est la valeur par défaut qui est ici "NS_STD". Pour récupérer une valeur réelle et un booléen, on trouve les deux instructions suivantes :

```
const real_t visch = configMap.getFloat("lbm", "viscosity_high", 1.0);
const bool double_Poiseuille_enabled = configMap.getBool("lbm",
"double_poiseuille_enabled", false);
```

Le dernier argument est la valeur par défaut.

L'écriture des résultats dans les fichiers de sortie VTK (en mode MPI ou pas) est également gérée par une option lue dans le fichier d'entrée et accessible par cette classe.


Dans `saveData(int nstep)`, on retrouve l'appel à `configMap.getBool` :

```
bool vtk_enabled = configMap.getBool("output","vtk_enabled",true);
if (vtk_enabled) {
#ifdef USE_MPI
    saveVTK_mpi(data, data_h, params, configMap, nstep);
#else
    saveVTK(data, data_h, params, configMap, nstep);
#endif
}
```

Un `grep` permet de constater que la classe `configMap` apparaît dans plusieurs fichiers dont `LBMParams.cpp`, `LBMRunFactory.h`, `main.cpp`, `saveVTK.cpp`.

3.1.6 Gestion du parallélisme MPI dans LBMRun.h

Les communications MPI peuvent se faire sur les fonctions de distribution pour les problèmes physiques simples. Dans certains cas il est aussi utile de pouvoir communiquer certains champs macroscopiques, en particulier pour les problèmes couplés. La décomposition de domaine et les communications seront brièvement décrites dans le chapitre 5.

		Note Technique DES	Page 37/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

3.2 Les instructions relatives à la bibliothèque Kokkos

Les instructions relatives à la bibliothèque Kokkos se retrouvent principalement dans les noyaux de calcul qui sont regroupés dans le sous-répertoire `src/kernels`. Les instructions du noyau de calcul qui résout le modèle de Navier-Stokes standard ont été décrites dans la section 2.2.1 via la définition des types. Toutes ces instructions sont regroupées dans le fichier `CollideAndStream_NS_STD_Functor.h`.

3.2.1 La « décoration » KOKKOS_INLINE_FUNCTION

Dans ce fichier, on trouve la définition de la classe `class CollideAndStream_ns_std_Functor` et les trois fonctions qu'elle utilise : la fonction `sqr` qui retourne le carré d'un réel, le « functor » en 2d et le « functor » en 3d. La déclaration de ces fonctions est précédée de la « décoration » `KOKKOS_INLINE_FUNCTION`. Cette commande doit précéder la définition de nouvelles fonctions dont les instructions ont pour vocation d'être exécutées soit en CUDA ou en OpenMP ou en pthreads. On retrouve cette « décoration » dans la déclaration de toutes les fonctions qui sont regroupées dans les fichiers `.h` du répertoire `kernels`. Par exemple dans le fichier `CollideAndStream_PhaseField_Functor.h`, la fonction `sqr` qui renvoie le carré d'un réel est préalablement déclarée par la commande `KOKKOS_INLINE_FUNCTION` :

```
KOKKOS_INLINE_FUNCTION
real_t sqr(const real_t& value) const
{
    return value*value;
}
```

3.2.2 La méthode statique apply


On retrouve aussi dans tous les noyaux de calcul la méthode « `apply` » qui construit le noyau, calcule le nombre de nœuds du maillage et transmet le noyau à Kokkos (par l'intermédiaire de la commande `Kokkos::parallel_for`) pour exécution. Par exemple

Dans `CollideAndStream_NS_STD_Functor`, on trouve la définition de la fonction `apply` suivante

```
static void apply(LBMPParams params,
                 FArray f0,
                 FArray f1)
{
    const int nbCells = dim==2 ?
        params.isize*params.jsize : params.isize*params.jsize*params.ksize;
    CollideAndStream_ns_std_Functor functor(params, f0, f1);
    Kokkos::parallel_for(nbCells, functor);
}
```

où les arguments de la fonction `Kokkos::parallel_for` sont `nbCells` et `functor` qui sont préalablement définis aux deux lignes précédentes. L'exécution correspond à appeler la fonction « `void operator()` » décrite ci-dessous. La fonction `apply` est définie dans les noyaux de calcul mais elle est appelée par une fonction du fichier `LBMRun.h`. Par exemple

Dans `update_navier_stokes`, on trouve l'instruction :

		Note Technique DES	Page 38/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

```
if (flow_opt == "NS_STD")
CollideAndStream_NS_STD_Functor<dim, npop>::apply(params, f_in, f_out);
```

3.2.3 Les fonctions void operator()

Dans le fichier `CollideAndStream_NS_STD_Functor.h` on retrouve deux fonctions déclarées par `void operator()`. Toutes les fonctions `void operator()` doivent être préalablement « décorées » par `KOKKOS_INLINE_FUNCTION`. De plus, toutes les fonctions appelées par les fonctions `void operator()` doivent elles aussi être décorées par l'instruction `KOKKOS_INLINE_FUNCTION`. Dans le fichier `CollideAndStream_NS_STD_Functor.h` les deux fonctions `void operator()` regroupent toutes les instructions pour l'exécution de l'algorithme LBM standard qui a été décrit dans la section 2.1.1 (pour rappel : mise à jour des moments, calcul de la fonction d'équilibre et collision/déplacement de la fonction de distribution).

Dans `CollideAndStream_NS_STD_Functor.h`, il y a deux fonctions `void operator()` pour le même algorithme, mais la première fonction effectue ces étapes en 2D (option `dim_==2`) :

```
template<int dim_ = dim>
KOKKOS_INLINE_FUNCTION
void operator()(const typename Kokkos::Impl::enable_if<dim_==2, int>::type& index)
const
{
    ...
}
```


et la seconde en 3D (option `dim_==3`) :

```
template<int dim_ = dim>
KOKKOS_INLINE_FUNCTION
void operator()(const typename Kokkos::Impl::enable_if<dim_==3, int>::type& index)
const
{
    ...
}
```

Ces deux fonctions `void operator()` doivent être déclarées toutes les deux dans le noyau de calcul sinon la compilation échoue et l'exécutable n'est pas créé.

3.2.4 Les tags pour différencier les fonctions void operator()

Dans la fonction `update_navier_stokes` contenue dans le fichier `LBMRun.h`, on peut voir que la fonction `apply` est appelée trois fois : la première pour l'option `NS_STD` (if `(flow_opt == "NS_STD")` décrite ci-dessus) et les deux autres pour l'option `NS_CAC` (if `(flow_opt == "NS_CAC")`). Pour l'option `NS_CAC`, la fonction `apply` est appelée deux fois mais en utilisant deux tags différents. Le premier prend la valeur `TAG_COLLIDE_AND_STREAM` et le second prend la valeur `TAG_UPDATE_MACRO_VAR`. La fonction `apply` du fichier `CollideAndStream_NS_CAC_Functor.h` effectue les appels à `Kokkos::parallel_for` en différenciant ces deux tags. Puis, il y a deux fonctions `void operator()`, une pour chaque tag. Par exemple dans le fichier `CollideAndStream_NS_CAC_Functor.h` on retrouve la première fonction `void operator()` avec le premier tag (premier argument dans la parenthèse) :

		Note Technique DES	Page 39/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

```
template<int dim_ = dim>
KOKKOS_INLINE_FUNCTION
void operator()(const TagCollideAndStream&,
                const typename std::enable_if<dim_==2, int>::type& index) const
{
    ...
}
```

Cette fonction correspond au tag `TagCollideAndStream`. La seconde fonction `void operator()` contient les instructions qui correspondent au second tag `TagUpdateMacro` :

```
template<int dim_ = dim>
KOKKOS_INLINE_FUNCTION
void operator()(const TagUpdateMacro&,
                const typename std::enable_if<dim_==2, int>::type& index) const
{
    ...
}
```

L'exemple est donné pour les functors 2d. Dans le fichier `CollideAndStream_NS_CAC_Functor.h`, on trouve deux fonctions `void operator()` en plus, une pour chaque tag, et qui correspondent à la version 3d.

3.3 Autres fonctions

3.3.1 Les énumérations : fichier `LBM_enums.h`

Les énumérations sont définies dans le fichier `LBM_enums.h`. Elles sont définies par la commande `enum` et regroupent les paramètres constants représentatifs de la dimension spatiale 2D ou 3D (`DimensionType`); du réseau D2Q9, D3Q15, etc ... disponible (`LBM_Lattice_t`); du nombre de population (`Npop_t`); du module physique (`LBM_physics_t`); de l'indice du champ macroscopique (`ComponentIndex`); numéro d'indice d'une face en limite de domaine (`FaceIdType`); du type de condition à la limite qui y est associée (`BoundaryConditionType`); etc ... Par exemple l'énumération `LBM_Lattice_t` est définie par


```
enum LBM_Lattice_t {D2Q5, D2Q9, D3Q7, D3Q19, D3Q19};
```

L'énumération `LBM_physics_t` est définie par

```
enum LBM_physics_t { LBM_FLUID, LBM_TRANSPORT, LBM_HEAT_TRANSFER, LBM_PHASE_FIELD,
                    LBM_FLUID_PF_COUPLING };
```

Numéro du champ macroscopique

Les valeurs des champs macroscopiques (tels que la densité, vitesse, le champ de phase, la pression) sont stockées dans un tableau `lbm_data(i,j,k,INUM)` qui est de type `LBMArray` où `i`, `j` et `k` sont les indices de position et `INUM` est le numéro du champ qui peut prendre les valeurs de 0 à 5. Les correspondances entre ces numéros et les champs physiques sont énumérées dans `enum ComponentIndex {...}` du fichier `LBM_enums.h` par l'intermédiaire des indices `ID=0` (ρ), `IU=1` (u_x), `IV=2` (u_y), `IW=3` (u_z), `IPHI=4` (ϕ) et `IP=5` (p). Ainsi, lorsqu'on veut utiliser la densité, on manipulera `lbm_data(i,j,k,fm[ID])` où `fm` est un alias pour obtenir un indice à partir d'un `enum` :

		Note Technique DES	Page 40/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.			

il est de type `id2index_t` qui est défini dans `FieldManager.h`. Si on souhaite stocker les valeurs de nouveaux champs macroscopiques (e.g. le potentiel chimique ou la température), on ajoutera des indices supplémentaires dans cette énumération (voir chapitre 6 pour le potentiel chimique).

Énumération des conditions initiales

Dans ce fichier se trouvent également les énumérations des conditions initiales `PhaseFieldInit`, `FluidInit` et `ProblemType`, qui sont relatives au champ de phase, au Navier-Stokes et au transport par advection diffusion. Dans l'énumération `PhaseFieldInit` on trouve les chaînes de caractères suivantes :

```
enum PhaseFieldInit { PHASE_FIELD_INIT_UNDEFINED, UNIF, TANH, ZALESK, SHRF_VORTEX };
```

Si on souhaite développer un nouveau cas test qui implique l'équation du champ de phase, le mot clé qui correspond au cas test sera à rajouter dans cette énumération. Il en est de même pour les nouveaux cas tests d'un problème physique de fluide ou de transport.

Utilisation en argument de fonctions

Ces énumérations peuvent être utilisées en argument de fonctions. Par exemple la fonction `make_boundary` définie dans le fichier `LBMRun.h` :

```
template<int dim,
        int npop>
void LBMRun<dim, npop>::make_boundary(FArray3d fdata,
                                     FaceIdType faceId, BoundaryConditionType bc_type) {...}
```

est déclarée par trois arguments. Le premier est de type `FArray3d` et les deux autres sont des énumérations `FaceIdType` et `BoundaryConditionType` qui sont définies dans le fichier `LBM_enums.h`.

Dans la fonction `copy_boundaries` (toujours dans le fichier `LBMRun.h`) :


```
template<int dim,
        int npop>
void LBMRun<dim, npop>::copy_boundaries(FArray3d fdata, Direction dir) {...}
```

le second est une énumération sur les directions x , y et z (`Direction`).

3.3.2 Gestion des champs macroscopiques : fichier `FieldManager.h`


Le nombre de champs macroscopiques à garder en mémoire et à écrire dans les fichiers de sortie dépend du problème physique et des indications contenues dans le jdd. Par exemple un simple problème diffusif ne contient qu'un seul champ scalaire, la concentration C , tandis qu'un problème d'ébullition (voir [5]) couple plusieurs EDPs (NS, CAC, thermique) et contient plusieurs champs (ρ , \mathbf{u} , T , ϕ , μ_ϕ , etc ...). Parmi eux, l'utilisateur peut s'intéresser qu'à un nombre réduit et l'indiquer dans le jdd pour éviter de générer un gros volume de fichiers de sortie.


C'est le fichier `FieldManager.h` qui est contenu dans le répertoire `src` qui se charge de la gestion des champs. On retrouve dans ce fichier les indices des champs macroscopiques pour la densité ($\rho \equiv \text{ID}$), pour les composantes de la vitesse ($u_x \equiv \text{IU}$, $u_y \equiv \text{IV}$, $u_z \equiv \text{IW}$), $\phi \equiv \text{IPHI}$ pour le champ de phase et pour la pression ($p \equiv \text{IP}$). Lorsque l'option de résolution d'un problème de champ de phase est demandée dans le jdd (`lbm_phase_field_enabled`) alors le nombre de champs est incrémenté `var_enabled[IPHI] = 1`. Il en est de même pour les autres problèmes

		Note Technique DES	Page 41/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

physiques. Ensuite il y a des fonctions qui établissent le lien entre le numéro de ces champs et des chaînes de caractères pour les sorties `map[IPHI] = "phi"` et inversement `map["phi"] = IPHI`.

Lorsqu'on crée un nouveau champ physique qu'on souhaite garder en mémoire et sauvegarder en sortie, il est nécessaire de le rajouter dans ce fichier `FieldManager.h`. On montrera les instructions à ajouter dans ce fichier et dans `LBM_enums.h` sur l'exemple du potentiel chimique $\mu_\phi(\mathbf{x}, t)$ dans le chapitre 6 (section 6.2.3).

		Note Technique DES	Page 42/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

		Note Technique DES	Page 43/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Chapitre 4

Description des principaux noyaux de calculs


4.1 Introduction

Les noyaux de calcul de LBM_saclay sont des fichiers avec l'extension « .h » qui sont regroupés dans le sous-répertoire `src/kernels`. Les principaux noms de ces fichiers sont précédés des noms « `CollideAndStream` », « `Compute` », « `Compute_Feq` », « `LBM_kernels` » et se terminent par « `functor.h` ».

Les cinq fichiers « `CollideAndStream` » sont les principaux noyaux de calcul qui regroupent toutes les instructions relatives à la simulation d'une EDP par la méthode LB. Ainsi, les instructions relatives au calcul de *i*) la fonction de distribution à l'équilibre, *ii*) à l'étape de collision, *iii*) de déplacement et du *iv*) calcul des moments (d'ordre zéro et un), sont toutes regroupées dans un unique fichier dont le nom commence par la chaîne de caractères « `CollideAndStream` ». Cette chaîne de caractères est suivie des extensions *a*) « `_NS_CAC` », ou *b*) « `NS_STD` » ou *c*) « `_PhaseField` » ou *d*) « `_transport` » pour préciser les modèles mathématiques résolus c'est-à-dire respectivement *a*) au modèle couplé « Navier-Stokes/Conservative Allen-Cahn », *b*) l'équation de « Navier-Stokes STandard », *c*) l'« équation du champ de phase » ou *d*) l'équation du transport par advection-diffusion.

Dans le répertoire, on trouve aussi des fichiers séparés pour les étapes *i*), puis *ii*), *iii*), et *iv*). Les fonctions contenues dans ces fichiers sont appelées ponctuellement par `LBMRun.h`, par exemple pour l'initialisation ou le calcul du moment. En effet, pour l'initialisation, après la lecture des champs de densité et de vitesses, la fonction d'équilibre peut être calculée en faisant appel à la fonction contenue dans le fichier « `Compute_Feq_NS_STD_Functor.h` » qui calcule la fonction d'équilibre des équations de Navier-Stokes STandard (`NS_STD`). De la même manière le fichier « `Compute_Moment_Order0_Functor.h` » ne contient que les instructions pour le calcul du moment d'ordre zéro : la densité pour un problème de Navier-Stokes, ou le champ de phase, ou la concentration pour l'ADE. Enfin le fichier « `Collision_BGK_Functor.h` » contient les instructions relatives à l'étape de collision par l'opérateur BGK.

On décrit dans ce chapitre le contenu de trois fichiers « `CollideAndStream` ». L'ordre de description des trois fichiers est choisi pour des raisons pédagogiques. On présente tout d'abord dans la section 4.2 l'algorithme LBM pour la simulation de l'équation du transport par advection-diffusion (« `CollideAndStream_transport_Functor.h` »). Ensuite la section 4.3 décrit le modèle CAC (« `CollideAndStream_PhaseField_Functor.h` ») qui est une généralisation de celle du transport, c'est-à-dire une équation scalaire de type « advection-diffusion » mais avec un terme en plus dans le terme de divergence qui rend l'équation non linéaire. Ce terme nécessite le calcul de gradients additionnels et on montrera comment les calculer en tenant compte de toutes les directions du réseau. Ensuite on présente dans la section 4.4 le modèle couplé de NS/CAC (« `CollideAndStream_NS_CAC_Functor.h` ») en modifiant l'équilibre pour qu'elle permette de simuler les équations NS en « compressibilité artificielle » et en rajoutant un terme force.

		Note Technique DES	Page 44/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Dans chacune de ces sections, on rappelle le modèle mathématique continu, puis l'algorithme LBM associé. Ensuite on extrait des fichiers en C++ les lignes permettant de faire le lien avec l'algorithme LBM. Chacun des fichiers contient les instructions pour la simulation du modèle en 2D, suivi des instructions pour la simulation en 3D. On ne décrit ici que la partie 2D, sachant que la partie 3D généralise la version 2D en ajoutant une dimension spatiale et une composante de plus pour les vecteurs. En plus du vecteur vitesse, les composantes des directions de déplacement sur le réseau et les termes de « forçage LBM » ont une dimension de plus.

4.2 Équation du transport par advection-diffusion

La section 4.2.1 rappelle l'équation continue du transport par Advection-Diffusion avec un terme source S_c . L'algorithme LBM est décrit dans la section 4.2.2 lorsque ce terme source est nul et dans la section 4.2.3 lorsqu'il est non nul.

4.2.1 Rappel de l'équation continue

L'équation du transport par advection-diffusion (Advection-Diffusion Equation – ADE) s'écrit

$$\frac{\partial C}{\partial t} + \nabla \cdot (\mathbf{u}C) = \nabla \cdot (\mathcal{D}\nabla C) + S_c \quad (4.1)$$

où $C \equiv C(\mathbf{x}, t)$ est la concentration, la vitesse $\mathbf{u} \equiv \mathbf{u}(\mathbf{x}, t)$ est la solution obtenue du « Navier-Stokes » et \mathcal{D} est le coefficient de diffusion. Il s'agit d'une équation de conservation de la forme $\partial C/\partial t = -\nabla \cdot \mathbf{J}_{tot}$ où \mathbf{J}_{tot} est le flux total défini par la somme du flux advectif $\mathbf{j}_{adv} = \mathbf{u}C$ et du flux diffusif $\mathbf{j}_{diff} = -\mathcal{D}\nabla C$. Le second terme du membre de droite $S^C \equiv S^C(\mathbf{x}, t)$ est un terme source s'il est positif ou un terme puits s'il est négatif.

4.2.2 Algorithme LBM pour l'ADE avec $S_C = 0$

En introduisant une nouvelle fonction de distribution h_i l'algorithme LBM pour simuler les Eq. (4.1) s'écrit :

$$h_i(\mathbf{x} + \mathbf{c}_i\delta t, t + \delta t) = h_i(\mathbf{x}, t) - \frac{1}{\tau_h} [h_i(\mathbf{x}, t) - h_i^{eq}(\mathbf{x}, t)] \quad (4.2)$$

avec la fonction de distribution à l'équilibre définie par :

$$h_i^{eq} = Cw_i \left[1 + \frac{\mathbf{c}_i \cdot \mathbf{u}}{c_s^2} \right] \quad (4.3)$$

Le coefficient de mobilité \mathcal{D} est relié au taux de collision τ_h par la relation :


$$\mathcal{D} = \frac{1}{3} \left(\tau_h - \frac{1}{2} \right) \frac{\delta x^2}{\delta t}, \quad (4.4)$$

et le moment d'ordre zéro est mis à jour par :

$$C = \sum_i h_i \quad (4.5)$$

Dans `CollideAndStream_transport_Functor.h`, le fichier regroupe les instructions qui permettent de simuler l'ADE. Tout d'abord, le moment d'ordre zéro est mis à jour par l'instruction :

```
for (int ipop=0; ipop<npop; ++ipop) {
    rho += f[ipop];
}
```

		Note Technique DES	Page 45/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Ensuite, ρ (nom de variable qui représente C) est utilisé dans le calcul de la fonction de distribution à l'équilibre feq (Eq. (4.3)) :

```
for (int ipop=0; ipop<npop; ++ipop) {
    real_t scal = dx/dt*(E[ipop][IX]*vx + E[ipop][IY]*vy);
    feq[ipop] = w[ipop] * rho * (1.0 + w1*scal);
}
```

Le taux de collision τ_h est appelé par l'instruction :

```
real_t tau_inv = 1.0/params.settings.tau;
```

Les instructions relatives aux étapes de déplacement et de collision qui utilisent feq et τ_{inv} sont indiquées par :

```
for (int ipop=0; ipop<npop; ++ipop) {
    if (i+E[ipop][IX] >= 0 and i+E[ipop][IX] < params.isize and
        j+E[ipop][IY] >= 0 and j+E[ipop][IY] < params.jsize) {
        f1(i+E[ipop][IX], j+E[ipop][IY], ipop) = f[ipop] - tau_inv * ( f[ipop] -
        feq[ipop] );
    }
}
```

4.2.3 Algorithme LBM pour l'ADE avec $S_C \neq 0$

Lorsque le terme source est non nul, on utilise l'astuce du changement de variable afin de conserver une précision d'ordre deux en temps. La précision d'ordre deux est obtenue grâce à l'intégration par la méthode des trapèzes du terme de collision d'équation de Boltzmann continue. On pourra se référer à l'annexe B pour comprendre ce changement de variable.

Lorsque $S_C \neq 0$ alors l'algorithme LBM doit être modifié de la manière suivante :

$$\bar{h}_i(\mathbf{x} + \mathbf{c}_i \delta t, t + \delta t) = \bar{h}_i(\mathbf{x}, t) - \frac{1}{\bar{\tau}_h + 1/2} \left[\bar{h}_i(\mathbf{x}, t) - \bar{h}_i^{eq}(\mathbf{x}, t) \right] + \mathcal{S}_i^C \delta t \quad (4.6)$$

où le terme source \mathcal{S}_i^C dans l'Eq. (4.6) est simplement défini par :


$$\mathcal{S}_i^C = w_i S^C \quad (4.7)$$

La fonction de distribution à l'équilibre \bar{h}_i^{eq} est définie par :

$$\bar{h}_i^{eq} = h_i^{eq} - \frac{\delta t}{2} \mathcal{S}_i^C \quad (4.8)$$

où h_i^{eq} est l'équilibre définie par l'Eq. (4.3). La notation \bar{h}_i et \bar{h}_i^{eq} traduit le changement de variable défini par l'Eq. (4.8) et qui est utilisé comme une astuce pour conserver une équation d'évolution similaire à celle décrite dans la section 4.2.2. Le coefficient de mobilité \mathcal{D} est relié au taux de collision $\bar{\tau}_h$ par la relation :

$$\mathcal{D} = \frac{1}{3} \bar{\tau}_h \frac{\delta x^2}{\delta t}, \quad (4.9)$$

		Note Technique DES	Page 46/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

et le moment d'ordre zéro est mis à jour en tenant compte du changement de variable :

$$C = \sum_i \bar{h}_i + \frac{\delta t}{2} \sum_i \mathcal{S}_i^C \quad (4.10)$$

Du côté de la programmation, il faut ajouter le terme $\delta t w_i S^C$ dans l'étape de collision, tenir compte du terme $-\delta t \mathcal{S}_i^C/2$ après le calcul de l'équilibre (Eq. (4.8)) et modifier le calcul du moment d'ordre zéro (Eq. (4.10)). On notera la nouvelle dépendance du coefficient de diffusion \mathcal{D} avec $\bar{\tau}_h$.

Un exemple C++ de prise en compte de termes sources sera présenté dans la section 4.4 pour le Navier-Stokes en version « compressibilité artificielle ».

4.3 Équation du champ de phase

Plusieurs types d'équations « à champ de phase » existent dans la littérature. La plus connue d'entre elles est celle de Cahn-Hilliard qui est très utilisée pour les problèmes de séparation de phase et les mélanges binaires. De nombreux travaux existent qui couplent les équations de Navier-Stokes à celle de Cahn-Hilliard. D'autres équations de type « champ de phase » apparaissent dans le domaine de la solidification et de la croissance cristalline. On montrera dans les tutoriels des chapitres 6 et 7 comment procéder pour développer dans LBM_saclay ces modèles. Dans ce chapitre on décrit une autre « équation à champ de phase », celle de Allen-Cahn en version conservative. Plusieurs travaux récents utilisent cette équation couplée avec celles de Navier-Stokes pour simuler des problèmes diphasiques avec suivi d'une interface. L'équation continue est rappelée dans la section 4.3.1 et l'algorithme LBM dans la section 4.3.2. Le noyau de calcul `CollideAndStream_PhaseField_Functor.h` est ensuite décrit dans la section 4.3.3.

4.3.1 Rappel de l'équation continue

L'équation du modèle « conservatif de Allen-Cahn » s'écrit :

$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\mathbf{u}\phi) = \nabla \cdot \left[M_\phi \left(\nabla \phi - \frac{4}{\xi} \phi(1-\phi) \mathbf{n} \right) \right] \quad (4.11)$$

Dans l'Eq. (4.11), $\phi \equiv \phi(\mathbf{x}, t)$ est le « champ de phase ». Il s'agit d'une indicatrice de phase qui prend la valeur 0 dans la première phase et 1 dans la seconde. Ses valeurs varient de façon continue entre ses deux valeurs extrêmes. L'interface est diffuse entre les deux phases et caractérisée par son épaisseur ξ et sa mobilité M_ϕ . La vitesse \mathbf{u} est la solution obtenue du « Navier-Stokes » et \mathbf{n} est le vecteur unitaire normal à l'interface. Il est défini par :


$$\mathbf{n} = \frac{\nabla \phi}{|\nabla \phi|} \quad (4.12)$$

L'Eq. (4.11) ressemble à l'équation d'advection-diffusion de la section précédente. Les instructions en C++ de l'algorithme LBM reprennent la même base que celle du transport mais en tenant compte en plus du terme non linéaire $-4\phi(1-\phi)\mathbf{n}/\xi$ dans la divergence et du calcul de la normale en l'interface \mathbf{n} .

4.3.2 Algorithme LBM associé

En introduisant une nouvelle fonction de distribution g_i l'algorithme LBM pour simuler les Eq. (4.11) et (4.12) s'écrit :

$$g_i(\mathbf{x} + \mathbf{c}_i \delta t, t + \delta t) = g_i(\mathbf{x}, t) - \frac{1}{\tau_g} [g_i(\mathbf{x}, t) - g_i^{eq}(\mathbf{x}, t)] \quad (4.13)$$

		Note Technique DES	Page 47/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

avec la fonction de distribution à l'équilibre définie par :

$$g_i^{eq, CAC} = \phi w_i \left[1 + \frac{\mathbf{c}_i \cdot \mathbf{u}}{c_s^2} \right] + w_i M_\phi \left[\frac{4}{\xi} \phi (1 - \phi) \right] \frac{(\mathbf{c}_i \cdot \mathbf{n})}{c_s^2} \quad (4.14)$$

La fonction à l'équilibre est séparée en deux termes distincts, le premier contient le produit scalaire $\mathbf{c}_i \cdot \mathbf{u}/c_s^2$ et le second contient le produit scalaire $\mathbf{c}_i \cdot \mathbf{n}/c_s^2$. Le premier terme qui contient le produit scalaire $\mathbf{c}_i \cdot \mathbf{u}/c_s^2$ est identique au terme de la fonction d'équilibre h_i^{eq} (Eq. (4.3)). Le crochet est multiplié par ϕ (à la place de C) qui est le nouveau champ.

Le coefficient de mobilité M_ϕ est relié au taux de collision τ_g par la relation :

$$M_\phi = \frac{1}{3} \left(\tau_g - \frac{1}{2} \right) \frac{\delta x^2}{\delta t}, \quad (4.15)$$

et le moment d'ordre zéro est mis à jour par :

$$\phi = \sum_i g_i \quad (4.16)$$

Le vecteur unitaire normal \mathbf{n} défini par l'Eq. (4.12) est calculé en utilisant la méthode des dérivées directionnelles adaptées au réseau choisi :

$$\mathbf{e}_i \cdot \nabla \phi|_{\mathbf{x}} = \frac{1}{2\delta x} [\phi(\mathbf{x} + \mathbf{e}_i \delta x) - \phi(\mathbf{x} - \mathbf{e}_i \delta x)] \quad (4.17a)$$

$$\nabla \phi|_{\mathbf{x}} = \frac{1}{e^2} \sum_{i=0}^{N_{pop}} w_i \mathbf{e}_i (\mathbf{e}_i \cdot \nabla \phi|_{\mathbf{x}}) \quad (4.17b)$$

L'Eq. (4.17a) calcule les dérivées de ϕ pour toutes les directions du réseau choisi. L'Eq. (4.17b) utilise toutes les dérivées directionnelles pour calculer les composantes du gradient de ϕ .


4.3.3 Fichier `CollideAndStream_PhaseField_Functor.h`

Le fichier « `CollideAndStream_PhaseField_Functor.h` » contient les instructions pour simuler les Eqs. (4.11) et (4.12). Ces instructions relatives 1) au calcul du vecteur \mathbf{n} (Eq. (4.12)) par la méthode des dérivées directionnelles (Eqs. (4.17a)-(4.17b)), 2) au calcul de la fonction de distribution à l'équilibre g_i^{eq} (Eq. (4.14)) et 3) les étapes de collision/déplacement (Eq. (4.13)).

4.3.3.1 Calcul du vecteur unitaire normal à l'interface \mathbf{n}

Le calcul du vecteur normal \mathbf{n} est réalisé par la fonction « `compute_normal_vector (...)` » où (...) représente la liste des arguments. La fonction regroupe les instructions pour les Eqs. (4.17a) et (4.17b) dans deux boucles séparées. La première calcule les dérivées directionnelles (Eq. (4.17a)) dans le conteneur de données `dPhidir` qui a le type `FState` car il possède les mêmes dimensions qu'une fonction de distribution f_i de la LBM : le membre de gauche de l'Eq. (4.17a) est une fonction scalaire indexée par i . Les dérivées directionnelles sont calculées par les instructions :

Dans `compute_normal_vector (...)`, les dérivées directionnelles sont calculées par :

		Note Technique DES	Page 48/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

```
// compute directional derivatives of phi
FState dPhidir;
for (int ipop=0; ipop<npop; ++ipop) {
    const int i0 = E[ipop][IX];
    const int j0 = E[ipop][IY];
    dPhidir[ipop] = ( lbm_data(i+i0,j+j0,fm[IPHI]) -
                     lbm_data(i-i0,j-j0,fm[IPHI]) ) / (2*dx);
}
```

Ensuite, les sommes pour obtenir chaque composante du gradient (Eq. (4.17b)) sont faites par les instructions :

```
// compute LBM gradients
for (int ipop=1; ipop<npop; ++ipop) {
    dPhidx += w[ipop]*E[ipop][IX]*dPhidir[ipop];
    dPhidy += w[ipop]*E[ipop][IY]*dPhidir[ipop];
}
dPhidx *= e2;
dPhidy *= e2;
```


Une fois que les composantes du gradient sont connues, on calcule la norme du gradient $|\nabla\phi|$, puis les composantes n_x et n_y de \mathbf{n} sont obtenues en divisant les composantes du gradient par sa norme. Les composantes du vecteur unitaire $\mathbf{n} = (n_x, n_y)^T$ sont notées dPhidx pour n_x et dPhidy pour n_y :

```
real_t norm = sqrt( sqr(dPhidx) + sqr(dPhidy) );
// normalize phi gradient to have surface normal vector
if (norm > 0.0) {
    dPhidx /= norm;
    dPhidy /= norm;
}
```

4.3.3.2 Utilisation dans la fonction à l'équilibre

La fonction à l'équilibre g_i^{eq} (Eq. (4.14)) diffère de celle de l'équation du transport ADE h_i^{eq} (Eq. (4.3)) par l'ajout d'un terme supplémentaire qui contient le produit scalaire $\mathbf{c}_i \cdot \mathbf{n}$. Les deux produits scalaires $\mathbf{c}_i \cdot \mathbf{u}$ et $\mathbf{c}_i \cdot \mathbf{n}$ sont respectivement notés `scal1` et `scal2`. et sont calculés selon la même instruction que celle déjà décrite pour $\mathbf{c}_i \cdot \mathbf{u}$ dans la section 2.2.1.

Dans `void operator()`, les instructions qui programment la fonction de distribution à l'équilibre g_i^{eq} (Eq. (4.14)) sont :

		Note Technique DES	Page 49/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.			

```

for (int ipop=0; ipop<npop; ++ipop) {
    f[ipop] = f0(i,j, ipop);
    real_t scal1 = dx/dt*(E[ipop][IX]*vx + E[ipop][IY]*vy);
    real_t scal2 = dx/dt*(E[ipop][IX]*dPhidx + E[ipop][IY]*dPhidy);
    feq[ipop] = w[ipop] * phi * (1.0 + w1*scal1) +
                w[ipop] * w1 * work * scal2;
}

```

Le taux de collision est obtenu en inversant la relation (4.15) :

```

real_t tau_phi = 0.50 + (3.0 * mobility * dt / (dx * dx));

```

Les instructions des étapes de collision et de déplacement sont :

```

for (int ipop=0; ipop<npop; ++ipop) {
    if (i+E[ipop][IX] >= 0 and i+E[ipop][IX] < params.isize and
        j+E[ipop][IY] >= 0 and j+E[ipop][IY] < params.jsize) {
        f1(i+E[ipop][IX],
            j+E[ipop][IY],ipop) = f[ipop] - ( f[ipop] - feq[ipop] ) / tau_phi;
    }
}

```

4.4 Modèle de Navier-Stokes/Conservative Allen-Cahn

Le fichier « CollideAndStream_NS_CAC_Functor.h » regroupe les instructions pour la simulation du modèle de Navier-Stokes (NS). La chaîne de caractère `_CAC_` indique que les couplage qui impliquent de ϕ sont pris en compte dans ce noyau. Le calcul de ϕ est réalisé par le modèle Conservatif de Allen-Cahn (CAC) qui est le même que celui décrit dans la section précédente (Eq. (4.11) avec \mathbf{n} défini par l'Eq. (4.12)). La particularité des équations NS décrites dans cette section, sont qu'elles sont formulées en version « compressibilité artificielle » dans laquelle les deux inconnues sont la pression p et la vitesse \mathbf{u} . On rappelle que dans le chapitre 2, l'algorithme standard calcule la densité ρ et la vitesse \mathbf{u} et la pression est liée à la densité par la loi d'état. Le couplage entre les équations de NS et celle de CAC est explicite : le noyau de calcul `CollideAndStream_NS_CAC_Functor.h` calcule la vitesse \mathbf{u} et la transmet au noyau `CollideAndStream_PhaseField_Functor.h`. Ce dernier calcule ϕ et le transmet au premier.

4.4.1 Rappel du modèle NS en « compressibilité artificielle »


Les deux équations du modèle de Navier-Stokes en version « compressibilité artificielle » s'écrivent :

$$\frac{1}{\beta} \frac{\partial p}{\partial t} + \nabla \cdot \mathbf{u} = 0 \quad (4.18a)$$

$$\frac{\partial \rho_0(\phi) \mathbf{u}}{\partial t} + \nabla \cdot [\rho_0(\phi) \mathbf{u} \mathbf{u}] = -\nabla p + \nabla \cdot [\eta_0 (\nabla \mathbf{u} + \nabla \mathbf{u}^T)] + \mathbf{F} \quad (4.18b)$$

Dans les équations (4.18a) et (4.18b), $\rho_0(\phi)$ est la densité interpolée entre la densité ρ_L (indice L pour « Low ») et ρ_H (indice H pour « High ») :

$$\rho_0(\phi) = \phi \rho_L + (1 - \phi) \rho_H. \quad (4.18c)$$

		Note Technique DES	Page 50/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Le vecteur vitesse est noté $\mathbf{u} \equiv \mathbf{u}(\mathbf{x}, t)$ et η_0 est la viscosité dynamique du fluide, produit de la viscosité cinématique ν par la densité ρ_0 . Dans l'Eq. (4.18a), β est le coefficient de compressibilité artificielle qui vaut $\beta = \rho_0 c_s^2$ et dans l'équation (4.18b) \mathbf{F} est un terme force externe qui est supposé indépendant de ϕ .

Comparé aux équations standards du chapitre 2, le modèle diffère donc sur deux points principaux : 1) la prise en compte de la pression p dans l'Eq. (4.18a) comme variable principale de calcul à la place de ρ et 2) la prise en compte d'une force externe \mathbf{F} dans la conservation de la quantité de mouvement (Eq. (4.18b)). On notera également la dépendance de la densité avec l'indicatrice de phase ϕ .

4.4.2 Algorithme LBM avec prise en compte d'un terme force

Pour tenir compte de ces différences, la méthode LB est modifiée par l'ajout d'un terme force microscopique \mathcal{F}_i dans l'équation d'évolution :

$$\bar{f}_i(\mathbf{x} + \mathbf{c}_i \delta t, t + \delta t) = \bar{f}_i(\mathbf{x}, t) - \frac{1}{\bar{\tau}_f + 1/2} \left[\bar{f}_i(\mathbf{x}, t) - \bar{f}_i^{eq}(\mathbf{x}, t) \right] + \mathcal{F}_i \delta t, \quad (4.19)$$

et par une redéfinition de la fonction à l'équilibre intermédiaire f_i^{eq} pour tenir compte de p :

$$f_i^{eq}(\mathbf{x}, t) = w_i \left[p + \rho_0(\phi) c_s^2 \left(\frac{\mathbf{c}_i \cdot \mathbf{u}}{c_s^2} + \frac{(\mathbf{c}_i \cdot \mathbf{u})^2}{2c_s^4} - \frac{\mathbf{u} \cdot \mathbf{u}}{2c_s^2} \right) \right]. \quad (4.20)$$

Le terme force \mathcal{F}_i est défini par :

$$\mathcal{F}_i = (\mathbf{c}_i - \mathbf{u}) \cdot [(\Gamma_i - w_i) \nabla \rho_0(\phi) c_s^2 + \Gamma_i \mathbf{F}] \quad (4.21)$$

Dans l'Eq. (4.21), la fonction Γ_i est définie par l'Eq. (2.4), les \mathbf{c}_i sont les vitesses du réseau, w_i sont les poids associés à chaque réseau et la force macroscopique externe est notée \mathbf{F} . Comme dans la section 4.2.3, l'ajout du terme $\delta t \mathcal{F}_i$ dans l'Eq. (4.19) nécessite d'effectuer un changement de variable de la fonction d'équilibre par la relation :

$$\bar{f}_i^{eq} = f_i^{eq} - \frac{\delta t}{2} \mathcal{F}_i \quad (4.22)$$

et les moments d'ordre zéro et un doivent être corrigés par les relations suivantes :


$$\rho_0 \mathbf{u} = \frac{1}{c_s^2} \sum_i \bar{f}_i \mathbf{c}_i + \frac{\delta t}{2} \mathbf{F} \quad (4.23a)$$

$$p = \sum_i \bar{f}_i + \frac{\delta t}{2} \mathbf{u} \cdot \nabla \rho_0(\phi) c_s^2 \quad (4.23b)$$

La correction par le produit scalaire $\mathbf{u} \cdot \nabla \rho_0(\phi) c_s^2$ dans l'Eq. (4.23b) s'explique par le fait que les développements de Chapman-Enskog conduisent à une équation de la forme $\partial p / \partial t + \nabla \cdot [\rho_0(\phi) c_s^2 \mathbf{u}] = 0$. La façon la plus simple d'obtenir l'Eq. (4.18a) est d'ajouter un terme source de la forme $\mathbf{u} \cdot \nabla \rho_0(\phi) c_s^2$ dans son membre de droite, terme qui viendra s'annuler avec l'un des deux termes issus du développement de la divergence :

$$\frac{\partial p}{\partial t} + \nabla \cdot [\rho_0(\phi) c_s^2 \mathbf{u}] = \mathbf{u} \cdot \nabla \rho_0(\phi) c_s^2 \quad (4.24)$$

Ce terme source est pris en compte dans la mise à jour du calcul du moment d'ordre zéro. Le facteur $\delta t/2$ provient de l'intégration en temps par la méthode des trapèzes de l'équation continue de Boltzmann (voir l'annexe B pour le détail des calculs). La correction par le terme $\mathbf{F} \delta t/2$ dans l'Eq. (4.23a) suit la même logique.

		Note Technique DES	Page 51/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

4.4.3 Fichier CollideAndStream_NS_CAC_Functor.h

La force microscopique F_i (Eq. (4.21)) et les moments d'ordre zéro et un (resp. (4.23a) et (4.23b)) font intervenir deux types de forces, la force externe \mathbf{F} et la force $\nabla \rho_0(\phi) c_s^2$ qui sera appelée « force de pression ». Ces deux forces sont prises en comptes en allouant deux tableaux de type `LBMArray` dans `LBMRun.h` :

```
// forces
if (params.lbm_fluid_enabled and
    params.lbm_phase_field_enabled) {
  if (dim==2) {
    force_pre = LBMArray("pressure forces", isize, jsize, dim);
    force_phi = LBMArray("capilarity forces", isize, jsize, dim);
  } else if (dim==3) {
    force_pre = LBMArray("pressure forces", isize, jsize, ksize, dim);
    force_phi = LBMArray("capilarity forces", isize, jsize, ksize, dim);
  }
}
```

Les instructions du modèle couplé sont définies par plusieurs fonctions qui sont toutes regroupées dans le fichier « `CollideAndStream_NS_CAC_Functor.h` ». Le fichier `Compute_Feq_NS_Incompressible_Functor.h` regroupe également les instructions du calcul de la fonction de distribution en version « compressibilité artificielle ». On ne décrit ici que les étapes relatives à la prise en compte de la force microscopiques et corrections dans le calcul des moments d'ordre zéro et un. Toutes les autres étapes (calcul de la fonction Γ_i , calcul de la fonction d'équilibre f_i^{eq} , etc ...) sont identiques aux noyaux de calcul déjà décrits.

Le calcul de la force microscopique F_i est réalisée par les instructions suivantes :

```
// compute microforce
FState microforce;
for (int ipop=0; ipop<npop; ++ipop) {
  real_t termX = (gamma[ipop] - w[ipop]) * fxPre + gamma[ipop] * fxPhi;
  real_t termY = (gamma[ipop] - w[ipop]) * fyPre + gamma[ipop] * fyPhi;
  microforce[ipop] =
    termX * (dx/dt*E[ipop][IX] - vx) +
    termY * (dx/dt*E[ipop][IY] - vy) ;
} // end compute microforce
```

Le calcul de la fonction de distribution à l'équilibre f_i^{eq} (Eq. (4.20)) se fait sur le même schéma que des noyaux précédents :

```
real_t scal = dx/dt*(E[ipop][IX]*vx + E[ipop][IY]*vy);
feq[ipop] = w[ipop] * ( pressure + rho*cs*cs * ( w1*scal + w2*scal*scal - w3*vv ) );
```

et le changement de variable pour l'équilibre \bar{f}_i^{eq} (Eq. (4.22)) est réalisé par l'instruction :

```
feq[ipop] -= 0.5*dt*microforce[ipop];
```

Fonctions externes pour le calcul de \mathbf{u} , p , $\rho_0(\phi)$ et $\nabla \rho_0(\phi) c_s^2$

Contrairement aux noyaux de calcul des sections 4.2 et 4.3, la mise à jour des moments d'ordre zéro (Eq. (4.23b)) et d'ordre un (Eq. (4.23a)) est réalisée par deux fonctions définies en externe : `compute_pressure` et `compute_velocity` et qui sont déclarées par les lignes :

```
KOKKOS_INLINE_FUNCTION
void compute_velocity (...) const
{
    ...
}
```

```
KOKKOS_INLINE_FUNCTION
real_t compute_pressure (...) const
{
    ...
}
```

où « (...) » représente la liste des arguments et « ... » la liste des instructions des deux fonctions. Ces deux fonctions sont appelées par la fonction `apply` de `CollideAndStream_NS_CAC_Functor` (elle-même appelée par `update_navier_stokes` dans le fichier `LBMRun.h`) avec un tag fixé à `tag2`. Pour la pression les instructions au cœur de la fonction sont :

```
real_t scal = vx*fx + vy*fy;
for (int ipop=0; ipop<npop; ++ipop) {
    pressure += f[ipop];
}
pressure += 0.5*dt*scal;
```


L'instruction à l'intérieur de la boucle est classique, elle réalise la somme sur i des fonctions de distribution \bar{f}_i . La toute dernière ligne tient compte du produit scalaire $\text{scal} \ (\delta t/2) \mathbf{u} \cdot \nabla \rho_0(\phi) c_s^2$ dans l'Eq. (4.23b). Les composantes `fx` et `fy` de la force ont préalablement été calculées par la fonction `compute_pressure_force` qui calcule les gradient de $\nabla \rho(\phi) c_s^2$ par la méthode des dérivées directionnelles (méthode déjà introduite dans la section 4.3.2) :

```
KOKKOS_INLINE_FUNCTION
void compute_pressure_force (...) const {
    ...
}
```

Cette fonction fait appel à la fonction `compute_density` pour la mise à jour de la densité $\rho_0(\phi)$ (Eq. (4.18c)) qui prend comme unique argument le champ de phase `phi` (ϕ) :

```
KOKKOS_INLINE_FUNCTION
real_t compute_density(real_t phi) const
{
    return phi*cacParams.rhoL + (1.0 - phi)*cacParams.rhoH;
} // compute_density
```

Le champ ϕ est calculé par le noyau `CollideAndStream_PhaseField_Functor.h`. Les densités ρ_L et ρ_H sont respectivement notées `rhoL` et `rhoH`. Pour la mise à jour des vitesses \mathbf{u} , la boucle sur les populations `ipop` contient les mêmes instructions que celles décrites dans le chapitre 2. Après la boucle, les corrections sont apportées pour tenir compte du terme $(\delta t/2) \mathbf{F}$ dans l'Eq. (4.23a) :

		Note Technique DES	Page 53/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		


```


for (int ipop=0; ipop<npop; ++ipop) {
    vx += dx/dt*E[ipop][IX]*f[ipop];
    vy += dx/dt*E[ipop][IY]*f[ipop];
}
vx = ( vx/(cs*cs) + coeff * fxPhi ) / rho;
vy = ( vy/(cs*cs) + coeff * fyPhi ) / rho;

```

où `coeff` à préalablement été défini à $dt/2$ et où `fxPhi` et `fyPhi` sont préalablement calculées par une autre fonction. Un exemple de calcul de `fxPhi` et `fyPhi` est présenté dans [5]. Les composantes correspondent aux forces de tension superficielle.

Enfin la pression est calculée par les deux fonctions `compute_pressure_force` suivie de `compute_pressure`. La première fonction calcule le terme $\nabla \rho_0(\phi) c_s^2$ en appliquant la méthode des dérivées directionnelles (Eqs. (4.17a) et (4.17b)). Les composantes du gradient sont notées `fx` et `fy`. La seconde fonction calcule successivement le produit scalaire entre le gradient et la vitesse (second terme du membre de droite de l'Eq. (4.23b)) puis ajoute la somme (premier terme du membre de droite de l'Eq. (4.23b)).

		Note Technique DES	Page 54/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

		Note Technique DES	Page 55/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Chapitre 5

Parallélisme à mémoire distribuée et super-calculateurs

Ce chapitre aborde le parallélisme à mémoire distribuée dans LBM_saclay et les tests réalisés sur les super-calculateurs GPUs. Dans la section 5.1 on décrit la « décomposition de domaines » dans LBM_saclay. Cette section décrit les indications à mettre dans le jeu de données (section 5.1.1) puis décrit les communications MPI qui s'opèrent sur les cellules fantômes de chaque sous-domaines. Ensuite, la section 5.2 décrit les tests réalisés sur trois super-calculateurs GPUs. Il rappelle les modules à charger et les scripts `batch` pour exécuter les calculs.

5.1 Décomposition de domaines

Dans LBM_saclay, le parallélisme à mémoire partagée est géré par MPI (Message Passing Interface) avec la bibliothèque `OpenMPI`. Un domaine parallélépipédique qui contient $N_x \times N_y \times N_z$ nœuds est décomposé en plusieurs sous-domaines

$$N_x = m_x \times n_x, \quad N_y = m_y \times n_y, \quad N_z = m_z \times n_z \quad (5.1)$$

où les nombres de sous-domaines dans chaque direction sont m_x , m_y et m_z . Chaque sous-domaine contient respectivement n_x , n_y et n_z nœuds.

5.1.1 Informations dans le jeu de données et IO en HDF5

Informations dans le jeu de données (fichier `.ini`)

Dans le jeu de données (fichier `.ini`), les seuls paramètres à renseigner sont les nombres de sous-domaines m_x , m_y , m_z (`mx=`, `my=`, `mz=` dans la section `[mpi]`) et le nombre de nœuds n_x , n_y et n_z (`nx=`, `ny=`, `nz=` dans la section `[mesh]`). Par exemple pour le cas test du film boiling [5], le maillage est composé de $n_x = 2048$ et $n_y = 768$. Pour une exécution sur 8 GPUs, on peut indiquer dans le fichier `.ini` les paramètres suivants `nx=256`, `ny=768` dans la section `[mesh]` et `mx=8`, `my=1` dans la section `[mpi]`.

Pour un nœud à 2 GPUs, il faut modifier les paramètres `mx/my/mz` du fichier `.ini` pour changer le nombre de tâches MPI. Chaque tâche se retrouve avec une copie du maillage spécifié dans la section `[mesh]` du fichier de paramètres, et le maillage global a donc les dimensions `mx*nx x my*ny x mz*nz`. Par conséquent `nx=256` sera modifié en `nx=1024` dans la section `[mesh]` et `mx=8` sera modifié en `mx=2` dans la section `[mpi]`.

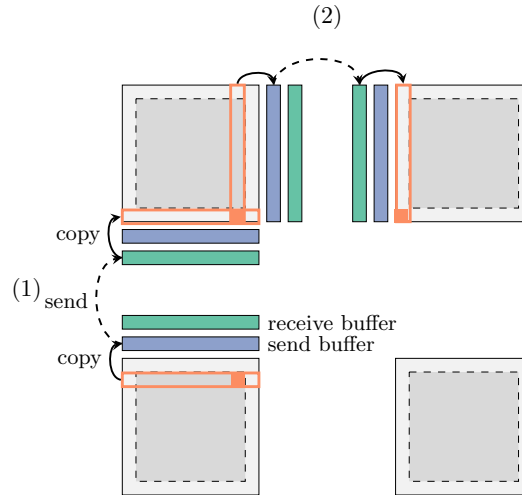


FIGURE 5.1 – Exemple d’une décomposition de domaines MPI 2×2 en 2D ($m_x = m_y = 2$) et des communications aux limites de chaque sous-domaine. Les zones tampon de chaque sous-domaine sont colorées en bleu (buffer d’envoi) et en vert (buffer de réception). Sur cet exemple, on montre comment les données du coin peuvent être transmises en deux étapes au processus en haut à droite. (1) Sous-domaine en bas à gauche : envoi des données de sa limite supérieure au buffer de réception du sous-domaine supérieur (en haut à gauche). (2) Sous-domaine en haut à gauche : communication des données de sa limite à droite au sous-domaine voisin de droite (en haut à droite). Finalement : avec cette façon de procéder, le coin en bas à droite reçoit bien les données du coin du sous-domaine en bas à gauche.

Entrées-Sorties : IO HDF5 parallèles


Le format HDF5 (Hierarchical Data Format version 5) est un format de fichiers qui permet de sauvegarder et de structurer des fichiers contenant de très grandes quantités de données. Un fichier HDF est un conteneur de fichiers. Ce format est très utile pour sauvegarder des résultats d’un calcul issu d’un super-calculateur. Il est implanté dans LBM_saclay pour les configurations suivantes :

1. 2D/3D, serial/MPI, CPU/GPU.
2. prise en compte du layout mémoire différent entre CPU et GPU (si on ne le fait pas les tableaux sont transposés dans les fichiers d’output, parce que HDF5 utilise le layout right en interne pour les tableaux multi-dimensionnels) - HDF5 parallel => écriture dans un seul fichier
3. gestion des lectures HDF5 : nécessaire sur un super-calculateur comme JEAN-ZAY pour faire des reprises (les jobs sont à durée limitée, il faut pouvoir faire une condition initiale à partir d’un run précédent). La gestion de reprises peut se faire
 - (a) en modifiant le nombre de processus MPI
 - (b) en modifiant la résolution : $nx, ny, nz \implies 2*nx, 2*ny, 2*nz$

5.1.2 Communications MPI et cellules fantômes

Chaque processus MPI possède une paire de deux zones « tampon » (des buffers, une “send” et une autre “receive”) pour chaque face de chaque sous-domaine (incluant les coins). Pour chacune d’elles, trois instructions sont appelées : 1. copy_boundaries, 2. transfert_boundaries_3d, et 3. copy_boundaries_back (voir Fig. 5.1).

Les communications se font comme suit :

		Note Technique DES	Page 57/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

1. Tous les sous-domaines copient les données de leur deux faces perpendiculaires à la direction $\pm x$, puis $\pm y$ et $\pm z$ dans leur buffer « send ».

(a) Dans LBMRun.h, la fonction

```
template<int dim, int npop>
void LBMRun<dim, npop>::make_boundaries_mpi(FArray3d fdata)
{
...
}
```

contient un appel

```
copy_boundaries(fdata, XDIR);
```

pour la direction XDIR et cet appel est répété pour les directions YDIR et ZDIR.

(b) La fonction copy_boundaries déclarée par :

```
template<int dim, int npop>
void LBMRun<dim, npop>::copy_boundaries(FArray3d fdata, Direction dir)
{
...
}
```

qui contient pour chaque direction l'appel suivant :

```
CopyFArray_To_BorderBuf<XMIN, THREE_D>::apply(borderBufSend_xmin_3d, fdata, gw,
nbIter);
```

L'exemple extrait porte sur XMIN. Plusieurs autres appels à cette même fonction sont répétés pour YMIN, YMAX, puis pour ZMIN et ZMAX.

2. Les données sont communiquées avec MPI dans les « buffer » de réception des sous-domaines voisins ($+x$ envoie les données au voisin de droite qui reçoit les données dans le buffer de réception $-x$, tandis que $-x$ envoie le buffer au voisin de gauche dans le buffer de réception $+x$).

(a) Dans la fonction make_boundaries_mpi, l'instruction est la suivante :

```
transfert_boundaries_3d(XDIR);
```


(b) et sa fonction est déclarée par

```
template<int dim, int npop>
void LBMRun<dim, npop>::transfert_boundaries_3d(Direction dir)
{
...
}
```

Dans cette fonction on retrouve les commandes suivantes :

```
params.communicator->sendrecv(borderBufSend_xmin_3d.data(),
borderBufSend_xmin_3d.size(),
data_type, params.neighborsRank[X_MIN], 111,
borderBufRecv_xmax_3d.data(),
borderBufRecv_xmax_3d.size(),
data_type, params.neighborsRank[X_MAX], 111);
```

L'exemple extrait porte sur XMIN. Plusieurs autres appels à cette même fonction sont répétés pour XMAX, puis pour YMIN, YMAX, et enfin pour ZMIN et ZMAX.

		Note Technique DES	Page 58/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.			

3. Les sous-domaines copient les données reçues dans les buffers dans les cellules fantômes correspondantes.

(a) Dans la fonction `make_boundaries_mpi`, l'instruction est la suivante :

```
copy_boundaries_back(fdata, XMIN);
```

(b) La fonction est déclarée par :

```
template<int dim, int npop>
void LBMRUN<dim, npop>::copy_boundaries_back(FArray3d fdata, BoundaryLocation loc)
{
    ...
}
```

dans laquelle on retrouve l'instruction suivante :

```
CopyBorderBuf_To_FArray<XMIN, THREE_D>::apply(fdata, borderBufRecv_xmin_3d, gw,
nbIter);
```

Les étapes ci-dessus sont répétées pour les faces perpendiculaires aux directions y et z . En procédant pour chaque axe séparément assure que les données des coins sont échangées de manière cohérente par les cellules fantômes (voir Fig. 5.1).

Dans l'exemple ci-dessus, les données communiquées sont les fonctions de distribution (`fdata`) qui sont nécessaires pour l'étape de déplacement. Dans les versions en cours de développement, on communique aussi les variables macroscopiques intermédiaires et leurs gradients. Ces champs macroscopiques (e.g. ϕ) et leurs dérivées (e.g. $\nabla\phi$) sont nécessaires pour les termes sources de certaines LBE.

5.2 Tests sur les super-calculateurs

Dans cette section on donne quelques informations sur les scripts qui ont été mis en œuvre pour les tests sur les super-calculateurs. Les tests préliminaires de LBM_saclay ont été faits en 2018 sur le calculateur de la Maison De La Simulation (MDLS). Celui est composé d'un seul nœud qui contient sept cartes graphiques Nvidia de l'architecture Kepler K80. Chaque carte graphique contient deux GPUs, ce qui fait un total de 14 GPUs. Ces cartes graphiques sont devenues obsolètes avec l'arrivée des V100 et maintenant des A100. On ne décrit pas les commandes de ce calculateur. Dans la suite on décrit les tests qui ont été faits sur ORCUS (section 5.2.1), la partition GPU de Jean-Zay (section 5.2.2) et celle de TOPAZE (section 5.2.3).

5.2.1 Nœuds GPUs d'ORCUS


Les caractéristiques du cluster ORCUS sont décrites dans le document [15]. Le fichier PDF est accessible après connexion sur le cluster. Les cartes graphiques sont des architectures Volta V100 (16Go HBM2).

Installation

Connexion à ORCUS.

La connexion au cluster se fait en `ssh` sur les nœuds de login `orcusloginamd1`, `orcusloginamd2`, `orcusloginint1` et `orcusloginint2`. Par exemple sur `loginint1` :

```
> ssh [loginintra]@orcusloginint1.intra.cea.fr
password : celui d'intra
```

			Note Technique DES	Page 59/116
	<u>Réf</u> : STMF/LMSF/NT/2022-70869			
			<u>Date</u> : 21/11/2022	<u>Indice</u> : A
LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.				

Récupération du code source.

Pour récupérer les sources du code, on peut faire un `scp` d'une machine locale vers ORCUS :

```
ma machine locale> scp LBM_saclay.tar [loginintra]@orcusloginint1.intra.cea.fr:~/.
```

Ou bien on peut les récupérer depuis un `git` directement à partir d'ORCUS. Par exemple, pour récupérer depuis la « fork » wverdier :

```
orcus> git clone --recursive git@gitlab.maisondelasimulation.fr:wverdier/LBM_saclay.git
```

S'il s'agit de la première connexion d'ORCUS sur le `gitlab` de la MDLS, il est nécessaire de générer une nouvelle clé `ssh`, puis de faire le copier-coller du contenu du fichier `id_rsa.pub` et de l'indiquer dans le cadre du `gitlab`. On pourra se référer à la section [1.1.2](#).

Charger les modules.

Charger les modules qui sont utilisés par LBM_saclay :

```
> module load nvidia_hpc_sdk/21.2_byo_compilers
> module load cmake/3.14.5
> module load hdf5/1.10.1
> module load compilers/gcc/8.3.0
```

Exécuter le cmake.

Pour créer le `Makefile`, on lance la commande `cmake` précédée par les informations relatives au compilateur et aux cartes graphiques Nvidia.

```
> export CC=gcc
> export CXX=../external/kokkos/bin/nvcc_wrapper
> cmake -DKokkos_CXX_STANDARD=14 -DKokkos_ENABLE_CUDA=ON -DKokkos_ARCH_VOLTA70=ON
-DUSE_HDF5=ON ..
```

Compiler


```
> make -j 8
```

Remarque

L'installation initiale du module `nvidia_hpc_sdk/21.2` sur ORCUS posait problème. Ce dernier a été contourné en installant le module dans un répertoire local partagé. Pour intégrer l'environnement CUDA local installé dans le répertoire `/tmpcatB/wv259384/cuda-11.2/bin`, il faut taper les commandes suivantes :

```
> export PATH="/tmpcatB/wv259384/cuda-11.2/bin:$PATH"
> export LD_LIBRARY_PATH="/tmpcatB/wv259384/cuda-11.2/lib64:$LD_LIBRARY_PATH"
```

Les deux dernières commandes peuvent être insérées dans le fichier `.bash_profile` du répertoire « home » personnel pour les lancer automatiquement à chaque connexion.

		Note Technique DES	Page 60/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Exécution d'un cas test

L'exécution d'un cas test passe par un script de lancement de « jobs ». On pourra se référer au manuel d'utilisation [15]. On pourra s'inspirer du fichier « job_lbms_cuda.sh », contenu dans le répertoire /home/catA/ac165432, en faire une copie puis modifier les chemins.

```
#!/bin/bash
#SBATCH -n 1 # Nombre de taches
#SBATCH -p gpuq_5218 # Partitions adaptées au job
#SBATCH --qos=normal # QOS
#SBATCH -t 01:00:00 # Walltime
#SBATCH -J myjob # Nom du job
#SBATCH -o myjob_%j.o # Fichier de sortie
#SBATCH -e myjob_%j.e # Fichier d'erreur
#SBATCH --mail-user=user@cea.fr # Adresse email
#SBATCH --mail-type=begin,end,fail # Envoi d'email à l'exécution, fin et échec du job
# Module nécessaires au fonctionnement de LBM_saclay.
module load compilers/gcc/8.3.0 hdf5/1.10.1 nvidia_hpc_sdk/21.2_byo_compilers
# Lance LBM_saclay. Remplacer les deux chemins comme nécessaire.
srun ~/chemin_executable/LBM_saclay/build_cuda/src/LBM_saclay $1
```

Pour exécuter le calcul il faut taper la commande :

```
> sbatch job_lbms_cuda.sh /chemin_du_repertoire/nom_du_cas_test.ini
```

5.2.2 Calculateur JEAN-ZAY

JEAN-ZAY est un super-calculateur de l'IDRIS <http://www.idris.fr/jean-zay/> qui comprend deux partitions principales : la CPU et la GPU. Sur ce calculateur le cas test du film boiling se trouve dans un répertoire partagé /gpfswork/rech/aih/commun/lbm-saclay-tests.


Connexion et chargement des modules

Pour pouvoir se connecter sur le calculateur JEAN-ZAY il faut avoir fait la demande d'ouverture d'un compte informatique. La demande de création d'un compte est à faire sur <https://www-dcc.extra.cea.fr/CCFR>. Le projet est GENDEN et le numéro de référence est R0091010339 (attention ce numéro change chaque année début novembre à chaque allocation). La documentation est disponible sur http://www.idris.fr/static/intro/doc_nouvel_utilisateur.html.

Une fois le compte créé, la commande de connexion pour le login ubt36ea est :

```
> ssh -XC ubt36ea@jean-zay.idris.fr
```

Ensuite il faut charger les modules nécessaires pour pouvoir compiler des différentes versions de LBM_saclay (taper `module avail` pour voir la liste des modules disponibles sur le calculateur) :

			Note Technique DES	Page 61/116
	<u>Réf</u> : STMF/LMSF/NT/2022-70869			
			<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.			

```
> module load cmake/3.18.0
> module load gcc/8.3.1
> module load cuda/10.2
> module load intel-compilers/19.1.3
> module load hdf5/1.12.0
> module load openmpi/4.0.2-cuda
```

Pour éviter d'avoir à charger ces modules à chaque connexion, on pourra les mettre dans le fichier `modules_compil`. D'autres modules sont à charger pour le post-traitement des résultats avec `paraview` et certains d'entre eux peuvent être en conflit avec les précédents. On pourra les mettre dans le fichier `module_postpro` :

```
> module load gcc/9.1.0
> module load intel-mpi/2019.4
> module load paraview
> module load gnuplot/5.2.8
```

Compilation version CPU/OpenMP


Pour créer le Makefile puis créer l'exécutable dans le répertoire `build_openmp` on écrit les commandes :

```
> mkdir build_openmp
> cd build_openmp
> cmake -DKokkos_ENABLE_OPENMP=ON ..
> make -j16
```

Puis, la soumission d'un job passe par l'écriture d'un script, par exemple `Job_CPU` dans lequel on trouve les instructions suivantes :

```
#!/bin/bash
#SBATCH -A aih@cpu Choix de la partition (cpu ou gpu)
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=20
#SBATCH --hint=nomultithread
#SBATCH -t 01:00:00 Temps limite du calcul
#SBATCH -J myjob Nom du job
#SBATCH -o myjob_%j.o Fichier de sortie
#SBATCH -e myjob_%j.e Fichier d'erreur
#SBATCH --mail-user=user@cea.fr Adresse email
#SBATCH --mail-type=begin,end,fail Envoi d'email à l'exécution, fin et échec du job
# Module nécessaires au fonctionnement de LBM_saclay
module load gcc/8.3.0 hdf5/1.12.0-mpi-cuda
# Lance LBM_saclay. Remplacer les deux chemins comme nécessaire.
srun ~/LBM_saclay/build_openmp/src/LBM_saclay $1
exit 0
```

La ligne d'instruction `#SBATCH -A aih@cpu` indique la partition pour l'exécution du code (ici `cpu`). L'autre option est simplement `gpu`.

			Note Technique DES	Page 62/116
	Réf : STMF/LMSF/NT/2022-70869			
			Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.			

D'autres exemples de scripts d'exécution sont répertoriés sur le site de l'IDRIS. Par exemple, les options à modifier pour exécuter le code en version multi-GPUs (MPI) peuvent être trouvés sur la page http://www.idris.fr/jean-zay/gpu/jean-zay-gpu-exec_multi_mpi_batch.html.

La commande pour la soumission du job est :

```
> cd $WORK
> sbatch Job_CPU /chemin_du_cas_test/nom_du_cas_test_lbm.ini
```

Version GPU

Pour créer le Makefile, puis pour créer l'exécutable dans le répertoire build_cuda on écrit les commandes :

```
> mkdir build_cuda
> cd build_cuda
> export CXX=~/.LBM_saclay-master-DVTS/external/kokkos/bin/nvcc_wrapper
> cmake -DKokkos_ENABLE_OPENMP=ON -DKokkos_ENABLE_CUDA=ON -DKokkos_ENABLE_CUDA_LAMBDA=ON
-DKokkos_ARCH_VOLTA70=ON -DUSE_HDF5=ON ..
> make -j16
```

Un exemple de script d'exécution est donné ci-dessous :


```
#!/bin/bash
#SBATCH -A aih@gpu Partition GPU
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --gres=gpu:1
#SBATCH --cpus-per-task=10
#SBATCH --hint=nomultithread
#SBATCH -t 01:00:00 Temps limite du calcul
#SBATCH -J myjob Nom du job
#SBATCH -o myjob_%j.o Fichier de sortie
#SBATCH -e myjob_%j.e Fichier d'erreur
#SBATCH --mail-user=user@cea.fr Adresse email
#SBATCH --mail-type=begin,end,fail Envoi d'email à l'exécution, fin et échec du job
# Module nécessaires au fonctionnement de LBM_saclay
module load gcc/8.3.0 hdf5/1.12.0-mpi-cuda
# Lance LBM_saclay. Remplacer les deux chemins comme nécessaire.
srun ~/LBM_saclay/build_openmp/src/LBM_saclay $1
exit 0
```

La commande de soumission du job est la suivante :

```
> cd $WORK
> sbatch Job_GPU ~/chemin_du_cas_test/nom_du_cas_test_lbm.ini
```

Version Cuda + MPI

Pour créer le Makefile en version Cuda+MPI on tape les commandes suivantes :

			Note Technique DES	Page 63/116
	<u>Réf</u> : STMF/LMSF/NT/2022-70869			
			<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.			

```
> export OMPI_CXX="chemin/vers/LBM_saclay/external/kokkos/bin/nvcc_wrapper"
> export CXX="chemin/vers/LBM_saclay/external/kokkos/bin/nvcc_wrapper"
> cmake -DKokkos_ENABLE_CUDA=ON -DKokkos_ARCH_VOLTA70=ON -DKokkos_ENABLE_HWLOC=ON
-DUSE_MPI=ON -DUSE_MPI_CUDA_AWARE_ENFORCED=ON -DUSE_HDF5=ON ../
```

Ensuite pour exécuter le code sur 16 GPUs V100 de 32Go on écrit dans un script les commandes suivantes (par exemple dans `slurm_lbmsaclay_jeanzay.sh`) :

```
#!/bin/bash
#SBATCH --job-name=job # nom du job
#SBATCH --account=aih@v100
#SBATCH --qos=qos_gpu-t4
#SBATCH -C v100-32g # pour reserver que des GPU avec 32G de memoire
#SBATCH --ntasks=16 # nombre total de tache MPI
#SBATCH --ntasks-per-node=4 # nombre de tache MPI par nœud (= nombre de GPU par nœud)
#SBATCH --gres=gpu:4 # nombre de GPU par nœud
#SBATCH --cpus-per-task=10 # nombre de cœurs CPU par tache (un quart du nœud ici)
#SBATCH --hint=nomultithread # hyperthreading desactive
#SBATCH --time=100:00:00 # temps d execution maximum demande (HH:MM:SS)
#SBATCH --output=out.o # nom du fichier de sortie
#SBATCH --error=out.e # nom du fichier d erreur

# nettoyage des modules charges en interactif et herites par default
module purge
# chargement des modules
module load cuda/10.2
module load gcc/8.3.1
module load openmpi/4.0.2-cuda
module load hdf5/1.12.0-mpi-cuda
module list


# echo des commandes lancees
set -x
# execution du code
srun chemin/vers/LBM_saclay/build-cuda-mpi/src/LBM_saclay chemin/vers/parameters.ini
--kokkos-num-devices=16
```

La commande de soumission du job est la suivante :

```
> cd $WORK
> sbatch slurm_lbmsaclay_jeanzay.sh
```

Post-traitement des résultats avec paraview sur les nœuds de visualisation

Sur JEAN-ZAY des nœuds sont dédiés à la visualisation des résultats en interactif ou en batch. On suivra la procédure décrite sur la page <http://www.idris.fr/jean-zay/pre-post/jean-zay-outils-visu-noeuds.html>. Pour l'utilisation de ces nœuds en interactif il faut réserver des ressources en temps (1 heure par défaut jusqu'à 4 heures) et en mémoire (10 cœurs donc 40 Go de mémoire) et d'utiliser la carte graphique avec une compression des données entre le serveur de visualisation et votre machine locale à l'aide d'une connexion client/serveur VNC.

		Note Technique DES	Page 64/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

La suite de ce paragraphe est un résumé des commandes pour une utilisation en mode interactif. Depuis un des nœuds frontaux de JEAN-ZAY, il faut taper la commande :

```
jean-zay> idrvnc-alloc
```

Cette commande retourne l'URL de connexion et le mot de passe aux deux dernières lignes. Puis il faut lancer le logiciel VNC client sur sa machine locale (par exemple `remmina` sur Ubuntu 20.04) :

```
ma machine locale> remmina
```

Dans la fenêtre qui s'ouvre, il faut entrer en mode VCN l'URL de login qui est indiqué par la commande `idrvnc-alloc` puis faire le copier-coller du mot de passe. Puis une fois que la fenêtre du nœud de visualisation de JEAN-ZAY est ouverte on y tape les commandes :

```
jean-zay-visu1> module load python 3.7.3
jean-zay-visu1> module load paraview/5.8.0-mpi-python3-nek5000
jean-zay-visu1> vglrun paraview
```

La fenêtre `paraview` s'ouvre.

5.2.3 Partition Topaze-A100

Topaze-A100 (<https://www.cea.fr/presse/Pages/actualites-communiqués/ntic/ccrt-topaze.aspx>) est un calculateur du CCRT co-conçu par Atos et le CEA. Il est ouvert aux utilisateurs depuis 2021. Après connexion il faut charger les modules nécessaires pour pouvoir compiler des différentes versions de `LBM_saclay` :


```
> module load cmake/3.20.3
> module load gnu/8.3.0
> module load flavor/gnu/cuda-11.4
> module load cuda/11.4
> module load mpi/openmpi/4.0.5
> module load flavor/hdf5/parallel
> module load hdf5
> module list
```

Version Cuda + MPI

Pour créer le `cmake`, puis pour créer l'exécutable dans le répertoire `build_cuda_mpi` on écrit les commandes :

```
> mkdir build_cuda_mpi
> cd build_cuda_mpi
> export CXX='../external/kokkos/bin/nvcc_wrapper'
> cmake -DKokkos_ENABLE_CUDA=ON -DKokkos_ARCH_AMPERE80=ON -DUSE_MPI=ON
-DUSE_MPI_CUDA_AWARE_ENFORCED=ON -DUSE_HDF5=ON ..
> make
```

Un exemple de script d'exécution est donné ci-dessous :

		Note Technique DES	Page 65/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

```
#!/bin/bash
#MSUB -r film_boiling # nom du job
#MSUB -q a100 #
#MSUB -Q test # ou normal pour calcul de prod
#MSUB -n 8 # nombre total de tache MPI
#MSUB -N 2 # nombre de nœuds
#MSUB -c 32 # nombre de cœurs par tache
#MSUB -o film_boiling_%j.o # nom du fichier de sortie
#MSUB -e film_boiling_%j.e # nom du fichier d erreur (ici commun avec la sortie)
#MSUB -T 600 # temps limite à modifier

# nettoyage des modules charges en interactif et herites par default
module purge
# chargement des modules
module load flavor/gnu/cuda-11.4
module load gnu/8.3.0
module load cuda/11.4
module load mpi/openmpi/4.0.5
module load flavor/hdf5/parallel
module load hdf5
module list
# echo des commandes lancees
set -x
# execution du code
ccc_mprun /ccc/scratch/cont002/den/eb219944/LBM/LBM_saclay/build/src/LBM_saclay
/ccc/scratch/cont002/den/eb219944/LBM/film-boiling/parameters_film_boiling.ini
--kokkos-num-devices=8
```

Pour l'exécution du code (`ccc_mprun`), on pourra modifier le premier argument pour cibler un `LBM_saclay` compilé soi-même, ou le second fichier pour cibler un autre fichier de paramètres. L'argument donné à `--kokkos-num-devices` doit correspondre à l'argument de `--ntasks` pour `sbatch` et doit aussi correspondre au produit `mx x my x mz` spécifié dans le fichier de paramètres.

Pour soumettre le job la commande est la suivante :


```
> ccc_msub nomduscript
```

Sur Topaze-A100 le cas test du « film boiling » s'est exécuté sur 8 GPUs en 290 secondes répartis comme suit :


```
total time : 289.684 secondes
lbm kernels time : 78.542 secondes 27.11%
boundaries time : 176.165 secondes 60.81%
io time : 9.320 secondes 3.22%
Perf : 1463.00 number of Mcell-updates/s
```

5.2.4 Perspectives : tests sur d'autres super-calculateurs

D'autres super-calculateurs multi-GPUs sont disponibles (au TGCC) ou le seront (au CINES). Au TGCC la partition `Irene-V100` de Joliot-Curie contient des cartes graphiques V100 identiques à celles déjà testées sur


		Note Technique DES	Page 66/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Jean-Zay. Au CINES, le calculateur **Adastra** possède des GPUs AMD MI250. La bibliothèque **Kokkos** compile déjà sur ces cartes graphiques et des tests seront réalisés avec **LBM_saclay** lorsque le calculateur sera en production.

		Note Technique DES	Page 67/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Deuxième partie

Développer son modèle dans LBM_saclay

		Note Technique DES	Page 69/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Préambule

Cette seconde partie se présente sous la forme de plusieurs tutoriels découpés en deux chapitres. Ces tutoriels permettent de décrire pas à pas les modifications à apporter dans `LBM_saclay` pour écrire son noyau de calcul et une condition initiale, puis d'en tenir compte dans `LBMRUN.h`. Les tutoriels sont présentés dans un ordre de complexité croissante en terme de modèles physiques. Le chapitre 6 est dédié au développement d'une unique équation à champ de phase bien connue dans la littérature : l'équation de Cahn-Hilliard. Le chapitre 7 décrit un exemple de couplage entre deux équations : celle du champ de phase couplée à celle de la température. L'exemple s'appuie sur un problème physique qui simule la croissance cristalline d'une substance pure sans écoulement.

Plusieurs tutoriels sont en cours d'écriture. Ainsi dans une version ultérieure de ce document, un chapitre sera consacré aux développements à réaliser pour simuler un écoulement diphasique isotherme en tenant compte d'une loi d'état. Enfin un autre chapitre donnera des indications pour simuler le modèle composé de quatre EDPs non-linéaires et couplées (Navier-Stokes, champ de phase et température). L'exemple s'appuie sur le problème du « film boiling » de la référence [5]. Ces tutoriels sont réalisés en s'appuyant sur une version de développement de `LBM_saclay`. Les fichiers sources de chacun d'eux accompagnent cette documentation et sont disponibles sur le gitlab.

Quelques recommandations


Déclaration de toutes les fonctions `void operator()`

Les fonctions incontournables des appels Kokkos sont les fonctions `void operator()`. Dans un noyau de calcul, il peut en exister plusieurs (deux ou trois) qui se différencient lors des appels avec des tags : par exemple un tag pour appeler la fonction `void operator()` qui calcule la collision et le déplacement et un autre tag pour appeler la fonction `void operator()` qui met à jour le calcul des moments. Dans certains cas on peut vouloir définir un nouveau tag pour appeler une fonction `void operator()` qui calcule uniquement la fonction d'équilibre pour l'initialisation du problème. Dans les chapitres qui suivent les développements sont réalisés et décrits en 2D (chapitre 6) et en 3D pour le chapitre 7. Quelle que soit la dimension physique choisie pour réaliser ses propres développements, toutes les fonctions `void operator()` doivent être déclarées en 2D ET en 3D. Par exemple si un noyau de calcul nécessite de définir trois fonctions `void operator()` en 2D, alors il faut déclarer six fonctions `void operator()` dans le noyau de calcul : trois pour le 2D et trois autres pour le 3D. Si l'une d'elles est absente, alors la compilation du code échouera et l'exécutable ne sera pas créé. Il est donc nécessaire de déclarer toutes les fonctions `void operator()` dans les noyaux de calcul, quitte à ne mettre aucune instruction à l'intérieur des fonctions 3D.

TOUTES les fonctions `void operator()` doivent être déclarées en 2D ET en 3D!!!!!!


Utilisation de fonctions dans `real_type.h`

L'utilisation des fonctions telles que `std::max(arg1,arg2)`, `std::min(arg1,arg2)`, `std::sqrt`, `std::abs`, etc ... sont à éviter lors de l'écriture des noyaux de calcul. La compilation se déroule bien en `OpenMP` mais en version `CUDA`, l'exécutable ne sera pas créé. Pour pouvoir compiler et créer l'exécutable sur toutes les architectures, on substituera à ces fonctions celles qui sont définies dans le fichier `real_type.h`.

		Note Technique DES	Page 70/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Modification du modèle programmation pour les problèmes physiques couplés

Les tutoriels qui suivent sont donnés pour la version « master » de LBM_saclay. Cependant, le modèle de programmation a évolué dans les versions actuelles du code. La principale raison est de faciliter les développements de problèmes physiques qui impliquent plusieurs EDPs couplées (trois ou plus) et d'éviter à avoir à gérer la combinatoire des branchements conditionnels entre les équations. En effet, même si cette version permet déjà de simuler des équations couplées (voir le second tutoriel), le nombre de fichiers à modifier et les différents endroits du code à adapter sont importants lorsque le problème est multi-physique. Par exemple, pour un système d'EDPs composé ses équations de Navier-Stokes couplées au champ de phase et à une ou deux EDPs sur les compositions, la complexité vient du fait que, même pour ce même jeu d'EDPs, leurs termes sources peuvent varier, ainsi que leurs conditions initiales, leurs paramètres et de leur interpolation. Pour les développeurs novices, il est plus aisé de regrouper dans un unique répertoire l'ensemble des fichiers où apparaissent clairement les modifications à apporter pour développer son propre modèle. Des versions du code sont en cours de développements dans les thèses de WERNER VERDIER [7, 16] et de TÉO BOUTIN [17].

		Note Technique DES	Page 71/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.			

Chapitre 6

Tutoriel sur l'équation de Cahn-Hilliard

Dans ce chapitre on donne un exemple de développement dans `LBM_saclay` pour simuler un modèle à champ de phase bien connu dans la littérature : l'équation de Cahn-Hilliard. Ainsi on verra comment écrire son noyau de calcul, puis d'en tenir compte dans `LBMRun.h`. Ce nouveau noyau calcule un nouveau champ macroscopique, le potentiel chimique, qu'il sera utile de stocker en mémoire dans le tableau de type `LBMArray`. Pour cela, on montrera les modifications à apporter à `LBM_enums.h` et `FieldManager.h`. Dans un premier temps, la simulation se fera en reprenant une condition initiale déjà programmée de façon vérifier les développements. Ensuite on verra comment écrire une nouvelle condition initiale pour simuler un autre phénomène, la décomposition spinodale, ce que permet de simuler l'équation de Cahn-Hilliard. Dans ce chapitre, le modèle mathématique et l'algorithme LBM sont rappelés dans la section 6.1. Ensuite, la section 6.2 présente la mise en œuvre dans `LBM_saclay`. Enfin la section 6.3 illustre les développements réalisés sur le cas test du vortex et la seconde sur un cas test de décomposition spinodale.

6.1 Équation de Cahn-Hilliard

L'équation de Cahn-Hilliard est une équation de conservation qui ressemble à celle d'advection-diffusion (ADE de la section 4.2) mais pour laquelle le flux n'est pas donné par la loi classique de Fick $\mathbf{j}_C = -\mathcal{D}\nabla C$ où \mathcal{D} est le coefficient de diffusion mais par celui de Cahn-Hilliard qui fait intervenir le potentiel chimique $\mu_\phi : \mathbf{j}_{\mu_\phi} = -M_\phi \nabla \mu_\phi$ où M_ϕ est la mobilité.

6.1.1 Rappel de l'équation continue


Dans la littérature on la retrouve écrite dans les publications avec la notation C (pour la composition) ou ϕ pour l'indicatrice de phase. Ici on utilise la notation ϕ :

$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\mathbf{u}\phi) = \nabla \cdot (M_\phi \nabla \mu_\phi) \quad (6.1a)$$

Dans cette équation ϕ joue les deux rôles : celui de la composition du système et celui de l'indicatrice de phase. Le potentiel chimique est défini par :

$$\mu_\phi = 4H\phi(\phi - 1) \left(\phi - \frac{1}{2} \right) - K \nabla^2 \phi. \quad (6.1b)$$

Le premier terme du membre de droite de l'Eq. (6.1b) est la dérivée du double-puits de potentiel. Ce dernier est choisi par convention dans ce document tel que ses minima sont égaux à 0 et à 1. D'autres définitions existent

		Note Technique DES	Page 72/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

dans la littérature (ex. minima en ± 1 ou $\pm \phi^*$). Le second terme est la dérivée du terme d'énergie de gradient dans la définition de l'énergie libre.

Les paramètres K et H

Les deux paramètres K (le coefficient d'énergie du gradient) et H (hauteur du double-puits de potentiel) sont reliés aux coefficients macroscopiques W et σ qui sont respectivement l'épaisseur de l'interface diffuse et la tension de surface :

$$\sigma = \frac{1}{6} \sqrt{2KH} \quad \text{et} \quad W = \sqrt{\frac{8K}{H}} \quad (6.2)$$

En pratique, les coefficients physiques σ et W sont fixés par l'utilisateur dans le fichier d'entrée et les relations Eq. (6.2) sont inversées pour en déduire les deux coefficients K et H . On en reparle dans la section 6.1.3.

Conditions initiales

Pour les simulations on utilisera deux conditions initiales différentes. La première est déjà développée dans LBM_saclay et a été appliquée pour étudier la déformation dans un vortex. La seconde condition initiale consiste à définir un champ aléatoire ϕ pour simuler la décomposition spinodale. Le cas test du vortex est décrit dans la publication [5, Sec 4.1.1] pour valider l'équation CAC (voir chapitre 4). La condition initiale pour ϕ s'écrit dans un domaine bidimensionnel de dimension $[0, 1]^2$:

$$\phi(\mathbf{x}, 0) = \frac{1}{2} \left[1 + \tanh \left(\frac{(R - d_0)}{\sqrt{2}W_0} \right) \right] \quad (6.3a)$$

où d_c est définie par :

$$d_0 = \sqrt{(x - x_0)^2 + (y - y_0)^2} \quad (6.3b)$$

avec $\mathbf{x}_0 = (x_0 = 0.5, y_0 = 0.3)^T$, $W_0 = 0.01$ et $R = 0.2$.

Les composantes de la vitesse \mathbf{u} sont quant à elles définies par :

$$u_x(\mathbf{x}, 0) = -u_0 \cos[\pi(x - 0.5)] \sin[\pi(y - 0.5)] \quad (6.4a)$$

$$u_y(\mathbf{x}, 0) = u_0 \sin[\pi(x - 0.5)] \cos[\pi(y - 0.5)] \quad (6.4b)$$

où $u_0 = 0.7853975$.

6.1.2 Rappel de l'algorithme LBM pour le Cahn-Hilliard


Pour l'équation de Cahn-Hilliard, l'algorithme LBM s'écrit (voir par exemple [18]) :

$$g_i(\mathbf{x} + \mathbf{c}_i \delta t, t + \delta t) = g_i(\mathbf{x}, t) - \frac{1}{\tau_g} [g_i(\mathbf{x}, t) - g_i^{eq}(\mathbf{x}, t)] \quad (6.5)$$

avec la fonction de distribution à l'équilibre définie par :

$$g_i^{eq, CH} = \mathcal{A}_i(\phi, \mu_\phi) + \phi w_i \frac{\mathbf{c}_i \cdot \mathbf{u}}{c_s^2} \quad (6.6a)$$

$$\mathcal{A}_i(\phi, \mu_\phi) = \begin{cases} \phi - 3\mu_\phi(1 - w_0) & \text{si } i = 0 \\ 3w_i\mu_\phi & \text{si } i \neq 0 \end{cases} \quad (6.6b)$$

		Note Technique DES	Page 73/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

La fonction à l'équilibre est séparée en deux termes distincts, le premier \mathcal{A}_i pour les moments d'ordre zéro et deux et le second terme pour le moment d'ordre un. Ce dernier contient le produit scalaire $(w_i \mathbf{c}_i / c_s^2) \cdot \mathbf{u} \phi$ pour le terme $\nabla \cdot (\mathbf{u} \phi)$.

Le coefficient de mobilité M_ϕ est relié au taux de collision τ_g par la relation :

$$M_\phi = \frac{1}{3} \left(\tau_g - \frac{1}{2} \right) \frac{\delta x^2}{\delta t}, \quad (6.7)$$

et le moment d'ordre zéro est mis à jour par :

$$\phi = \sum_i g_i \quad (6.8)$$

Le Laplacien de ϕ qui apparaît dans la définition du potentiel chimique μ_ϕ est calculé par la méthode des dérivées directionnelles :

$$(\mathbf{e}_i \cdot \nabla)^2 \phi|_{\mathbf{x}} = \frac{1}{\delta x^2} [\phi(\mathbf{x} + \mathbf{e}_i \delta x) - 2\phi(\mathbf{x}) + \phi(\mathbf{x} - \mathbf{e}_i \delta x)] \quad (6.9a)$$

$$\nabla^2 \phi|_{\mathbf{x}} = 3 \sum_{i=0}^{N_{pop}} w_i (\mathbf{e}_i \cdot \nabla)^2 \phi|_{\mathbf{x}} \quad (6.9b)$$

L'Eq. (6.9a) est la formule du calcul d'un laplacien par différences finies mais pour chaque direction du réseau. L'Eq. (6.9b) réalise la somme de chaque direction en les pondérant par w_i .


6.1.3 Quelques remarques

Différences avec l'Eq. CAC de la section 4.3. L'équation de Cahn-Hilliard (Eq. (6.1a)) ressemble à celle de Allen-Cahn de la section 4.3 (Eq. (4.11)) à la différence du flux qui s'exprime comme $\mathbf{j}_{CH} = -M_\phi \nabla \mu_\phi$ pour la première et $\mathbf{j}_{AC} = -M_\phi (\nabla \phi - 4\phi(1-\phi)\mathbf{n}/\xi)$ pour la seconde. Cette différence du flux se traduit par une modification de la fonction d'équilibre : Eqs. (6.6a)-(6.6b) pour le Cahn-Hilliard et Eq. (4.14) pour le Allen-Cahn. Néanmoins le tableau `g0` déjà déclaré pour le modèle de Allen-Cahn peut être repris, de même que le coefficient de mobilité M_ϕ et son taux de collision associé τ_g .

Les fonctions d'équilibres $g_i^{eq, CAC}$ et $g_i^{eq, CH}$. Dans la construction des deux fonctions d'équilibre $g_i^{eq, CH}$ (Eq. 6.6a)-(6.6b) et $g_i^{eq, CAC}$ (Eq. 4.14), la principale différence vient de ce que le moment d'ordre zéro et le moment d'ordre deux sont différents pour l'équation CH : il vaut ϕ pour le premier et μ_ϕ pour le second. Pour l'équation CAC ses deux moments d'ordre zéro et deux sont identiques : ils valent tous les deux ϕ . C'est la raison pour laquelle dans l'Eq. (6.6a), le premier terme du membre de droite $\mathcal{A}_i(\phi, \mu_\phi)$ est séparé en deux parties dans l'Eq. (6.6b) : la première ligne pour le moment d'ordre 0 et la seconde ligne pour le moment d'ordre 2 et qui ne dépend que de μ_ϕ .

Paramètres H et K . Le calcul du potentiel chimique est réalisé par la relation Eq. (6.1b). Celle-ci fait apparaître deux nouveaux paramètres H et K qui sont reliés à deux paramètres physiques σ et W , respectivement la tension de surface et l'épaisseur de l'interface diffuse. Les paramètres physiques σ et W seront lus dans le fichier de données `.ini`, puis on calculera les paramètres H et K en utilisant les relations Eq. (6.2) :

$$K = \frac{3}{2} W \sigma \quad \text{et} \quad H = 12 \frac{\sigma}{W} \quad (6.10)$$

			Note Technique DES	Page 74/116
	Réf : STMF/LMSF/NT/2022-70869			
			Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.			

Laplacien $\Delta\phi$. Dans sa définition, le potentiel chimique μ_ϕ fait apparaître le laplacien de ϕ qu'il faudra calculer à l'aide des relations Eqs. (6.9a) et (6.9b). L'Eq. (6.9a) n'est pas locale, elle implique les premiers voisins $\phi(\mathbf{x} \pm \mathbf{e}_i \delta x)$. On s'affranchira des problèmes de conditions aux limites qui en découlent en considérant pour les cas tests uniquement des conditions aux limites périodiques. Les relations (6.9a) et (6.9b) ne sont valables que dans le « bulk » et doivent être modifiées aux limites du domaines.

6.2 Mise en œuvre dans LBM_saclay

Le plus simple est de repartir du noyau de calcul `CollideAndStream_PhaseField_Functor.h` en créant une copie et en la renommant `CollideAndStream_CahnHilliard_Functor.h`. Ce fichier contiendra la nouvelle classe `CollideAndStream_Cahn_Hilliard_Functor`. Après y avoir apporté des modifications relatives au Cahn-Hilliard (section 6.2.1), ce nouveau noyau de calcul sera pris en compte dans `LBMRun.h` (section 6.2.2). Puis on décrit la modification à apporter à `LBM_enums.h` et `FieldManager.h` (section 6.2.3) et on décrira les conditions initiales (section 6.2.4).

Pour pouvoir refaire ce tutoriel il faut avoir copié au préalable les deux fichiers `FiniteDiffs.h` et `UtilsCompute.h` qui sont contenus dans le répertoire Tutos dans le répertoire `src/kernels`. On parle de ces deux fichiers dans la section 6.2.1.2.

6.2.1 Nouveau noyau `CollideAndStream_CahnHilliard_Functor.h`

Dans ce nouveau noyau de calcul, on définit une nouvelle classe `CollideAndStream_Cahn_Hilliard_Functor` qui doit contenir les mêmes fonctions que `CollideAndStream_phase_field_Functor`, c'est-à-dire 1) le constructeur de la classe `CollideAndStream_Cahn_Hilliard_Functor`, 2) une fonction `apply` et 3) plusieurs fonctions précédées par la commande `KOKKOS_INLINE_FUNCTION` parmi lesquelles doit figurer la fonction `void operator()`.

Dans les deux premières lignes d'entête du fichier, on remplace tout d'abord la chaîne de caractères `PHASE_FIELD` par `CAHN_HILLIARD` :

```
#ifndef COLLIDE_AND_STREAM_CAHN_HILLIARD_FUNCTOR_H_
#define COLLIDE_AND_STREAM_CAHN_HILLIARD_FUNCTOR_H_
```


On remplacera également toutes les chaînes de caractères `_phase_field_` par `_Cahn_Hilliard_` dans les noms des fonctions définies dans la classe, à l'exception de celles qui apparaissent dans les deux lignes dont les instructions contiennent la commande `configMap.getFloat("phase_field", ...)`. Ensuite, dans cette nouvelle classe il faut définir une nouvelle structure de paramètres relatifs à l'équation de Cahn-Hilliard (section 6.2.1.1), puis tenir compte du calcul du potentiel chimique (section 6.2.1.2) qui sera lui-même calculé dans une fonction dédiée (section 6.2.1.3). Enfin il faut modifier la fonction d'équilibre (section 6.2.1.4).

6.2.1.1 Nouvelle structure de paramètres `CH_params`

Dans cette structure de paramètres, on définit la tension superficielle σ (`sigma`) et l'épaisseur de l'interface W (`Width`) après la commande `#include "LBM_Base_Functor.h"` :

```
#include "LBM_Base_Functor.h"
struct CH_Params {real_t Width, sigma; };
```

Les valeurs de ces deux paramètres doivent être indiquées dans le fichier d'entrée `.ini`. Elles seront lues dans la fonction `update_phase_field` dans le fichier `LBMRun.h` (voir section 6.2.2). Cette structure de paramètres doit passer en arguments du constructeur de la classe `CollideAndStream_Cahn_Hilliard_Functor`, puis elle doit être initialisée :

		Note Technique DES	Page 75/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.			

Dans `CollideAndStream_Cahn_Hilliard_Functor` on passe l'argument, puis on initialise :

```
CollideAndStream_Cahn_Hilliard_Functor(LBMPParams params, ..., real_t xi, CH_Params
chParams) :
..., chParams(chParams)
```

Dans `apply` on passe l'argument aussi l'argument :

```
static void apply(ConfigMap &configMap, ..., CH_Params chParams);
```

Dans `CollideAndStream_Cahn_Hilliard_Functor functor()` :

```
CollideAndStream_Cahn_Hilliard_Functor functor(params, fm, ..., param_xi, chParams);
```

Dans les membres de la classe (à la toute fin du fichier) on ajoute aussi la ligne :

```
real_t mobility;
real_t xi;
const CH_Params chParams ;
; // CollideAndStream_Cahn_Hilliard_Functor
#endif // COLLIDE_AND_STREAM_CAHN_HILLIARD_FUNCTOR_H
```

6.2.1.2 Pré-requis pour le calcul du potentiel chimique


Parmi les fonctions précédées de la « décoration » `KOKKOS_INLINE_FUNCTION`, les fonctions 2D et 3D relatives au calcul de \mathbf{n} (`compute_normal_vector`) peuvent être supprimées. On les remplacera par le calcul du potentiel chimique μ_ϕ , par exemple en définissant une nouvelle fonction `compute_chemical_potential`. Le calcul du potentiel chimique nécessite le calcul du laplacien de ϕ ($\Delta\phi$), dont le calcul est déjà pris en compte dans la fonction `compute_laplacian(...){...}` dans le fichier `FiniteDiffs.h`. Ce dernier utilise également un fichier d'utilitaires pour les calculs : `UtilsCompute.h` qui doit être présent dans le répertoire `kernels`. Pour utiliser la fonction, en entête de `CollideAndStream_CahnHilliard_Functor.h`, on inclut les lignes suivantes entre les instructions `#include ...` et le struct `CH_Params` :

```
#include "LBM_Base_Functor.h"
#include "kernels/FiniteDiffs.h"
using kernels::finite_diffs::compute_laplacian;
struct CH_params {real_t Width, sigma};
```

La fonction `compute_laplacian` sera appelée dans la fonction `compute_chemical_potential` qui sera décrite dans la section 6.2.1.3. L'appel de la fonction `compute_chemical_potential` est réalisé dans la fonction `void operator()` qui réalise la mise à jour avec l'option `TagUpdate`.

Dans `void operator()`, qui est déclaré par :

```
template<int dim_ = dim>
KOKKOS_INLINE_FUNCTION
void operator()(const TagUpdate&, const typename std::enable_if<dim_==2, int>::type&
index) const {...}
```

		Note Technique DES	Page 76/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

on remplace les lignes suivantes

```
// phase field gradient
real_t dPhidx=0, dPhidy=0;
// compute normal vector
compute_normal_vector(dPhidx,dPhidy, i,j, isize,jsize, dx, e2);
```

par :

```
lbm_data(i, j, fm[IMU]) = compute_chemical_potential(i, j, isize, jsize, dx, e2);
```

Le potentiel chimique est stocké dans le tableau des champs macroscopiques `lbm_data` avec un nouvel indice `IMU` et qui doit être pris en compte dans `LBM_enums.h` (voir section 6.2.3).

6.2.1.3 Nouvelle fonction `compute_chemical_potential`

La nouvelle fonction `compute_chemical_potential` doit être déclarée par :

```
KOKKOS_INLINE_FUNCTION
real_t compute_chemical_potential( int i, int j, int isize, int jsize,
                                   real_t dx, real_t e2) const {...}
```

où ... dans les accolades `{}` représente les instructions qui sont décrites ci-dessous.

Dans cette fonction, les coefficients H et K de l'Eq. (6.1b) sont définis à l'aide des coefficients macroscopiques lus dans le fichier d'entrée, i.e. la tension superficielle σ (`sigma`) et l'épaisseur de l'interface diffuse W (`Width`) à l'aide des Eqs. (6.10) et de la structure de données `chParams`.


Dans `compute_chemical_potential`, on définit les paramètres `param_K` et `param_H`, puis on récupère la valeur locale de `phi` à partir du tableau `lbm_data` :

```
real_t param_K = (3.0*0.5) * chParams.sigma * chParams.Width ;
real_t param_H = 12.0 * (chParams.sigma / chParams.Width) ;
const real_t phi = lbm_data(i, j, fm[IPHI]);
```

Enfin, le potentiel chimique est la différence de deux termes `mu1` et `mu2` qui sont respectivement calculés par :

```
const real_t mu1 = 4.0 * param_H * phi * (phi - 0.5) * (phi - 1.0) ;
const real_t mu2 = param_K * compute_laplacian<npop>(i,j,fd_bounds,dx,
                                                    E, w, e2, lbm_data, fm[IPHI]);
return mu1-mu2 ;
```

En regardant la fonction `compute_laplacian<Q>(...)` qui se trouve dans le fichier `FiniteDiffs.h`, on peut voir qu'elle est séparée en deux sous-fonctions qui calculent le laplacien dans le bulk (Eqs. (6.9a)-(6.9b)) et sur les bords. Cette fonction utilise parmi ses arguments `fd_bounds` (voir section 6.2.1.3) qui doit être préalablement déclaré à la fin des déclarations des membres de la classe (à la fin du fichier) et initialisé dans le constructeur :

		Note Technique DES	Page 77/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.			

```
const CH_Params chParams ;
Kokkos::Array<int, 6> fd_bounds;
```

Après `CollideAndStream_Cahn_Hilliard_Functor(...)`, on initialise `fd_bounds` :

```
... chParams(chParams),
fd_bounds(kernels::finite_diffs::centered_stencil_bounds(
    params.isize, params.jsize, params.ksize, params.ghostWidth, bc_bitflag))
```

Enfin, la fonction `compute_laplacian` utilise en argument un entier (`bc_bitflag`) qui doit être déclaré dans la classe et passer en argument des fonctions de cette classe. L'entier `bc_bitflag` est l'indice qui indique le type de condition aux limites (périodique, Dirichlet, Neumann) appliqué à chaque face du domaine de calcul. Sur le même schéma que pour la structure de paramètres `chParams`, on le rajoutera successivement aux arguments du constructeur `CollideAndStream_Cahn_Hilliard_Functor(...)`, et des fonctions `static void apply(...)`, `CollideAndStream_NS_CAC_Functor(...)` et sans oublier de mettre à jour les membres de la classe à la fin du fichier. Par exemple dans le constructeur de la classe, on ajoute l'argument :

```
...
CH_Params chParams,
int bc_bitflag) :
```

6.2.1.4 Modification de l'équilibre


Dans `void operator()` qui réalise la mise à jour (`TagUpdate`), il faut modifier les instructions qui calculent la fonction de distribution à l'équilibre $g_i^{eq, CAC}$ et les remplacer par $g_i^{eq, CH}$. Ainsi les lignes

```
for (int ipop=0; ipop<npop; ++ipop) {
    f[ipop] = f0(i,j, ipop);
    real_t scal1 = dx/dt*(E[ipop][IX]*vx + E[ipop][IY]*vy);
    real_t scal2 = dx/dt*(E[ipop][IX]*dPhidx + E[ipop][IY]*dPhidy);
    feq[ipop] = w[ipop] * phi * (1.0 + w1*scal1) + w[ipop] * w1 * work * scal2;
}
```

doivent être remplacées par le calcul de la fonction d'équilibre relative au Cahn-Hilliard. Dans l'Eq. (6.6b), le calcul du coefficient \mathcal{A}_i se calcule différemment pour $i = 0$ (`ipop=0`) et pour $i \neq 0$ (boucle `for`). Pour `ipop=0`, l'équilibre `feq` ne fait pas apparaître le terme advectif puisque $\mathbf{c}_0 = (0, 0)^T$:

```
const real_t PotChim = lbm_data(i,j,fm[IMU]);
int ipop = 0;
feq[ipop] = phi - 3.0*PotChim*(1-w[ipop]);
f [ipop] = f0(i,j,ipop);
```

Pour $i \neq 0$ (boucle `for`), on tient compte de $\mathcal{A}_{i \neq 0}$ et du produit scalaire `scal1`. Le coefficient c_s^2 du dénominateur est noté `w1` qui est déjà défini par l'instruction `const real_t w1 = 3.0 / (c*c);` :

		Note Technique DES	Page 78/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

```
for (ipop=1; ipop<npop; ++ipop) {
    f[ipop] = f0(i,j, ipop);
    real_t scal1 = dx/dt*(E[ipop][IX]*vx + E[ipop][IY]*vy);
    feq[ipop] = 3.0*w[ipop]*PotChim + phi*w[ipop]*w1*scal1;
}
```

Les mêmes modifications de la fonction de distribution à l'équilibre doivent être apportées à la fonction `void operator()` qui a le tag `TagComputeFeq`.

6.2.2 Modifications à apporter dans `LBMRun.h`

L'écriture du noyau de calcul `CollideAndStream_CahnHilliard_Functor.h` est terminée, il faut maintenant en tenir compte dans le fichier `LBMRun.h`. en l'incluant (section 6.2.2.1) puis en faisant les appels (section 6.2.2.2).

6.2.2.1 Inclure le nouveau noyau de calcul

Dans le fichier pilote `LBMRun.h` il faut d'abord inclure le nouveau noyau de calcul :

```
// phase field
#include "kernels/CollideAndStream_PhaseField_Functor.h"
#include "kernels/CollideAndStream_CahnHilliard_Functor.h"
```

6.2.2.2 Faire l'appel de la nouvelle fonction


Ensuite il faut tenir compte de cette nouvelle classe « Cahn-Hilliard » dans la fonction `update_phase_field`. Pour cela, dans cette fonction, on ajoutera une condition sur l'option du modèle `phase_field_model` qui peut prendre les deux valeurs « Allen-Cahn » ou « Cahn-Hilliard » selon le modèle choisi dans le fichier d'entrée. On suit l'exemple de ce qui est déjà fait pour la fonction `update_navier_stokes` dans laquelle les deux options « NS_STD » ou « NS_CAC » peuvent être choisies.

Dans `update_phase_field`, on lit l'option du modèle :

```
if (params.phase_field_init == SHRF_VORTEX) {
    ...
}
// phase_field_model can be : Allen-Cahn, Cahn-Hilliard
const std::string phase_field_model = configMap.getString("lbm", "phase_field_model",
"Allen-Cahn");
```

Ensuite il faut ajouter les conditions `if` des instructions à réaliser selon l'option choisie :

```
if (phase_field_model == "Allen-Cahn") {
    auto tag = CollideAndStream_phase_field_Functor<dim,npop>::TAG_UPDATE;
    CollideAndStream_phase_field_Functor ...
}
if (phase_field_model == "Cahn-Hilliard") {
    ...
}
```

			Note Technique DES	Page 79/116
	<u>Réf</u> : STMF/LMSF/NT/2022-70869			
			<u>Date</u> : 21/11/2022	<u>Indice</u> : A
LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.				

À l'intérieur du if pour l'option Cahn-Hilliard, les ... sont les instructions qui lisent les valeurs des paramètres :

```
if (phase_field_model == "Cahn-Hilliard") {
    real_t sigma = configMap.getFloat("phase_field","sigma",0.01);
    real_t Width = configMap.getFloat("phase_field","Width",0.01);
    CH_Params chParams = {.Width=Width, .sigma=sigma};
```

puis qui appellent la fonction apply de la classe CollideAndStream_Cahn_Hilliard_Functor pour réaliser les calculs :

```
int bc_bitflag = 0 ;
auto tag = CollideAndStream_Cahn_Hilliard_Functor<dim,npop>::TAG_UPDATE;
CollideAndStream_Cahn_Hilliard_Functor<dim,npop>::apply(configMap, params,
    fm, data, f_in, f_out, tag, chParams, bc_bitflag);
}
```

L'entier `bc_bitflag` est l'indice qui indique le type de condition aux limites appliqué à chaque face du domaine de calcul. Ici il est imposé à 0 pour indiquer des conditions aux limites périodiques pour toutes les faces. Dans une des versions de développement du code, cet indice est généralisé dans un tableau qui est pris en compte dans `LBMParams.cpp` et `LBMParams.h`. Afin de garder une longueur raisonnable de ce chapitre on décrira cette généralisation dans une version ultérieure de ce document.

6.2.3 Modifications dans `LBM_enums.h` et `FieldManager.h`

La prise en compte du nouvel indice IMU pour stocker le potentiel chimique dans le tableau `lbm_data` nécessite d'en tenir compte dans les fichiers `LBM_enums.h` et `FieldManager.h` qui sont tous les deux contenus dans `src`.

Dans `LBM_enums.h` on ajoute l'indice du potentiel chimique dans l'énumération et on incrémente le nombre de composantes :


```
enum ComponentIndex {..., IMU=6, COMPONENT_SIZE=7};
```

Dans `FieldManager.h` on tient compte du nouveau champ, le potentiel chimique, en incrémentant le nombre de champs :

```
if (params.lbm_phase_field_enabled) {
    var_enabled[IPHI] = 1;
    var_enabled[IMU] = 1;
}
```

On ajoute ensuite les noms associés à ce nouvel indice IMU. Tout d'abord dans la fonction `static int2str_t get_id2names_all() {...}`, on ajoute la ligne :

```
map[IP] = "pressure";
map[IMU] = "chempot";
```

		Note Technique DES	Page 80/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.			

Puis dans `static str2int_t get_names2id_all() {...}` on ajoute la ligne :

```
map["pressure"] = IP;
map["chempot"] = IMU;
```

6.2.4 Conditions initiales

Cette section est dédiée aux conditions initiales de l'équation de Cahn-Hilliard. Dans la section 6.2.4.1 on rappelle la condition initiale du cas test du vortex. Dans la section 6.2.4.2, on décrit comment programmer une condition initiale qui représente un mélange uniforme d'un système binaire.

6.2.4.1 La condition initiale `InitPhi_Shrf_Vortex.h`

Les conditions initiales pour $\phi(x, 0)$ (Eqs. (6.3a)-(6.3b)) et pour $u(x, 0)$ (Eqs. (6.4a)-(6.4b)) sont déjà programmées dans le fichier `InitPhi_Shrf_Vortex.h` qui est situé dans le répertoire `src/init_cond/phase_field`. Ce fichier est intégré dans le fichier `InitCond.h` qui regroupe les différentes conditions initiales par l'instruction `#include "init_cond/phase_field/InitPhi_Shrf_Vortex.h"`. Ce fichier `InitCond.h` est lui-même inclus dans `LBMRun.h` par la commande `#include "InitCond.h"`.

Le fichier `InitPhi_Shrf_Vortex.h` contient la classe `Shrf_Vortex_Functor` qui est munie d'une fonction `apply`. Celle-ci est appelée dans la fonction `init_condition_phase_field` dans le fichier `LBMRun.h`. Une fois que $\phi(x, 0)$ et $u(x, 0)$ ont été initialisés et stockés dans le tableau `ldata`, celui-ci est utilisé pour calculer la fonction d'équilibre en passant en argument de la fonction `apply` de la classe `CollideAndStream_phase_field_Functor` avec l'option `TAG_COMPUTE_FEQ` (voir le second encadré de la section 3.1.3).

6.2.4.2 Nouvelle condition initiale `InitPhi_Unif_Mixture.h`

Pour simuler un autre cas test avec l'équation de Cahn-Hilliard, on initialise un mélange à partir d'une répartition aléatoire et uniforme de ϕ à l'intérieur du domaine de calcul. À chaque nœud du maillage on attribue au champ ϕ une valeur aléatoire issue d'une loi uniforme. Pour cela, on crée un nouveau fichier `InitPhi_Unif_Mixture.h` dans le répertoire `src/init_cond/phase_field` en créant une copie du fichier `InitPhi_Zalesak.h`. Dans ce nouveau fichier on apporte les modifications en suivant l'exemple qui est présenté sur la page https://github.com/kokkos/kokkos/blob/master/example/tutorial/Algorithms/01_random_numbers/random_numbers.cpp.


Dans `InitPhi_Unif_Mixture.h`, on inclut la bibliothèque de fonctions aléatoires `Kokkos_Random.hpp` :

```
#endif // __CUDA_ARCH__
#include <Kokkos_Random.hpp>
#include "kokkos_shared.h"
```

Ensuite on remplace toutes les chaînes de caractères `Zalesak` par `Unif_Mixture` (le faire aussi pour toutes les chaînes en caractères majuscules). Dans la nouvelle classe `class Unif_Mixture_Functor` on ajoute une nouvelle variable `rand_pool` dans le constructeur qu'on initialise ensuite :

```
Unif_Mixture_Functor(Unif_Mixture_Params gParams, ...,
                    LBMArray data, Kokkos::Random_XorShift64_Pool<> rand_pool) :
... data(data), rand_pool(rand_pool)
```

et qu'on déclare dans les membres de la classe en fin de fichier :

		Note Technique DES	Page 81/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

```
LBMArray data;
Kokkos::Random_XorShift64_Pool<> rand_pool;
}; // Unif_Mixture_Functor
```

Dans `apply`, on initialise le générateur du nombre aléatoire :

```
// create and launch our functor
std::size_t nbIter = dim==2 ? params.isize*params.jsize :
                             params.isize*params.jsize*params.ksize;
// initialiser le générateur du nombre aléatoire
Kokkos::Random_XorShift64_Pool<> rand_pool(5374857);
```

Dans `void operator()`, on peut supprimer ou commenter toutes les instructions qui initialisent la vitesse et celles qui mettent à zéro le champ ϕ . Ensuite on ajoute les lignes suivantes :

```
// Get a random number state from the pool for the active thread
Kokkos::Random_XorShift64_Pool<>::generator_type rand_gen = rand_pool.get_state();
```

Puis on initialise le champ par :

```
data(i ,j , fm[IPHI]) = rand_gen.drand();
rand_pool.free_state(rand_gen);
```

On tient compte de ce nouveau fichier en faisant le `#include` dans le fichier `InitCond.h`. Puis dans le fichier `LBMRun.h`, on tient compte de cette condition initiale dans la fonction `init_condition_phase_field` en rajoutant une nouvelle possibilité avec la commande.

Dans `init_condition_phase_field`, on ajoute les lignes relatives à la condition `else if` :

```
else if (params.phase_field_init == UNIF_MIXTURE) {
    Unif_Mixture_Functor<dim,npop>::apply(configMap, params, fm, ldata);
}
```


Cette nouvelle option `UNIF_MIXTURE` doit aussi être définie dans les deux fichiers `LBMParams.cpp` et `LBM_enums.h` :

Dans `LBMParams.cpp`, on tient compte de l'option `UNIF_MIXTURE` :

```
if ( phase_field_init_str == "shrf_vortex" or phase_field_init_str == "SHRF_VORTEX" )
    phase_field_init = SHRF_VORTEX;
if ( phase_field_init_str == "unif_mixture" or phase_field_init_str == "UNIF_MIXTURE"
)
    phase_field_init = UNIF_MIXTURE;
```

Dans `LBM_enums.h`, on ajoute l'option `UNIF_MIXTURE` dans l'énumération `PhaseFieldInit` :

```
enum PhaseFieldInit {..., SHRF_VORTEX, UNIF_MIXTURE} ;
```

		Note Technique DES	Page 82/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

6.3 Tests

Pour créer l'exécutable qui tient compte des développements, on re-compile le code à l'aide des commandes décrites dans la section 1.3 ou 1.4 du chapitre 1. Par exemple, pour une exécution sur CPU, on se place dans le répertoire `build_openmp` puis on tape la commande `make`. Dans cette section on vérifie les développements sur le cas test de la déformation d'une interface dans un vortex (section 6.3.1). La solution de référence est celle obtenue avec le modèle CAC (section 4.3.2). Ensuite dans la section 6.3.2 on réalise une simulation de décomposition spinodale.

6.3.1 Vérification des développements du Cahn-Hilliard

La vérification consiste à retrouver les mêmes résultats entre ceux obtenus avec le modèle conservatif de Allen-Cahn et ceux obtenus avec les nouveaux développements relatifs au Cahn-Hilliard. Pour cela, on repart du jeu de données `test_lbm_d2q9_shrf_vortex.ini` qui se situe dans le répertoire `settings/phase_field`. On en crée une copie qu'on renomme `test_lbm_d2q9_shrf_vortex_ch.ini`. Dans ce fichier, à l'intérieur de la section `[lbm]`, on remplace Allen-Cahn par Cahn-Hilliard pour l'option `phase_field_model` et

Section `[lbm]`, modifier l'option du modèle :

```
[lbm]
problem=SHRF_VORTEX
phase_field_enabled=yes
phase_field_model=Cahn-Hilliard
```

Le modèle de Cahn-Hilliard est un peu différent de celui de Allen-Cahn conservatif car il contient la tension de surface. Pour les simulations, on prendra les valeurs des paramètres qui sont indiquées ci-dessous.

Section `[phase_field]`, définir les valeurs des paramètres :

```
[phase_field]
...
mobility=3e-2
sigma=1e-5
Width=0.03
```

Enfin, on modifie le préfixe du nom des fichiers de sortie `outputPrefix` :

Section `[output]`, modifier le préfixe :

```
[output]
write_variables=phi,vx,vy
outputPrefix=LBM_D2Q9_shrf_vortex_CH
outputVtkAscii=yes
```

Après exécution, les contours du champ de phase obtenus avec l'option Cahn-Hilliard ϕ^{CH} sont comparés sur la figure 6.1 à ceux obtenus avec l'option Allen-Cahn ϕ^{CAC} à trois temps différents.

6.3.2 Simulation avec la condition initiale aléatoire

Le modèle de Cahn-Hilliard permet de simuler la décomposition spinodale (ce que ne permet pas le modèle CAC) en modifiant la condition initiale. Pour cela on copie le fichier `test_lbm_d2q9_shrf_vortex_ch.ini` et on le recommande `test_lbm_d2q9_spin_decomp_ch.ini`.

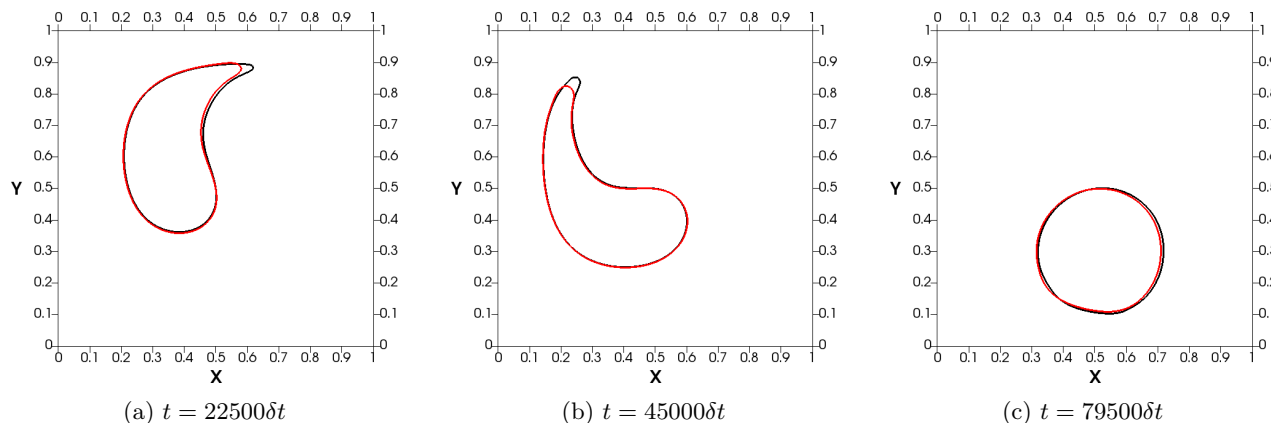



FIGURE 6.1 – Validation des développements du modèle de Cahn-Hilliard dans LBM_saclay : contour $\phi = 0.5$ pour le modèle CAC (ligne noire) et le modèle CH (ligne rouge) à (a) $t = 22500\delta t$, (b) $t = 45000\delta t$ et (c) $t = 79500\delta t$.

Dans `test_lbm_d2q9_spin_decomp_ch.ini`, on modifie les sections suivantes :

```
[run]
lbm_name=D2Q9
tEnd=50.0
nStepmax=480000
...
[lbm]
problem=UNIF_MIXTURE
phase_field_enabled=yes
phase_field_model=Cahn-Hilliard
...
[phase_field]
init=UNIF_MIXTURE
...
[output]
write_variables=phi,vx,vy
outputPrefix=LBM_D2Q9_spin_decomp
outputVtkAscii=yes
```

La simulation de décomposition spinodale est illustrée sur les figures 6.2 pour plusieurs temps.

		Note Technique DES	Page 84/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

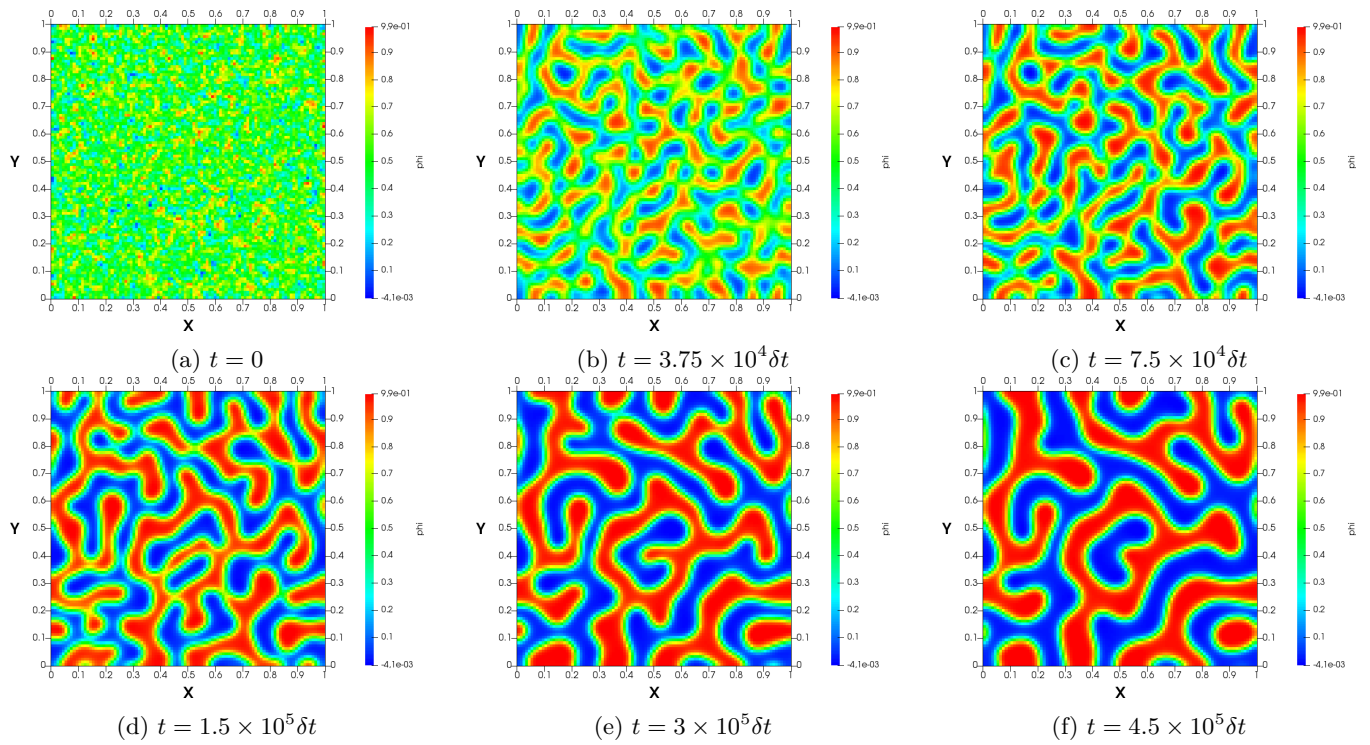



FIGURE 6.2 – Simulation de décomposition spinodale avec le modèle de Cahn-Hilliard pour plusieurs temps.

		Note Technique DES	Page 85/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Chapitre 7

Tutoriel sur un modèle de croissance cristalline

Dans ce chapitre on présente un exemple de couplage dans `LBM_saclay` entre deux équations : une équation pour le champ de phase et une équation pour la température. L'exemple est repris d'un modèle de croissance cristalline établi dans [19, 20]. On suppose dans ce chapitre que le tutoriel du chapitre précédent a été suivi. Ainsi on concentre la description sur les nouveautés liées au modèle et aux nouveaux ajouts à faire dans le fichier `LBMRun.h` pour introduire une nouvelle fonction de distribution. En suivant la même structure du chapitre précédent, la section 7.1 présente le modèle continu et l'algorithme LBM associé pour le simuler. La section 7.2 présente ensuite les nouveaux noyaux de calcul à écrire et les modifications à apporter dans `LBMRun.h`. La section décrit aussi les modifications à apporter aux fichiers `LBMPParams.cpp`, `LBMPParams.h` et `FieldManager.h`. Elle se termine par la description des conditions initiales. Enfin la section 7.3 présente une simulation.

7.1 Modèle à champ de phase pour la croissance cristalline

Le modèle de croissance cristalline qui est repris ici a été établi dans les références [19, 20].

7.1.1 Rappel du modèle continu

Le modèle à champ de phase de croissance cristalline est composé d'une équation sur le champ de phase ϕ qui est couplée à une équation sur la température adimensionnée notée θ :


$$\tau_0 a_s^2(\mathbf{n}) \frac{\partial \phi}{\partial t} = W_0^2 \nabla \cdot (a_s^2(\mathbf{n}) \nabla \phi) + W_0^2 \nabla \cdot \mathcal{N} + (\phi - \phi^3) - \lambda \theta (1 - \phi^2)^2, \quad (7.1)$$

$$\frac{\partial \theta}{\partial t} = \kappa \nabla^2 \theta + \frac{1}{2} \frac{\partial \phi}{\partial t}, \quad (7.2)$$

Dans ces équations, \mathbf{n} est le vecteur unitaire, normal à l'interface :

$$\mathbf{n}(\mathbf{x}, t) = - \frac{\nabla \phi}{|\nabla \phi|}, \quad (7.3)$$

et la fonction $a_s(\mathbf{n})$ est une fonction d'anisotropie qui est caractéristique de la forme des cristaux. Dans le cas de

		Note Technique DES	Page 86/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

la croissance dendritique équiaxe elle prend la forme simplifiée :

$$a_s(\mathbf{n}) = 1 - 3\varepsilon_s + 4\varepsilon_s \sum_{\alpha=x,y,z} n_\alpha^4. \quad (7.4)$$

où ε_s est un coefficient scalaire à spécifier dans le fichier d'entrée. En la fonction vectorielle \mathcal{N} est définie à partir de $a_s(\mathbf{n})$ par :

$$\mathcal{N}(\mathbf{x}, t) = |\nabla\phi|^2 a_s(\mathbf{n}) \left(\frac{\partial a_s(\mathbf{n})}{\partial(\partial_x\phi)}, \frac{\partial a_s(\mathbf{n})}{\partial(\partial_y\phi)}, \frac{\partial a_s(\mathbf{n})}{\partial(\partial_z\phi)} \right)^T. \quad (7.5)$$

Dans l'Eq. (7.1), les coefficients scalaires τ_0 , W_0 et λ sont respectivement le temps caractéristique, l'épaisseur de l'interface et le coefficient de couplage avec la température. Dans l'Eq. (7.2), le coefficient κ est la diffusivité thermique.

Comme conditions initiales, on supposera pour ϕ une sphère positionnée au centre d'un cube et une température uniforme négative dans tout le domaine.

7.1.2 Rappel de l'algorithme LBM

L'algorithme LBM est repris de la référence [6] à laquelle on se reportera pour davantage d'explications sur les détails de l'algorithme.

Equation du champ de phase

L'équation d'évolution s'écrit :

$$a_s^2(\mathbf{n})g_i(\mathbf{x}+\mathbf{e}_i\delta x, t+\delta t) = g_i(\mathbf{x}, t) - (1-a_s^2(\mathbf{n}))g_i(\mathbf{x}+\mathbf{e}_i\delta x, t) - \frac{1}{\eta_\phi(\mathbf{x}, t)} [g_i(\mathbf{x}, t) - g_i^{eq}(\mathbf{x}, t)] + w_i Q_\phi(\mathbf{x}, t) \frac{\delta t}{\tau_0}, \quad (7.6)$$

avec la fonction à l'équilibre :

$$g_i^{eq}(\mathbf{x}, t) = w_i \left(\phi(\mathbf{x}, t) - \frac{1}{e^2} \mathbf{e}_i \cdot \mathcal{N}(\mathbf{x}, t) \frac{\delta t}{\delta x} \frac{W_0^2}{\tau_0} \right). \quad (7.7)$$

Le terme source est défini par :


$$Q_\phi(\mathbf{x}, t) = [\phi - \lambda\theta(1 - \phi^2)] (1 - \phi^2). \quad (7.8)$$

Le taux de collision doit être recalculé à chaque pas de temps car il dépend de la fonction anisotrope $a_s^2(\mathbf{n})$ qui dépend au travers de \mathbf{n} du champ de phase ϕ :

$$\eta_\phi(\mathbf{x}, t) = \frac{1}{e^2} a_s^2(\mathbf{n}) \frac{W_0^2}{\tau_0} \frac{\delta t}{\delta x^2} + \frac{1}{2}. \quad (7.9)$$

Enfin le moment d'ordre zéro est mis à jour après les étapes de déplacement et de collision par :

$$\phi(\mathbf{x}, t) = \sum_{i=0}^{N_{pop}} g_i(\mathbf{x}, t) \quad (7.10)$$

		Note Technique DES	Page 87/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Equation de la thermique

L'équation d'évolution pour la température agit sur une nouvelle fonction de distribution s_i :

$$s_i(\mathbf{x} + \mathbf{e}_i \delta x, t + \delta t) = s_i(\mathbf{x}, t) - \frac{1}{\eta_\theta} [s_i(\mathbf{x}, t) - s_i^{eq}(\mathbf{x}, t)] + w_i Q_\theta(\mathbf{x}, t) \delta t. \quad (7.11)$$

où la fonction d'équilibre s_i^{eq} est celle qui reproduit la diffusion standard :

$$s_i^{eq}(\mathbf{x}, t) = w_i \theta(\mathbf{x}, t), \quad (7.12)$$

Le terme source fait apparaître la dérivée temporelle du champ de phase ϕ :

$$Q_\theta(\mathbf{x}, t) = \frac{1}{2} \frac{\partial \phi}{\partial t} \quad (7.13)$$

La diffusivité thermique κ est reliée au taux de collision par la relation classique :

$$\kappa = e^2 \left(\eta_\theta - \frac{1}{2} \right) \frac{\delta x^2}{\delta t}. \quad (7.14)$$

Enfin la température est mise à jour en calculant le moment d'ordre zéro de la fonction de distribution h_i :

$$\theta(\mathbf{x}, t) = \sum_{i=0}^{N_{pop}} s_i(\mathbf{x}, t). \quad (7.15)$$


7.1.3 Quelques remarques

Équation d'évolution pour g_i . Dans l'Éq. (7.6), le coefficient $a_s^2(\mathbf{n})$ apparaît en facteur de $g_i(\mathbf{x} + \mathbf{e}_i \delta x, t + \delta t)$ et un nouveau terme $-(1 - a_s^2(\mathbf{n}))g_i(\mathbf{x} + \mathbf{e}_i \delta x, t)$ est ajouté dans le membre de droite. Comparé à l'algorithme standard, ces deux termes supplémentaires permettent de tenir compte du facteur $a_s^2(\mathbf{n})$ devant le terme $\partial \phi / \partial t$ dans l'Éq. (7.1). En considérant que $a_s(\mathbf{n})$ est explicite, l'étape de déplacement sera simplement formulée par :

$$g_i(\mathbf{x} + \mathbf{e}_i \delta x, t + \delta t) = \frac{1}{a_s^2(\mathbf{n})} \left\{ g_i(\mathbf{x}, t) - (1 - a_s^2(\mathbf{n}))g_i(\mathbf{x} + \mathbf{e}_i \delta x, t) - \frac{1}{\eta_\phi(\mathbf{x}, t)} [g_i(\mathbf{x}, t) - g_i^{eq}(\mathbf{x}, t)] + w_i Q_\phi(\mathbf{x}, t) \frac{\delta t}{\tau_0} \right\},$$

Composantes du vecteur $\mathcal{N}(\mathbf{x}, t)$. Dans la fonction d'équilibre Eq. (7.7), le second terme dans la parenthèse fait apparaître le produit scalaire entre les vecteurs de déplacement et le vecteur $\mathcal{N}(\mathbf{x}, t)$ à la place de la vitesse d'advection \mathbf{u} . En utilisant la relation Eq. (7.4) pour $a_s(\mathbf{n})$ on peut calculer analytiquement chaque composante du vecteur $\mathcal{N}(\mathbf{x}, t)$ (voir [21, Eq. (18)]) :

$$\frac{\partial a_s(\mathbf{n})}{\partial (\partial_\alpha \phi)} = -\frac{16\varepsilon_s}{|\nabla \phi|^6} \times (\partial_\alpha \phi) \left[(\partial_\beta \phi)^4 - (\partial_\alpha \phi)^2 (\partial_\beta \phi)^2 - (\partial_\alpha \phi)^2 (\partial_\gamma \phi)^2 + (\partial_\gamma \phi)^4 \right]. \quad (7.16)$$

		Note Technique DES	Page 88/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Si on écrit explicitement chaque composante, on obtient en posant :

$$\frac{\partial a_s(\mathbf{n})}{\partial(\partial_x \phi)} = -\frac{16\varepsilon_s}{|\nabla \phi|^6} \times (\partial_x \phi) \left[(\partial_y \phi)^4 - (\partial_x \phi)^2 (\partial_y \phi)^2 - (\partial_x \phi)^2 (\partial_z \phi)^2 + (\partial_z \phi)^4 \right] \quad (7.17a)$$

$$\frac{\partial a_s(\mathbf{n})}{\partial(\partial_y \phi)} = -\frac{16\varepsilon_s}{|\nabla \phi|^6} \times (\partial_y \phi) \left[(\partial_x \phi)^4 - (\partial_y \phi)^2 (\partial_x \phi)^2 - (\partial_y \phi)^2 (\partial_z \phi)^2 + (\partial_z \phi)^4 \right] \quad (7.17b)$$

$$\frac{\partial a_s(\mathbf{n})}{\partial(\partial_z \phi)} = -\frac{16\varepsilon_s}{|\nabla \phi|^6} \times (\partial_z \phi) \left[(\partial_x \phi)^4 - (\partial_z \phi)^2 (\partial_x \phi)^2 - (\partial_z \phi)^2 (\partial_y \phi)^2 + (\partial_y \phi)^4 \right] \quad (7.17c)$$

Chacune des composantes ci-dessus doit être multipliée par $|\nabla \phi|^2 a_s(\mathbf{n})$ en facteur de la définition Eq. (7.5).

Calcul du taux de collision η_ϕ à chaque pas de temps. Le taux de collision pour l'équation d'évolution Eq. (7.9) doit être mis à jour à chaque pas de temps car il est fonction de la fonction $a_s(\mathbf{n})$ qui dépend de la normale à l'interface \mathbf{n} qui varie au cours du temps.

7.2 Mise en œuvre dans LBM_saclay

Le modèle mathématique est composé de deux EDPs couplées, la première pour le champ de phase ϕ et la seconde pour la température θ . On crée deux noyaux de calcul, chacun dédié à la mise à jour de chaque grandeur macroscopique. On présente les développements à faire pour réaliser des simulations en 3D.


Pour l'équation du champ de phase, le plus simple est de repartir une nouvelle fois du noyau de calcul du Allen-Cahn conservatif : `CollideAndStream_PhaseField_Functor.h`. Ce fichier contient la classe qui résout déjà une équation du champ de phase et le calcul de la normale à l'interface \mathbf{n} . On fait une copie de ce fichier qu'on renomme en `CollideAndStream_KarmaRappel_Phi_Functor.h`. Ce nouveau fichier (section 7.2.1) contiendra la nouvelle classe `CollideAndStream_Karma_Rappel_Phi_Functor` avec son constructeur. Elle sera munie d'une fonction `apply`, d'une fonction `void operator()` et de plusieurs fonctions précédées par la décoration `KOKKOS_INLINE_FUNCTION`. Pour l'équation en température, on crée une copie du fichier `CollideAndStream_transport_Functor.h` qu'on renomme `CollideAndStream_KarmaRappel_Tpr_Functor.h`. Ce fichier (section 7.2.2) contiendra la classe qui lui est associée et les mêmes fonctions `apply` et `void operator()`. Même si trois fonctions de distributions différentes sont déjà allouées en mémoire (`f0`, `g0` et `h0`), il est préférable d'en introduire une nouvelle, `s0` dédiée aux problèmes de transfert de chaleur (`heat_transfer`). On décrira comment ajouter cette nouvelle fonction `s0` et la mettre à jour au cours des itérations dans `LBMRun.h` dans la section 7.2.3.

7.2.1 Nouveau noyau `CollideAndStream_KarmaRappel_Phi_Functor.h`

Dans ce nouveau noyau on commence par définir une structure des paramètres du modèle (section 7.2.1.1). Ensuite on décrit dans la section 7.2.1.2 les instructions pour calculer successivement les composantes du $\nabla \phi$ qui apparaît dans la définition du vecteur normal \mathbf{n} (Eq. (7.3)), la fonction anisotrope $a_s(\mathbf{n})$ (Eq. (7.4)) est décrite dans la section 7.2.1.3 et le vecteur \mathcal{N} (Eq. (7.5)) dans la section 7.2.1.4. Enfin on détaillera dans la section 7.2.1.5 les trois fonctions `void operator()` contenues dans la classe.

7.2.1.1 Nouvelle structure de paramètres `KR_params`

On introduit tout d'abord une nouvelle structure de paramètres relatifs au modèle de Karma-Rappel : `KR_params`. Les paramètres sont W_0 (`KR_Width0`), τ_0 (`KR_Tau0`), λ (`KR_Lambda`), ε_s (`KR_Eps`) pour l'équation du champ de phase et κ (`KR_Diffusivity`) pour l'équation de la température.

		Note Technique DES	Page 89/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

```
#include "LBM_Base_Functor.h"
struct KR_Params {real_t KR_Width0, KR_Tau0, KR_Lambda, KR_Eps, KR_Diffusivity};
```

Il faut ensuite passer en argument cette nouvelle structure de paramètres `KR_Params` et `krParams` dans le constructeur de la classe, l'initialisation, la déclaration des membres de la classe, la fonction `apply`, etc ... On ne le montre pas ici, on se référera à la section 6.2.1.1 qui a détaillé cette étape pour les paramètres du modèle de Cahn-Hilliard (`CH_Params` et `chParams`).

7.2.1.2 Dans la fonction `compute_gradient_phi`

Comme on a fait une copie du fichier `CollideAndStream_PhaseField_Functor.h`, il existe déjà une fonction dédiée au calcul du vecteur unitaire \mathbf{n} . On renomme cette fonction en `compute_gradient_phi` et déclarée par :

```
KOKKOS_INLINE_FUNCTION
void compute_gradient_phi (real_t& dPhidx, real_t& dPhidy, real_t& dPhidz,
                          int i, int j, int k, int isize, int jsize, int ksize, real_t dx, real_t e2) const
{...}
```

Les ... dans les accolades sont les instructions qui calculent les composantes du gradient, puis celles du vecteur \mathbf{n} en divisant par la norme. On commentera les lignes qui calculent les composantes de \mathbf{n} pour ne conserver dans cette fonction que le calcul des composantes du gradient de ϕ `dPhidx`, `dPhidy` et `dPhidz`.

7.2.1.3 Nouvelle fonction `compute_anisotropy_function_As_STD`


Une fois que le gradient est connu, on peut calculer la fonction d'anisotropie $a_s(\mathbf{n})$ définie par l'Eq. (7.4). Pour cela, on définit une nouvelle fonction `compute_anisotropy_function_As_STD` déclarée par

```
KOKKOS_INLINE_FUNCTION
real_t compute_anisotropy_function_As_STD (real_t dPhidx, real_t dPhidy,
                                           real_t dPhidz) const {...}
```

Dans le nom de la fonction, `STD` précise qu'il s'agit de la fonction `STanDard` pour une croissance équiaxe. D'autres formes plus complexes de cette fonction existent (voir par exemple [6, Eq. (11) ou Eq. (12)]) et pourront être définies ultérieurement. En entrée la fonction doit connaître le paramètre ε_s et les composantes du gradient de ϕ préalablement calculées `dPhidx`, `dPhidy` et `dPhidz`. À l'intérieur de la fonction on utilise le paramètre ε_s de la structure `krParams`. Les principales instructions sont :

```
real_t Eps = krParams.KR_Eps ;
real_t As_n ;
real_t norm = sqrt( sqr(dPhidx) + sqr(dPhidy) + sqr(dPhidz) );
if (norm > 0.0) {
    ...
    As_n = 1.0 - 3.0*Eps + 4.0*Eps*(xx4 + yy4 + zz4)/ww4 ;
} else {
    As_n = 1.0 - 3.0*Eps ;
}
return As_n ;
```

Les ... représentent les instructions intermédiaires pour définir `xx4`, `yy4`, `zz4` et `ww4` à partir de `xx=-dPhidx`, `yy=-dPhidy`, `zz=-dPhidz`, et `ww=norm`. Pour élever successivement au carré puis à la puissance quatre chaque terme, on utilisera la fonction `sqr()` déjà définie dans la classe.

		Note Technique DES	Page 90/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

7.2.1.4 Nouvelle fonction `compute_anisotropy_vector_N`

On définit une nouvelle fonction `compute_anisotropy_vector_N` pour calculer les trois composantes du vecteur $\mathcal{N}(\mathbf{x}, t)$ (Eq. (7.5)) où les trois composantes sont respectivement définies par les Eqs. (7.17a), (7.17b) et (7.17c) :

```
KOKKOS_INLINE_FUNCTION
void compute_anisotropy_vector_N (real_t& CompNx, real_t& CompNy, real_t& CompNz,
                                   real_t dPhidx, real_t dPhidy, real_t dPhidz,
                                   real_t As_n ) const
{...}
```

Cette fonction calcule les composantes `CompNx`, `CompNy` et `CompNz` en utilisant comme arguments les composantes du gradient de `phi` et la fonction anisotrope `As_n`. Les principales étapes sont les suivantes :

```
real_t Eps = krParams.KR_Eps ;
real_t factor = -16.0*Eps ;
real_t norm = sqrt( sqr(dPhidx) + sqr(dPhidy) + sqr(dPhidz) );
...
if (norm > 0.0) {
    CompNx = factor*xx * (yy4 - xx2*yy2 - xx2*zz2 + zz4) / ww6 ;
    CompNy = factor*yy * (xx4 - yy2*xx2 - yy2*zz2 + zz4) / ww6 ;
    CompNz = factor*zz * (xx4 - zz2*yy2 - zz2*xx2 + yy4) / ww6 ;
}
CompNx *= norm*norm * As_n ;
CompNy *= norm*norm * As_n ;
CompNz *= norm*norm * As_n ;
```

Comme précédemment les ... représentent les instructions intermédiaires pour définir à partir de `xx=dPhidx` (resp. `yy` et `zz`), le carré `xx2` et la puissance quatre et `xx4` (resp. pour `yy2`, `yy4`, `zz2` et `zz4`). On définit également la puissance six de la norme à partir de `ww=norm`.


7.2.1.5 Les fonctions `void operator()`

On écrit trois fonctions `void operator()`, la première munie du tag `TagCollideAndStream`, la seconde munie du tag `TagUpdateMacro` et la dernière avec le tag `TagComputeFeq`.

Dans `void operator()` avec `TagCollideAndStream`, déclarée par

```
template<int dim_ = dim>
KOKKOS_INLINE_FUNCTION void operator()(const TagCollideAndStream&,
                                       const typename std::enable_if<dim_==3, int>::type& index) const {...}
```

où les ... sont les intructions décrites ci-dessous. Tout d'abord on appelle les trois fonctions définies dans les sections 7.2.1.2, 7.2.1.3 et 7.2.1.4 :

			Note Technique DES	Page 91/116
	<u>Réf</u> : STMF/LMSF/NT/2022-70869			
			<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.			

```

if (i>0 and i < isize-1 and j>0 and j < jsize-1 and k>0 and k < ksize-1 ) {
    real_t dPhidx=0, dPhidy=0, dPhidz=0;
    compute_gradient_phi (dPhidx, dPhidy, dPhidz,
                          i, j, k, isize, jsize, ksize, dx, e2);
// compute anisotropy function a_s(n) and its square
    real_t As_n = compute_anisotropy_function_As_STD (dPhidx,dPhidy,dPhidz) ;
    real_t As2 = As_n * As_n;
// Compute anisotropy vector N
    real_t CompNx=0, CompNy=0, CompNz=0;
    compute_anisotropy_vector_N (CompNx, CompNy, CompNz,
                                dPhidx, dPhidy, dPhidz, As_n ) ;

```

Puis on calcule la fonction de distribution à l'équilibre en ayant préalablement défini les paramètres $W0$ et $Tau0$ avec `krParams` :

```

FState f, feq;
const real_t phi = lbm_data(i,j,k,fm[IPHI]);
const real_t fact = W0*W0/Tau0 ;
for (int ipop=0; ipop<npop; ++ipop) {
    f[ipop] = f0(i,j,k, ipop);
    real_t scal1 = E[ipop][IX]*CompNx + E[ipop][IY]*CompNy + E[ipop][IZ]*CompNz;
    feq[ipop] = w[ipop] * (phi - e2*scal1*(dt/dx)*fact) ;
}

```

Ensuite on réalise les étapes de collision et de déplacement en remarquant qu'on utilise la fonction de distribution `f0` à droite de l'égalité pour tenir compte des nœuds voisins :

```

real_t tau_phi = (3.0 * As2 * fact * dt / (dx * dx)) + 0.5 ;
const real_t Tempr = lbm_data(i,j,k,fm[ITPR]);
real_t SourceTerm = (phi - Lambda*Tempr*(1.0-phi*phi))*(1.0-phi*phi) ;
for (int ipop=0; ipop<npop; ++ipop) {
    if (i+E[ipop][IX] >= 0 and i+E[ipop][IX] < params.isize and
        j+E[ipop][IY] >= 0 and j+E[ipop][IY] < params.jsize and
        k+E[ipop][IZ] >= 0 and k+E[ipop][IZ] < params.ksize) {
        f1(i+E[ipop][IX], j+E[ipop][IY], k+E[ipop][IZ],ipop) = (1/As2) *
            (f[ipop] - (1.0 - As2)*f0(i+E[ipop][IX], j+E[ipop][IY], k+E[ipop][IZ],ipop)
             - ( f[ipop] - feq[ipop] ) / tau_phi
             + w[ipop]*SourceTerm*dt/Tau0);
    }
}


```

Dans `void operator()` avec `TagComputeFeq`, déclarée par

```

template<int dim_ = dim>
KOKKOS_INLINE_FUNCTION
void operator()(const TagComputeFeq&,
               const typename std::enable_if<dim_==3, int>::type& index) const {...}

```

		Note Technique DES	Page 92/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

on reprend les mêmes instructions que celles de `void operator()` avec `TagCollideAndStream` pour la mise à jour de la fonction à l'équilibre sans faire le déplacement ni la collision.

Dans `void operator()` avec `TagUpdateMacro`, déclarée par

```
template<int dim_ = dim>
KOKKOS_INLINE_FUNCTION
void operator()(const TagUpdateMacro&,
               const typename std::enable_if<dim_==3, int>::type& index) const {...}
```

après avoir réalisé la déclaration préliminaire des variables `i`, `j`, `k`, `isize`, `jsize`, `ksize`, `dt` et appelé la fonction `index2coord`, on met à jour le calcul de la nouvelle variable macroscopique ϕ sans oublier de mettre à jour la dérivée $\partial\phi/\partial t$ qui apparaît dans le terme source de l'équation de la chaleur :

```
if (i>0 and i < isize-1 and j>0 and j < jsize-1 and k>0 and j < ksize-1) {
    const real_t phi_prev = lbm_data (i, j, k, fm[IPHI]);
    real_t phi = 0.0;
    for (int ipop = 0 ; ipop < npop ; ++ipop) {
        phi += f1(i, j, k, ipop);
    }
    lbm_data(i,j,k,fm[IPHI]) = phi ;
    lbm_data(i,j,k,fm[DPHIDT]) = (phi-phi_prev)/dt;
} // end if
```

Comme pour le potentiel chimique μ_ϕ du chapitre 6, on tiendra compte du nouvel indice `DPHIDT` du tableau `lbm_data` dans le fichier `FieldManager.h`.

7.2.2 Nouveau noyau `CollideAndStream_KarmaRappel_Tpr_Functor.h`

Le noyau de calcul `CollideAndStream_KarmaRappel_Tpr_Functor.h` est dédié au calcul du champ de température θ qui obéit une EDP de type « advection-diffusion » avec un terme source. Le noyau LBM d'une telle équation a déjà été décrit dans le chapitre 4 (section 4.2). On pourra reprendre cet exemple pour l'adapter à l'équation Eq. (7.2). Dans cette équation le terme advectif est nul ce qui simplifie la fonction d'équilibre. Par ailleurs le terme $\partial\phi/\partial t$ a déjà été calculé dans le noyau de calcul `CollideAndStream_KarmaRappel_Tpr_Functor.h` et stocké dans `lbm_data` avec l'indice `DPHIDT`.


Comme pour le noyau `CollideAndStream_KarmaRappel_Phi_Functor.h`, la classe devra contenir trois fonctions `void operator()` munies des tags `TagCollideAndStream`, `TagUpdateMacro`, et `TagComputeFeq`. Dans la fonction munie du tag `TagUpdateMacro`, on stockera le champ dans `lbm_data` avec un nouvel indice `ITPR`.

7.2.3 Ajouts dans le fichier `LBMRun.h`

Ces deux nouveaux noyaux doivent être inclus dans `LBMRun.h` par la commande `#include`. Dans la section 7.2.3.1 on décrit les instructions à ajouter dans `update_phase_field`. Dans la section 7.2.3.2 on décrit comment ajouter une nouvelle fonction de distribution pour la température `s0` puis on décrit la nouvelle fonction `update_heat_transfer`.

7.2.3.1 Ajouts pour le noyau « champ de phase »

Pour tenir compte du noyau de calcul relatif au champ de phase, il s'agit principalement de tenir compte d'une nouvelle option de calcul dans `update_phase_field`.

		Note Technique DES	Page 93/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Dans `update_phase_field`, on introduit l'option suivante :

```
if (phase_field_model == "Karma-Rappel") {
...
}
```

où les « ... » sont les instructions détaillées ci-dessous. On commence par lire les valeurs des paramètres depuis le jeu de données :

```
real_t KR_Width0 = configMap.getFloat("phase_field","KR_Width0",0.01);
real_t KR_Tau0 = configMap.getFloat("phase_field","KR_Tau0",0.01);
real_t KR_Lambda = configMap.getFloat("phase_field","KR_Lambda",0.01);
real_t KR_Eps = configMap.getFloat("phase_field","KR_Eps",0.01);
real_t KR_Diffusivity = configMap.getFloat("heat_transfer","KR_Diffusivity",0.01);
```

puis on les garde en mémoire dans la structure `krParams` de type `KR_Params` et déclarée dans :

```
KR_Params krParams = {.KR_Width0=KR_Width0, .KR_Tau0=KR_Tau0, .KR_Lambda=KR_Lambda,
.KR_Eps=KR_Eps, .KR_Diffusivity=KR_Diffusivity};
```

Ensuite on applique la fonction `apply` munie du tag qui précise la collision et le déplacement :

```
auto tag1=CollideAndStream_KR_Phi_Functor<dim,npop>::TAG_COLLIDE_AND_STREAM;
CollideAndStream_KR_Phi_Functor<dim,npop>::apply(params, fm, data, f_in, f_out,
krParams, tag1);
```

Après cette étape de collision et de déplacement, il faut mettre à jour les conditions aux limites :

```
make_boundaries(f_out);
```


Enfin on calcule le moment d'ordre 0 (ϕ) et sa dérivée temporelle ($\partial\phi/\partial t$) en appelant une nouvelle fois la fonction `apply` mais avec le tag `TAG_UPDATE_MACRO_VAR` :

```
auto tag2 = CollideAndStream_KR_Phi_Functor<dim,npop>::TAG_UPDATE_MACRO_VAR;
CollideAndStream_KR_Phi_Functor<dim, npop>::apply(params, fm, data, f_in, f_out,
krParams, tag2);
```

7.2.3.2 Ajouts pour le noyau « température »

Nouveau tableau `s0` : l'algorithme LBM pour la température introduit une nouvelle fonction de distribution s_i . Pour en tenir compte dans `LBMRun.h`, on déclare un nouveau tableau `s0` de type `FArray` dédié au transfert de chaleur :

```
FArray h0; /*!< LBM distribution function array - transport */
FArray s0; /*!< LBM distribution function array - heat transfer */
FArray f_tmp; /*!< LBM distribution function array */
```

			Note Technique DES	Page 94/116
	Réf : STMF/LMSF/NT/2022-70869			
			Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.			

qui devra ensuite être alloué en mémoire en introduisant l'option `lbm_heat_transfer_enabled` (ex. en 3D) :

```
if (params.lbm_heat_transfer_enabled)
    s0 = FArray("s0" , isize, jsize, ksize, NPOP);
```

Ensuite dans la fonction `update`, on réalisera la mise à jour d'une nouvelle fonction `update_heat_transfer` en ajoutant l'instruction :

```
if (params.lbm_heat_transfer_enabled)
    update_heat_transfer(s0, f_tmp);
```

La nouvelle fonction `update_heat_transfer` est déclarée par

```
void update_heat_transfer(FArray& f_in, FArray& f_out) {
    ...
}
```

où les « ... » sont les instructions détaillées ci-dessous. Elles sont calquées sur le `update_phase_field`, c'est-à-dire qu'on appelle deux fois la fonction `apply` avec deux tags différents, le premier pour le « collide and stream » :

```
real_t KR_Diffusivity = configMap.getFloat("heat_transfer","KR_Diffusivity",0.01);
auto tag1=CollideAndStream_KR_Tpr_Functor<dim,npop>::TAG_COLLIDE_AND_STREAM;
CollideAndStream_KR_Tpr_Functor<dim,npop>::apply(params, fm, data, f_in, f_out,
                                                KR_Diffusivity, tag1);
```

et le second pour le « update macro » :

```
auto tag2 = CollideAndStream_KR_Tpr_Functor<dim,npop>::TAG_UPDATE_MACRO_VAR;
CollideAndStream_KR_Tpr_Functor<dim, npop>::apply(params, fm, data, f_in, f_out,
                                                KR_Diffusivity, tag2);
```


7.2.4 Ajouts dans les fichiers `LBMPParams.cpp`, `LBMPParams.h` et `FieldManager.h`

Enfin on tient compte des options de calcul dans les fichiers `LBMPParams.cpp`, `LBMPParams.h` et `FieldManager.h`.

Dans `LBMPParams.cpp`, on ajoute la lecture de l'option `lbm_heat_transfer_enabled` :

```
lbm_transport_enabled = configMap.getBool("lbm","transport_enabled",false);
lbm_heat_transfer_enabled = configMap.getBool("lbm","heat_transfer_enabled",false);
lbm_phase_field_enabled = configMap.getBool("lbm","phase_field_enabled",false);
```

On ajoute ensuite l'option pour la condition initiale :

		Note Technique DES	Page 95/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

```
std::string heat_transfer_init_str = std::string(configMap.getString("heat_transfer",
"init", "unknown"));
if ( heat_transfer_init_str == "unif" or
    heat_transfer_init_str == "UNIF" )
    heat_transfer_init = UNIFORM_VALUE;
```

Dans **LBMPParams.h**, le booléen `lbm_heat_transfer_enabled` est déjà déclaré et il apparaît dans `LBMPParams()`. On ajoutera simplement la variable `heat_transfer_init` dédiée à la condition initiale :

```
int fluid_init;
int phase_field_init;
int heat_transfer_init;
```

et

```
LBMPParams() :
...
    phase_field_init(FLUID_INIT_UNDEFINED),
    heat_transfer_init()
{}
```

Dans **FieldManager.h**, on tient compte des nouveaux champs θ et $\partial\phi/\partial t$ en ajoutant :

```
if (params.lbm_heat_transfer_enabled) {
    var_enabled[ITPR] = 1;
    var_enabled[DPHIDT] = 1; }
```

Puis sur le même modèle que le potentiel chimique μ_ϕ et l'indice IMU, on ajoutera les lignes qui correspondent à ITPR et DPHIDT dans les fonctions `static int2str_t get_id2names_all()` et `static str2int_t get_names2id_all()`.


7.2.5 Conditions initiales

7.2.5.1 Condition initiale pour ϕ : fichier **InitPhi_Shrf_Vortex.h**

La condition qui initialise une sphère diffuse en 3D est déjà programmée dans le fichier **InitPhi_Shrf_Vortex.h**. Par contre cette condition initiale est prévue pour le modèle conservatif de Allen-Cahn où le champ de phase varie entre 0 et +1. Dans le modèle de solidification de Karma-Rappel, le champ de phase est défini tel qu'il varie entre -1 et +1. On en tient compte dans le fichier **InitPhi_Shrf_Vortex.h**. Cette condition initiale calcule un champ de vitesse à partir d'une valeur U_0 indiquée dans le jeu de données. On prendra $U_0 = 0$.

Dans **InitPhi_Shrf_Vortex.h**, on modifie la ligne suivante :

```
data(i,j,k, fm[IPHI]) = tanh( (r0 - sqrt( sqr(x-x0) + sqr(y-y0) +
                                         sqr(z-z0)) )/ sqrt(2.0) / spread );
```

		Note Technique DES	Page 96/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.			

7.2.5.2 Condition initiale pour θ : nouveau fichier UniformValue.h

Pour la condition initiale en température, une façon de procéder est de créer une nouvelle classe `UniformValue_Functor` dans un nouveau fichier `UniformValue.h` qui pourra s'inspirer de `GaussianHill.h`. Cette classe possèdera sa fonction `apply` et son `void operator()`.

Dans `LBMRun.h`, on définit une nouvelle fonction `init_condition_heat_transfer` pour la condition initiale. Cette fonction appelle la fonction `apply` puis initialise la fonction à l'équilibre `s0` :

```
template<int dim, int npop>
void LBMRun<dim, npop>::init_condition_heat_transfer (LBMArrary ldata)
{
    if ( params.heat_transfer_init == UNIFORM_VALUE ) {
        UniformValue_Functor<dim, npop>::apply(configMap, params, fm, ldata);
    }
    Compute_Feq_transport_Functor<dim, npop>::apply(params, fm, ldata, s0);
    Kokkos::deep_copy(f_tmp, s0);
}
```

7.3 Tests

On repart du cas test `test_lbm_d3q19_shrf_vortex.ini` en copiant le fichier et en le renommant `test_lbm_d3q19_crystal_growth.ini`. Dans la section `[run]`, on indiquera `nx=ny=nz=300` avec un pas de discrétisation en espace égal à $\delta x = 0.4$ ce qui correspond à des positions `xmin=ymin=zmin=0` et `xmax=ymax=zmax=120`. Le pas de temps, le temps final de simulation, le nombre d'itération max, et le nombre de sauvegardes (écriture des fichiers) seront fixés à `dt=0.008`, `tEnd=80`, `nStepmax=10000` et `nOutput=100`.

Dans les section `[lbm]`, et `[phase_field]` on indique les options suivantes :

```
[lbm]
gamma0=1.666
problem=SHRF_VORTEX
phase_field_enabled=yes
phase_field_model=Karma-Rappel
fluid_enabled=no
heat_transfer_enabled=yes
transport_enabled=no
```

```
[phase_field]
init=SHRF_VORTEX
collision=BGK
coupling_with_fluid=no
KR_Width0=1.0
KR_Tau0=1.0
KR_Lambda=6.3826
KR_Eps=0.05
```

Ensuite dans les sections `[shrf_vortex]` on indique les paramètres d'une sphère diffuse positionnée au entre du domaine de calcul. Enfin dans la section `[heat_transfer]` indique les paramètres de l'équation de la température.

```
[shrf_vortex]
x0=60.0
y0=60.0
z0=60.0
r0=4.0
spread=0.5
U0=0.0
```

```
[heat_transfer]
init=UNIF
KR_Diffusivity=4.0
Tpr_Init=-0.55
vx=0.0
vy=0.0
```

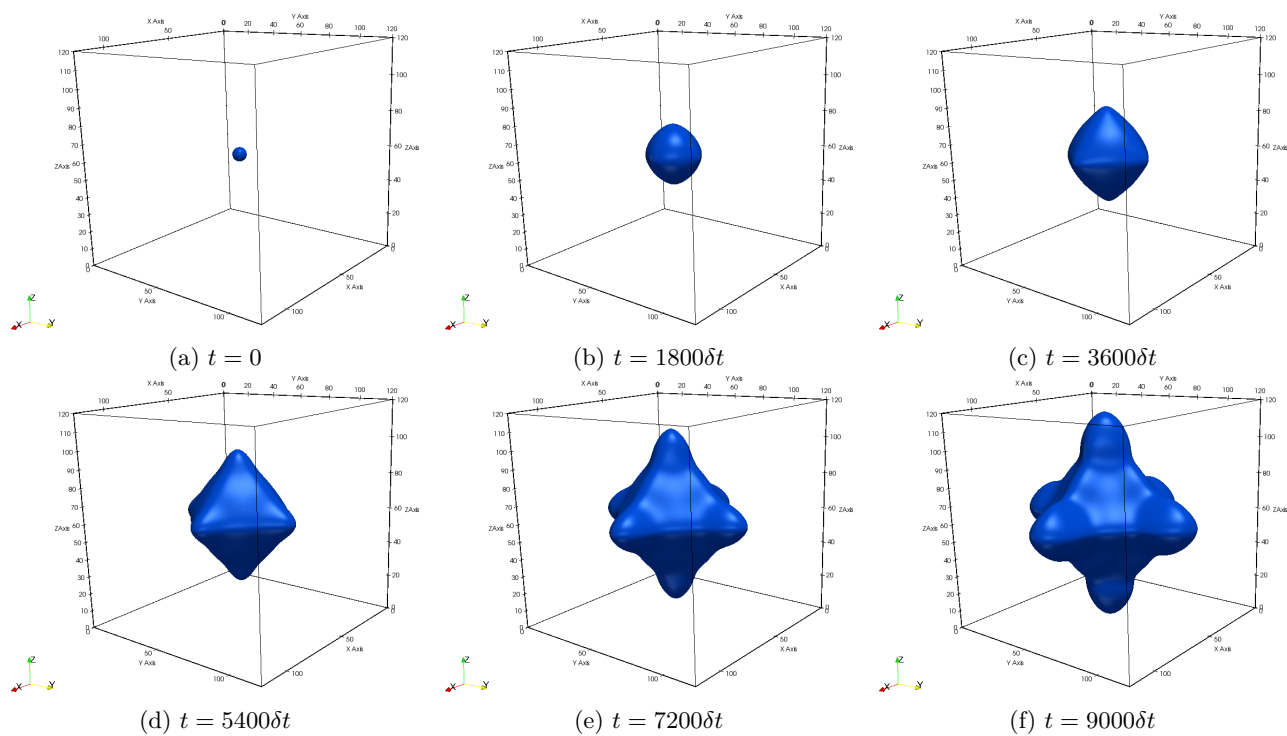




FIGURE 7.1 – Évolution du champ de phase $\phi = 0.5$ en plusieurs temps.

		Note Technique DES	Page 98/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

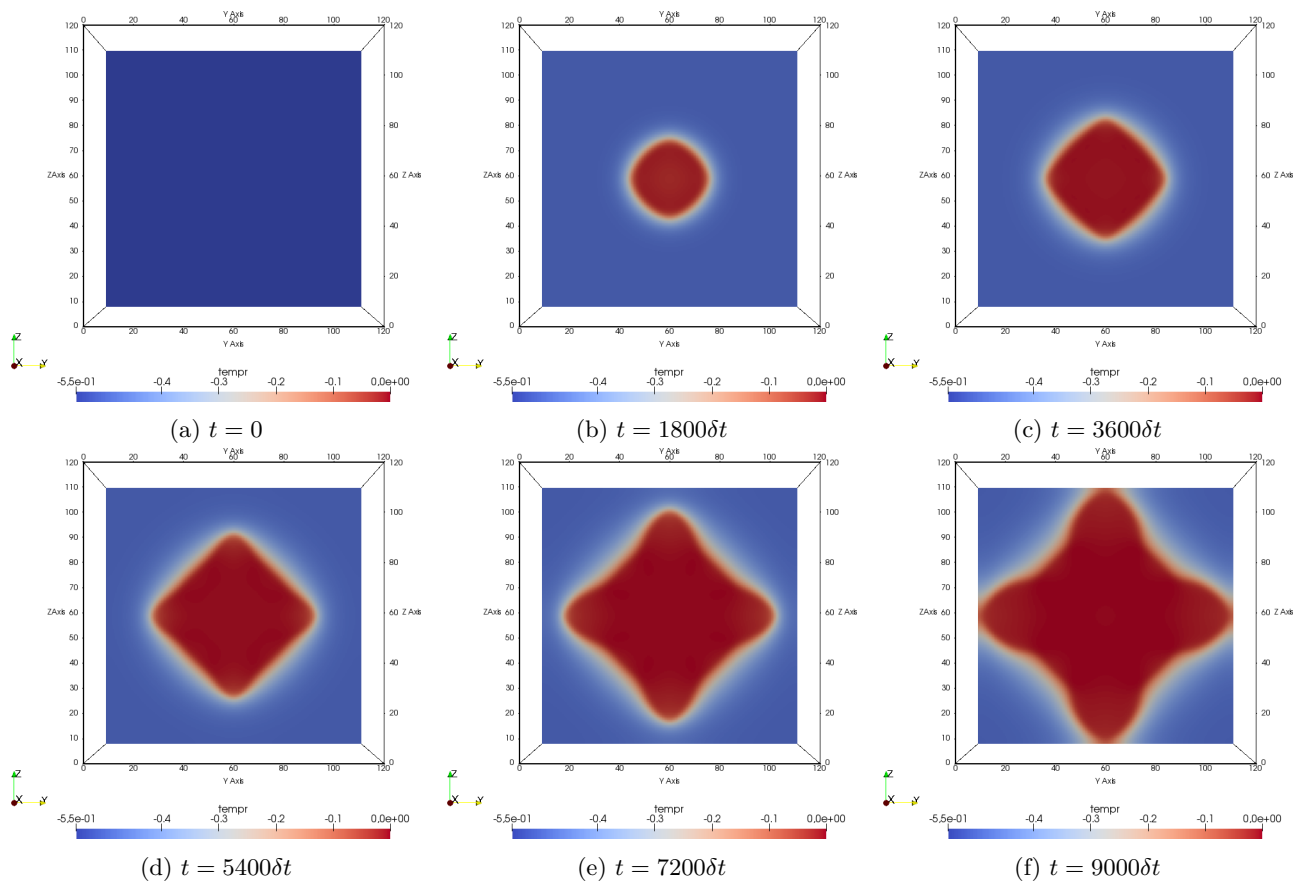




FIGURE 7.2 – Évolution du champ de température (plan yz , coupe en $x = 60$) en plusieurs temps.

		Note Technique DES	Page 99/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Troisième partie

Conclusion – Perspectives

		Note Technique DES	Page 101/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Conclusion


Ce rapport documente le code de calcul **LBM_saclay**, application logicielle dédiée aux simulations HPC multi-architectures sur base LBM. Cette documentation est séparée en deux parties distinctes. La première est descriptive. Elle détaille les éléments de base de la LBM tout en les associant aux lignes de code C++ et en décrivant les fichiers sources associés (réseaux 2D et 3D). Les principales étapes de l'algorithme sont décrites pour chaque modèle mathématique présent dans la version « master ». La seconde partie détaille deux tutoriels, qui sont présentés dans deux chapitres différents. Ces tutoriels détaillent pas à pas la manière de développer son propre modèle dans **LBM_saclay**. Le premier exemple est donné sur l'équation classique de Cahn-Hilliard. Le second exemple est celui d'un modèle de croissance cristalline qui couple deux EDPs, une pour le champ de phase et une autre pour la température. Ces deux tutoriels ont été refaits avec succès par plusieurs étudiants venus se former à la LBM et au HPC et leur ont permis de prendre en main le code source.


Travaux en cours

Afin de faciliter les développements de problèmes physiques qui impliquent plusieurs EDPs couplées (trois ou plus), un nouveau modèle de programmation a été mis en œuvre qui a conduit à une ré-organisation de cette version « master ». En effet, même si elle permet déjà de simuler des équations couplées (voir le second tutoriel), les fichiers à modifier et les différents endroits du code à adapter sont nombreux lorsque le problème est multi-physique, par exemple pour un problème physique modélisé par les équations de Navier-Stokes couplées au champ de phase et à une ou deux EDPs sur les compositions. La complexité vient de la combinatoire entre les différents termes sources des EDPs, des conditions initiales différentes par équations, des paramètres de ces équations et de leurs interpolations et des méthodes numériques utilisées pour simuler le modèle. Pour les développeurs novices, il est plus aisé de regrouper dans un unique répertoire un nombre limité de fichiers où apparaissent clairement les modifications à apporter pour développer son propre modèle. Des versions du code utilisent ce modèle de programmation dans les thèses de WERNER VERDIER [7, 16] et de TÉO BOUTIN [17].

Perspectives


Les principales perspectives de cette documentation sont d'enrichir la seconde partie en ajoutant par exemple un tutoriel sur le modèle de « Shan & Chen ». On rappelle que cette approche diphasique spécifique a très largement contribué au succès de la LBM. L'approche considère le fluide comme un « fluide de Van Der Waals » qui permet de simuler un changement de phase liquide/gaz avec une loi d'état de type « Van Der Waals » (ou « Peng-Robinson » ou encore « Carnahan-Starling »). Un second tutoriel est aussi en préparation qui précise la façon de remplacer l'étape classique de déplacement par un schéma de Lax-Wendroff. Ce schéma est quelque fois utilisé dans des problèmes d'« AMR » (Adaptative Mesh Refinement) pour la gestion des tailles de mailles différentes dans le domaine de calcul. Enfin des tests de **LBM_saclay** sont prévus sur les derniers super-calculateurs, en particulier **Adastra** au CINES.

		Note Technique DES	Page 102/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

		Note Technique DES	Page 103/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Quatrième partie

Annexes

		Note Technique DES	Page 105/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Annexe A

Développements de Chapman-Enskog pour les équations de Navier-Stokes

A.1 Introduction

Cette note détaille les calculs intermédiaires du développement de Chapman-Enskog de l'équation cinétique de Boltzmann sur réseau pour la simulation des équations de Navier-Stokes. Les développements sont inspirés de [22, pp 332–336] avec la méthode de Chapman-Enskog. Une méthode alternative mais équivalente aux développements de C-E peuvent être trouvés dans [23]. Dans cette première note, on suppose que la fonction de distribution à l'équilibre et le réseau associé sont connus pour se concentrer sur la conduite des calculs et les approximations. La prochaine note décrira la méthodologie pour choisir cette fonction de distribution et son réseau pour résoudre une équation de transport par advection-diffusion et l'équation de Cahn-Hilliard.

A.1.1 Rappels de la méthode

La méthode de Boltzmann sur réseau consiste à résoudre l'équation cinétique suivante :

$$f_i(\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t) = f_i(\mathbf{x}, t) - \frac{1}{\lambda} \left[f_i(\mathbf{x}, t) - f_i^{(0)}(\mathbf{x}, t) \right] \quad (\text{A.1})$$


où $\mathbf{x} = (x, y, z)^T$, Δx est le pas d'espace, Δt est le pas de temps et λ est un coefficient de relaxation de la fonction de distribution $f_i(\mathbf{x}, t)$ vers une fonction à l'équilibre $f_i^{(0)}(\mathbf{x}, t)$. Ces deux dernières fonctions sont dépendantes de la vitesse de déplacement \mathbf{e}_i qui est représentée par l'indice $i = 1, \dots, N$. La fonction de distribution à l'équilibre est choisie de la forme :

$$f_i^{(0)}(\mathbf{x}, t) = \rho(\mathbf{x}, t) w_i \left[1 + 3 \frac{(\mathbf{e}_i \cdot \mathbf{u}(\mathbf{x}, t))}{c^2} + \frac{9}{2} \frac{(\mathbf{e}_i \cdot \mathbf{u}(\mathbf{x}, t))^2}{c^4} - \frac{3}{2} \frac{\mathbf{u}^2(\mathbf{x}, t)}{c^2} \right] \quad (\text{A.2})$$

où $\rho(\mathbf{x}, t)$ est la densité, $\mathbf{u} = (u_x, u_y, u_z)^T$ est le vecteur vitesse, $c = \frac{\Delta x}{\Delta t}$ la vitesse du réseau et les w_i sont des poids constants qui dépendent du réseau choisi.

Dans la suite de cette note, afin d'alléger les notations, on supprimera les dépendances en \mathbf{x} et t des fonctions de distributions f_i et $f_i^{(0)}$ et des variables macroscopiques ρ et \mathbf{u} . On choisit également le réseau en deux dimensions d'espace et neuf directions de déplacement (réseau D2Q9). Pour ce réseau $i = 0, \dots, 8$ et :

$$\mathbf{e}_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad (\text{A.3a})$$

		Note Technique DES	Page 106/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

$$\mathbf{e}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} c, \mathbf{e}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} c, \mathbf{e}_3 = \begin{pmatrix} -1 \\ 0 \end{pmatrix} c, \mathbf{e}_4 = \begin{pmatrix} 0 \\ -1 \end{pmatrix} c \quad (\text{A.3b})$$

$$\mathbf{e}_5 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} c, \mathbf{e}_6 = \begin{pmatrix} -1 \\ 1 \end{pmatrix} c, \mathbf{e}_7 = \begin{pmatrix} -1 \\ -1 \end{pmatrix} c, \mathbf{e}_8 = \begin{pmatrix} 1 \\ -1 \end{pmatrix} c \quad (\text{A.3c})$$

Les poids w_i prennent les valeurs :

$$w_0 = \frac{4}{9}, \quad w_{1,\dots,4} = \frac{1}{9}, \quad w_{5,\dots,8} = \frac{1}{36} \quad (\text{A.4})$$

A.1.2 Objectifs

L'objectif de cette note est de démontrer à l'aide du développement de Chapman-Enskog que l'équation (A.1) associée avec la fonction de distribution à l'équilibre (A.2) et du réseau D2Q9 (relations (A.3a)-(A.3c) et (A.4)) permet de résoudre le système d'équations de Navier-Stokes quasi-incompressible suivant :

$$\partial_t \rho + \partial_\alpha (\rho u_\alpha) = 0 \quad (\text{A.5})$$

$$\partial_t (\rho u_\alpha) + \partial_\beta (\rho u_\alpha u_\beta) = -\partial_\alpha p + \partial_\beta [\nu \rho (\partial_\beta u_\alpha + \partial_\alpha u_\beta)] + \mathcal{O}(\rho u^3) \quad (\text{A.6})$$

L'équation (A.5) est l'équation de conservation de la masse et (A.6) est l'équation de conservation de la quantité de mouvement. Dans ces équations, les indices grecs valent en deux dimensions $\alpha = x, y$ et $\beta = x, y$ respectivement et la convention de sommation d'Einstein sur les indices grecs répétés est utilisée (par exemple $\partial_\alpha u_\alpha = \partial_x u_x + \partial_y u_y$). Le symbole ∂_α est une notation condensée pour exprimer la dérivée partielle en espace $\frac{\partial}{\partial \alpha}$. Par exemple $\partial_x \equiv \frac{\partial}{\partial x}$, $\partial_y \equiv \frac{\partial}{\partial y}$ et $\partial_t \equiv \frac{\partial}{\partial t}$. Dans l'équation (A.6) p est la pression, ν est la viscosité cinématique.

Dans la suite de cette note, la section A.2 rappelle quelques résultats préliminaires sur les moments d'ordre 0, 1, 2 et 3 de la fonction de distribution à l'équilibre (A.2). Ces résultats seront utilisés dans les calculs de la section A.3 suivante consacrée au développement de Chapman-Enskog. La section A.4 sera consacrée à une brève discussion de ce développement.


Les calculs intermédiaires du développement de Chapman-Enskog sont rarement détaillés dans les publications soit à cause du format de la publication (lettre) ou bien parce qu'elles se concentrent sur les aspects novateurs. Les calculs sont davantage détaillés dans des thèses.

Signalons que dans cette annexe, on suppose que la fonction de distribution à l'équilibre $f_i^{(0)}$ ainsi qu'une géométrie de réseau sont connues. En pratique le réseau et $f_i^{(0)}$ sont construits de façon à ce que les moments de $f_i^{(0)}$ soient égaux aux quantités physiques de l'EDP macroscopique. Cette approche est décrite dans l'annexe de la référence [21] pour des EDPs de type transport.

A.2 Grandeurs conservées et résultats préliminaires sur la fonction à l'équilibre

Les grandeurs macroscopiques conservées sont la densité et la quantité de mouvement :

$$\begin{cases} \sum_i f_i &= \rho \\ \sum_i f_i \mathbf{e}_i &= \rho \mathbf{u} \end{cases} \quad (\text{A.7})$$

		Note Technique DES	Page 107/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

qui doivent être conservées au cours d'une collision et impliquent :

$$\begin{cases} \sum_i f_i^{(0)} &= \rho \\ \sum_i f_i^{(0)} \mathbf{e}_i &= \rho \mathbf{u} \end{cases} \quad (\text{A.8})$$

Dans la suite de cette note, afin d'alléger les notations, on prendra $\Delta x = \Delta t = 1$ étant entendu que dans le cas général Δx et Δt peuvent prendre des valeurs différentes de l'unité. On a donc dans l'équation (A.2) $c = 1$. De même, conformément à la littérature sur le sujet on posera :

$$c_s^2 = \frac{1}{3} \quad (\text{A.9})$$

En utilisant les notations avec les indices grecs, la fonction de distribution à l'équilibre s'écrit :

$$f_i^{(0)} = \rho w_i \left[1 + 3e_{i\gamma} u_\gamma + \frac{9}{2} e_{i\gamma} e_{i\theta} u_\gamma u_\theta - \frac{3}{2} u_\gamma u_\theta \delta_{\gamma\theta} \right] \quad (\text{A.10})$$

Sur le réseau D2Q9, on peut montrer que les moments d'ordre 0, 1, 2 et 3 de la fonction $f_i^{(0)}$ valent respectivement :

$$\sum_i f_i^{(0)} = \rho \quad (\text{A.11a})$$

$$\sum_i f_i^{(0)} e_{i\alpha} = \rho u_\alpha \quad (\text{A.11b})$$

$$\sum_i f_i^{(0)} e_{i\alpha} e_{i\beta} = c_s^2 \rho \delta_{\alpha\beta} + \rho u_\alpha u_\beta \quad (\text{A.11c})$$

$$\sum_i f_i^{(0)} e_{i\alpha} e_{i\beta} e_{i\gamma} = c_s^2 \rho (u_\alpha \delta_{\beta\gamma} + u_\beta \delta_{\alpha\gamma} + u_\gamma \delta_{\alpha\beta}) \quad (\text{A.11d})$$

Remarque : dans le cas général où on cherche à résoudre une équation aux dérivées partielles macroscopique donnée, on cherche la fonction de distribution à l'équilibre $f_i^{(0)}$ et le nombre de directions de déplacement du réseau de manière à ce que ses moments d'ordre 0 et éventuellement 1 soient égaux aux quantités conservées dans l'EDP et on détermine les moments d'ordre supérieurs de telle façon à reproduire l'équation macroscopique.

A.3 Développement de Chapman-Enskog

A.3.1 Première partie : développement de Taylor et séparation des échelles


Les développements présentés dans cette première partie sont classiques et peuvent être trouvés dans beaucoup de références. Dans cette note les développements sont inspirés de [22]. On pourra se référer à cette référence pour davantage de précisions, en particulier sur le paramètre ε qui est interprété comme le nombre de Knudsen.

En posant $\Delta x = \Delta t = 1$ et en utilisant les indices grecs, l'équation de Boltzmann sur réseau devient :

$$f_i(x + e_{i\beta}, t + 1) = f_i(x, t) - \frac{1}{\lambda} \left[f_i(x, t) - f_i^{(0)}(x, t) \right] \quad (\text{A.12})$$

Le développement en série de Taylor du membre de gauche de l'équation (A.12) au second ordre en espace et au second ordre en temps donne :

$$f_i(x + e_{i\beta}, t + 1) \simeq f_i + e_{i\beta} \partial_\beta f_i + \frac{1}{2} e_{i\alpha} e_{i\beta} \partial_\alpha \partial_\beta f_i + \partial_t f_i + \frac{1}{2} \partial_t^2 f_i + e_{i\beta} \partial_\beta \partial_t f_i$$

		Note Technique DES	Page 108/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

En remplaçant dans (A.12), on obtient :

$$e_{i\beta}\partial_\beta f_i + \frac{1}{2}e_{i\alpha}e_{i\beta}\partial_\alpha\partial_\beta f_i + \partial_t f_i + \frac{1}{2}\partial_t^2 f_i + e_{i\beta}\partial_\beta\partial_t f_i = -\frac{1}{\lambda} [f_i - f_i^{(0)}] \quad (\text{A.13})$$

On effectue une séparation des échelles en espace ($x_1 = \varepsilon x$) et en temps en distinguant un temps caractéristique de convection $t_0 = \varepsilon t$ et un temps caractéristique de diffusion $t_1 = \varepsilon^2 t$ de telle sorte que :

$$\begin{cases} t &= \frac{1}{\varepsilon}t_0 + \frac{1}{\varepsilon^2}t_1 \\ x &= \frac{1}{\varepsilon}x_1 \end{cases} \quad (\text{A.14})$$

Ce qui conduit aux dérivées partielles suivantes :

$$\begin{cases} \partial_t &= \varepsilon\partial_{t_0} + \varepsilon^2\partial_{t_1} \\ \partial_\alpha &= \varepsilon\partial_\alpha^{(1)} \end{cases} \quad (\text{A.15})$$

La fonction de distribution est développée jusqu'au second ordre :

$$f_i \simeq f_i^{(0)} + \varepsilon f_i^{(1)} + \varepsilon^2 f_i^{(2)} \quad (\text{A.16})$$

Ce développement et les équations (A.7) et (A.8) impliquent que :

$$\sum_i f_i^{(1)} = \sum_i f_i^{(2)} = 0 \quad (\text{A.17a})$$

$$\sum_i f_i^{(1)} e_{i\alpha} = \sum_i f_i^{(2)} e_{i\alpha} = 0 \quad (\text{A.17b})$$

Remarque : l'équation (A.17b) est déduite car la quantité de mouvement est une grandeur locale conservée pour les équations de Navier-Stokes. Dans le cas du transport par advection-diffusion, ce n'est plus le cas et il faudra calculer le moment d'ordre 1 de la fonction $f_i^{(1)}$ (voir note suivante).

A.3.1.1 Moments des termes en ε

On remplace les dérivées partielles en espace et en temps dans (A.13) en utilisant les relations (A.15) et on utilise également le développement de la fonction de distribution en utilisant (A.16). On regroupe ensuite les termes en ε et ε^2 .

Le regroupement des termes du premier ordre en ε donne l'égalité suivante :

$$\partial_\alpha^{(1)} e_{i\alpha} f_i^{(0)} + \partial_{t_0} f_i^{(0)} = -\frac{1}{\lambda} f_i^{(1)} \quad (\text{A.18})$$


En utilisant les relations préliminaires (A.11a) et (A.11b), le calcul du moment d'ordre 0 (somme sur i) de cette équation on obtient :

$$\partial_{t_0} \rho + \partial_\alpha^{(1)} (\rho u_\alpha) = 0 \quad (\text{A.19})$$

De même, en utilisant les relations préliminaires (A.11b) et (A.11c), le calcul du moment d'ordre 1 (en multipliant l'équation (A.18) par $e_{i\beta}$ puis en faisant la somme sur i) conduit à l'équation suivante :

$$\partial_{t_0} (\rho u_\alpha) + \partial_\beta^{(1)} (c_s^2 \rho \delta_{\alpha\beta} + \rho u_\alpha u_\beta) = 0 \quad (\text{A.20})$$

Ces deux moments d'ordre 0 et 1 des termes en ε (équations (A.19) et (A.20)) seront très utilisés pour le calcul du second moment de la fonction $f_i^{(1)}$ en deuxième partie de ce développement.

		Note Technique DES	Page 109/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

A.3.1.2 Moments des termes en ε^2

Le regroupement des termes en ε^2 conduit à la relation :

$$e_{i\beta} \partial_\beta^{(1)} f_i^{(1)} + \frac{1}{2} e_{i\alpha} e_{i\beta} \partial_\alpha^{(1)} \partial_\beta^{(1)} f_i^{(0)} + \partial_{t_0} f_i^{(1)} + \partial_{t_1} f_i^{(0)} + \frac{1}{2} \partial_{t_0}^2 f_i^{(0)} + e_{i\beta} \partial_\beta^{(1)} \partial_{t_0} f_i^{(0)} = -\frac{1}{\lambda} f_i^{(2)} \quad (\text{A.21})$$

En appliquant séparément $\frac{1}{2} e_{i\beta} \partial_\beta$ et $\frac{1}{2} \partial_{t_0}$ sur l'équation (A.18) et en effectuant la somme des deux résultats, cette équation (A.21) se simplifie en :

$$\partial_{t_1} f_i^{(0)} + \left(1 - \frac{1}{2\lambda}\right) \left[\partial_\beta^{(1)} e_{i\beta} f_i^{(1)} + \partial_{t_0} f_i^{(1)} \right] = -\frac{1}{\lambda} f_i^{(2)}$$

Grâce aux équations (A.11a), (A.17a) et (A.17b), le moment d'ordre 0 de cette équation donne :

$$\partial_{t_1} \rho = 0 \quad (\text{A.22})$$

et son moment d'ordre 1 :

$$\partial_{t_1} (\rho u_\alpha) + \left(1 - \frac{1}{2\lambda}\right) \left[\partial_\beta^{(1)} \sum_i f_i^{(1)} e_{i\alpha} e_{i\beta} \right] = 0 \quad (\text{A.23})$$

A.3.1.3 Regroupement des moments

On regroupe maintenant les moments d'ordre 0 et 1 calculés séparément pour les termes en ε et les termes en ε^2 . En effectuant $\varepsilon \times \text{Eq (A.19)}$ + $\varepsilon^2 \times \text{Eq (A.22)}$ le moment d'ordre 0 donne :

$$\partial_t \rho + \partial_\alpha (\rho u_\alpha) = 0 \quad (\text{A.24})$$

qui correspond à la première équation du système recherché, c'est à dire l'équation de conservation de la masse. De même, en effectuant $\varepsilon \times \text{Eq (A.20)}$ + $\varepsilon^2 \times \text{Eq (A.23)}$ le moment d'ordre 1 donne :

$$\partial_t (\rho u_\alpha) + \partial_\beta (\Pi_{\alpha\beta}^{(0)} + \Pi_{\alpha\beta}^{(1)}) = 0 \quad (\text{A.25})$$

où on a posé :

$$\Pi_{\alpha\beta}^{(0)} = \sum_i f_i^{(0)} e_{i\alpha} e_{i\beta} \quad (\text{A.26a})$$


$$\Pi_{\alpha\beta}^{(1)} = \left(1 - \frac{1}{2\lambda}\right) \varepsilon \sum_i f_i^{(1)} e_{i\alpha} e_{i\beta} \quad (\text{A.26b})$$

Les notations $\Pi_{\alpha\beta}^{(0)}$ et $\Pi_{\alpha\beta}^{(1)}$ sont les moments d'ordre 2 des fonctions $f_i^{(0)}$ et $f_i^{(1)}$ respectivement. $\Pi_{\alpha\beta}^{(0)}$ est connue grâce à la définition de la fonction de distribution à l'équilibre et au calcul préliminaire (A.11c). Il s'agit donc de calculer le moment d'ordre 2 de la fonction $f_i^{(1)}$ qui constitue la seconde partie de la démonstration.

A.3.2 Seconde partie : coefficient de viscosité et équation de conservation de la quantité de mouvement

A.3.2.1 Viscosité

Pour calculer le second moment de la fonction $f_i^{(1)}$ ($\sum_i f_i^{(1)} e_{i\alpha} e_{i\beta}$) on utilise les résultats préliminaires (A.11c) et (A.11d) pour les moments d'ordre 2 et d'ordre 3 de la fonction de distribution à l'équilibre $f_i^{(0)}$:

		Note Technique DES	Page 110/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

$$\begin{aligned}
\sum_i f_i^{(1)} e_{i\alpha} e_{i\beta} &= -\lambda \left[\partial_\gamma^{(1)} \left(\sum_i f_i^{(0)} e_{i\alpha} e_{i\beta} e_{i\gamma} \right) + \partial_{t_0} \left(\sum_i f_i^{(0)} e_{i\alpha} e_{i\beta} \right) \right] \\
&= -\lambda \left[\partial_\gamma^{(1)} c_s^2 \rho (u_\alpha \delta_{\beta\gamma} + u_\beta \delta_{\alpha\gamma} + u_\gamma \delta_{\alpha\beta}) + \partial_{t_0} (c_s^2 \rho \delta_{\alpha\beta} + \rho u_\alpha u_\beta) \right] \\
&= -\lambda c_s^2 \left[\partial_\beta^{(1)} (\rho u_\alpha) + \partial_\alpha^{(1)} (\rho u_\beta) + \partial_\gamma^{(1)} (\rho u_\gamma) \delta_{\alpha\beta} \right] - \lambda \partial_{t_0} (c_s^2 \rho \delta_{\alpha\beta}) - \lambda \partial_{t_0} (\rho u_\alpha u_\beta)
\end{aligned}$$

Pour les deux derniers termes, les dérivées partielles en temps sont converties en dérivées partielles en espace en utilisant les moments d'ordre 0 et d'ordre 1 de l'équation d'ordre ε (équations (A.19) et (A.20) respectivement). Pour l'avant dernier terme, on obtient :

$$-\lambda \partial_{t_0} (c_s^2 \rho \delta_{\alpha\beta}) = +\lambda c_s^2 \partial_\gamma^{(1)} (\rho u_\gamma) \delta_{\alpha\beta}$$

Ce terme s'annule avec celui contenu dans le crochet qui est de signe négatif.

Pour le dernier terme, on utilise le résultat suivant (voir annexe A pour la démonstration) :

$$\partial_{t_0} (\rho u_\alpha u_\beta) = -\partial_\gamma^{(1)} (\rho u_\alpha u_\beta u_\gamma) - c_s^2 (u_\alpha \partial_\beta^{(1)} \rho + u_\beta \partial_\alpha^{(1)} \rho) \quad (\text{A.27})$$

En développant les dérivées partielles contenues dans le crochet, on obtient finalement :

$$\begin{aligned}
\sum_i f_i^{(1)} e_{i\alpha} e_{i\beta} &= -\lambda c_s^2 \left[\partial_\beta^{(1)} (\rho u_\alpha) + \partial_\alpha^{(1)} (\rho u_\beta) \right] + \lambda \partial_\gamma^{(1)} (\rho u_\alpha u_\beta u_\gamma) + \lambda c_s^2 (u_\alpha \partial_\beta^{(1)} \rho + u_\beta \partial_\alpha^{(1)} \rho) \\
&= -\lambda c_s^2 \left[\rho \partial_\beta^{(1)} u_\alpha + \rho \partial_\alpha^{(1)} u_\beta \right] + \lambda \partial_\gamma^{(1)} (\rho u_\alpha u_\beta u_\gamma)
\end{aligned}$$

Le terme $\Pi_{\alpha\beta}^{(1)}$ devient :

$$\begin{aligned}
\Pi_{\alpha\beta}^{(1)} &= \left(1 - \frac{1}{2\lambda} \right) \varepsilon \sum_i f_i^{(1)} e_{i\alpha} e_{i\beta} \\
&= \left(1 - \frac{1}{2\lambda} \right) \varepsilon \left\{ -\lambda c_s^2 \left[\rho \partial_\beta^{(1)} u_\alpha + \rho \partial_\alpha^{(1)} u_\beta \right] + \lambda \partial_\gamma^{(1)} (\rho u_\alpha u_\beta u_\gamma) \right\} \\
&= c_s^2 \left(\lambda - \frac{1}{2} \right) \left\{ -\left[\rho \partial_\beta u_\alpha + \rho \partial_\alpha u_\beta \right] + \frac{1}{c_s^2} \partial_\gamma (\rho u_\alpha u_\beta u_\gamma) \right\}
\end{aligned}$$

En posant :

$$\nu = c_s^2 \left(\lambda - \frac{1}{2} \right) \quad (\text{A.28})$$


on obtient :

$$\Pi_{\alpha\beta}^{(1)} = -\nu \left[\rho \partial_\beta u_\alpha + \rho \partial_\alpha u_\beta \right] + \frac{1}{c_s^2} \nu \partial_\gamma (\rho u_\alpha u_\beta u_\gamma) \quad (\text{A.29})$$

L'équation (A.28) donne la relation entre le paramètre de relaxation λ de l'équation cinétique de Boltzmann et la viscosité cinématique ν du fluide.

Remarque : en toute rigueur, lors du développement de Taylor, il est nécessaire de considérer $\Delta x \neq 1$ et $\Delta t \neq 1$ et on voit apparaître le rapport $\Delta x^2 / \Delta t$ en facteur devant le terme entre parenthèses :

$$\nu = c_s^2 \left(\lambda - \frac{1}{2} \right) \frac{\Delta x^2}{\Delta t}$$

		Note Technique DES	Page 111/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

A.3.2.2 Equation de conservation de la quantité de mouvement

En remplaçant (A.11c) et (A.29) dans (A.25), on obtient :

$$\partial_t(\rho u_\alpha) + \partial_\beta [c_s^2 \rho \delta_{\alpha\beta} + \rho u_\alpha u_\beta - \nu(\rho \partial_\beta u_\alpha + \rho \partial_\alpha u_\beta)] + \frac{1}{c_s^2} \partial_\beta \nu \partial_\gamma (\rho u_\alpha u_\beta u_\gamma) = 0$$

$$\partial_t(\rho u_\alpha) + \partial_\beta (\rho u_\alpha u_\beta) = -\partial_\alpha (c_s^2 \rho) + \partial_\beta [\nu(\rho \partial_\beta u_\alpha + \rho \partial_\alpha u_\beta)] + \mathcal{O}(\rho u^3)$$

où on a considéré $-\frac{1}{c_s^2} \partial_\beta \nu \partial_\gamma (\rho u_\alpha u_\beta u_\gamma) = \mathcal{O}(\rho u^3)$. Finalement, en posant $p = c_s^2 \rho$, on obtient l'équation de conservation de la quantité de mouvement :

$$\partial_t(\rho u_\alpha) + \partial_\beta (\rho u_\alpha u_\beta) = -\partial_\alpha p + \partial_\beta [\nu \rho (\partial_\beta u_\alpha + \partial_\alpha u_\beta)] + \mathcal{O}(\rho u^3)$$

qui correspond à la seconde équation recherchée du système d'équations de Navier-Stokes.

A.4 Récapitulatif

En effectuant le développement de Chapman-Enskog (ordre deux en espace, ordre deux en temps et en ε^2) et en calculant les moments d'ordre 0 et 1 des équations en ε et ε^2 , on a montré que les équations (A.1) et (A.2) permettent de résoudre sur un réseau D2Q9 les équations de Navier-Stokes :

$$\partial_t \rho + \partial_\alpha (\rho u_\alpha) = 0 \quad (\text{A.30})$$

$$\partial_t (\rho u_\alpha) + \partial_\beta (\rho u_\alpha u_\beta) = -\partial_\alpha p + \partial_\beta [\nu \rho (\partial_\beta u_\alpha + \partial_\alpha u_\beta)] + \mathcal{O}(u^3) \quad (\text{A.31})$$

avec la pression liée à la densité par la relation $p = c_s^2 \rho$ et une viscosité cinématique ν donnée par :

$$\nu = c_s^2 \left(\lambda - \frac{1}{2} \right) \frac{\Delta x^2}{\Delta t} \quad (\text{A.32})$$

Dans l'équation (A.31), le terme en $\mathcal{O}(u^3)$ est de la forme $-\frac{1}{c_s^2} \partial_\beta \nu \partial_\gamma (\rho u_\alpha u_\beta u_\gamma)$ qui est négligeable pour les faibles nombres de Mach. Lorsque ce dernier devient important, il est nécessaire de corriger la fonction de distribution à l'équilibre pour éliminer/diminuer l'influence de ce terme. On pourra se référer à [9] qui analyse plusieurs $f_i^{(0)}$ ou termes forces en ce sens.

A.5 Démonstration de l'égalité (A.27)

On démontre dans cette annexe la relation :

$$\partial_{t_0} (\rho u_\alpha u_\beta) = -\partial_\gamma^{(1)} (\rho u_\alpha u_\beta u_\gamma) - c_s^2 (u_\alpha \partial_\beta^{(1)} \rho + u_\beta \partial_\alpha^{(1)} \rho)$$


Le développement du membre de gauche donne :

$$\partial_{t_0} (\rho u_\alpha u_\beta) = u_\alpha u_\beta \partial_{t_0} \rho + \rho u_\alpha \partial_{t_0} u_\beta + \rho u_\beta \partial_{t_0} u_\alpha$$

En remarquant que $\rho u_\alpha \partial_{t_0} u_\beta = u_\alpha \partial_{t_0} (\rho u_\beta) - u_\alpha u_\beta \partial_{t_0} \rho$ et que $\rho u_\beta \partial_{t_0} u_\alpha = u_\beta \partial_{t_0} (\rho u_\alpha) - u_\alpha u_\beta \partial_{t_0} \rho$, on obtient :

$$\partial_{t_0} (\rho u_\alpha u_\beta) = -u_\alpha u_\beta \partial_{t_0} \rho + u_\alpha \partial_{t_0} (\rho u_\beta) + u_\beta \partial_{t_0} (\rho u_\alpha)$$

Les dérivées partielles en temps sont converties en dérivées partielles en espace à l'aide des équations (A.19) pour le premier terme et (A.20) pour les deux derniers :

		Note Technique DES	Page 112/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

$$\begin{aligned}
\partial_{t_0}(\rho u_\alpha u_\beta) &= u_\alpha u_\beta \partial_\gamma^{(1)}(\rho u_\gamma) - u_\alpha \left[\partial_\gamma^{(1)}(c_s^2 \rho \delta_{\beta\gamma}) + \partial_\gamma^{(1)}(\rho u_\beta u_\gamma) \right] - u_\beta \left[\partial_\gamma^{(1)}(c_s^2 \rho \delta_{\alpha\gamma}) + \partial_\gamma^{(1)}(\rho u_\alpha u_\gamma) \right] \\
&= u_\alpha u_\beta \partial_\gamma^{(1)}(\rho u_\gamma) - u_\alpha \left[\partial_\beta^{(1)}(c_s^2 \rho) + \partial_\gamma^{(1)}(\rho u_\beta u_\gamma) \right] - u_\beta \left[\partial_\alpha^{(1)}(c_s^2 \rho) + \partial_\gamma^{(1)}(\rho u_\alpha u_\gamma) \right] \\
&= - \left[u_\alpha \partial_\gamma^{(1)}(\rho u_\beta u_\gamma) + u_\beta \partial_\gamma^{(1)}(\rho u_\alpha u_\gamma) - u_\alpha u_\beta \partial_\gamma^{(1)}(\rho u_\gamma) \right] - c_s^2 \left[u_\alpha \partial_\beta^{(1)} \rho + u_\beta \partial_\alpha^{(1)} \rho \right] \quad (\text{A.33})
\end{aligned}$$

Les trois premiers termes contenus dans le premier crochet peuvent être exprimés sous la forme condensée $\partial_\gamma^{(1)}(\rho u_\alpha u_\beta u_\gamma)$. Il existe plusieurs façons de la démontrer, on présente dans cette annexe une façon de procéder.

Un premier développement donne $\partial_\gamma^{(1)}(\rho u_\alpha u_\beta u_\gamma) = u_\alpha \partial_\gamma^{(1)}(\rho u_\beta u_\gamma) + \rho u_\beta u_\gamma (\partial_\gamma^{(1)} u_\alpha)$ c'est-à-dire :

$$u_\alpha \partial_\gamma^{(1)}(\rho u_\beta u_\gamma) = \partial_\gamma^{(1)}(\rho u_\alpha u_\beta u_\gamma) - \rho u_\beta u_\gamma (\partial_\gamma^{(1)} u_\alpha) \quad (\text{A.34})$$

Un second développement conduit à $\partial_\gamma^{(1)}(\rho u_\alpha u_\beta u_\gamma) = u_\beta \partial_\gamma^{(1)}(\rho u_\alpha u_\gamma) + \rho u_\alpha u_\gamma (\partial_\gamma^{(1)} u_\beta)$ c'est-à-dire :

$$u_\beta \partial_\gamma^{(1)}(\rho u_\alpha u_\gamma) = \partial_\gamma^{(1)}(\rho u_\alpha u_\beta u_\gamma) - \rho u_\alpha u_\gamma (\partial_\gamma^{(1)} u_\beta) \quad (\text{A.35})$$

Enfin, un dernier développement donne $\partial_\gamma^{(1)}(\rho u_\alpha u_\beta u_\gamma) = u_\alpha u_\beta \partial_\gamma^{(1)}(\rho u_\gamma) + \rho u_\gamma \partial_\gamma^{(1)}(u_\alpha u_\beta)$ c'est-à-dire :

$$-u_\alpha u_\beta \partial_\gamma^{(1)}(\rho u_\gamma) = -\partial_\gamma^{(1)}(\rho u_\alpha u_\beta u_\gamma) + \rho u_\gamma \partial_\gamma^{(1)}(u_\alpha u_\beta) \quad (\text{A.36})$$

En effectuant la somme (A.34), (A.35) et (A.36), on obtient :


$$u_\alpha \partial_\gamma^{(1)}(\rho u_\beta u_\gamma) + u_\beta \partial_\gamma^{(1)}(\rho u_\alpha u_\gamma) - u_\alpha u_\beta \partial_\gamma^{(1)}(\rho u_\gamma) = \partial_\gamma^{(1)}(\rho u_\alpha u_\beta u_\gamma) - \rho u_\beta u_\gamma (\partial_\gamma^{(1)} u_\alpha) - \rho u_\alpha u_\gamma (\partial_\gamma^{(1)} u_\beta) + \rho u_\gamma \partial_\gamma^{(1)}(u_\alpha u_\beta)$$

Dans l'équation ci-dessus, les trois derniers termes du membre de droite s'annulent entre eux, on a donc bien démontré que :

$$\left[u_\alpha \partial_\gamma^{(1)}(\rho u_\beta u_\gamma) + u_\beta \partial_\gamma^{(1)}(\rho u_\alpha u_\gamma) - u_\alpha u_\beta \partial_\gamma^{(1)}(\rho u_\gamma) \right] = \partial_\gamma^{(1)}(\rho u_\alpha u_\beta u_\gamma)$$

Finalement en remplaçant ce résultat dans (A.33), on obtient la relation recherchée :

$$\partial_{t_0}(\rho u_\alpha u_\beta) = -\partial_\gamma^{(1)}(\rho u_\alpha u_\beta u_\gamma) - c_s^2 \left[u_\alpha \partial_\beta^{(1)} \rho + u_\beta \partial_\alpha^{(1)} \rho \right]$$

		Note Technique DES	Page 113/116
		<u>Réf</u> : STMF/LMSF/NT/2022-70869	
		<u>Date</u> : 21/11/2022	<u>Indice</u> : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Annexe B

Changement de variable dans la LBE pour la prise en compte d'une source

L'équation de Boltzmann discrète avec un terme source (ou terme force externe) \mathcal{S}_i^ϑ peut s'écrire sous la forme suivante avec le terme de collision BGK :

$$\frac{\partial \vartheta_i}{\partial t} + \mathbf{c}_i \cdot \nabla \vartheta_i = -\frac{\vartheta_i - \vartheta_i^{eq}}{\tau_\vartheta} + \mathcal{S}_i^\vartheta. \quad (\text{B.1})$$

Dans ce qui suit, les calculs sont développés en posant $\vartheta \equiv f$, $\mathcal{S}_i^\vartheta = \mathcal{S}_i^f = \mathcal{S}_i$ et $\tau_\vartheta \equiv \tau$, mais l'approche du changement de variable est également valable pour $\vartheta \equiv h$ et $\vartheta \equiv s$. Les termes qui sont évalués en \mathbf{x} et t sont notés $f_i \equiv f_i(\mathbf{x}, t)$, $f_i^{eq} \equiv f_i^{eq}(\mathbf{x}, t)$ et $\mathcal{S}_i \equiv \mathcal{S}_i(\mathbf{x}, t)$, tandis que les termes évalués en $\mathbf{x} + \mathbf{c}_i \delta t$ et $t + \delta t$ sont notés avec une étoile : $f_i^* \equiv f_i(\mathbf{x} + \mathbf{c}_i \delta t, t + \delta t)$, $f_i^{*eq} \equiv f_i^{eq}(\mathbf{x} + \mathbf{c}_i \delta t, t + \delta t)$ et $\mathcal{S}_i^* \equiv \mathcal{S}_i(\mathbf{x} + \mathbf{c}_i \delta t, t + \delta t)$. Avec ces notations, l'intégration de l'Eq. (B.1) entre t et $t + \delta t$ donne :

$$f_i^* = f_i - \frac{\delta t}{2\tau} (f_i^* - f_i^{*eq}) - \frac{\delta t}{2\tau} (f_i - f_i^{eq}) + \frac{\delta t}{2} \mathcal{S}_i^* + \frac{\delta t}{2} \mathcal{S}_i \quad (\text{B.2})$$

où la loi des trapèzes à été appliquée pour le membre de droite de l'Eq. (B.1). Dans cette expression, pour les termes en $\mathbf{x} + \mathbf{c}_i \delta t$ et $t + \delta t$, on introduit le changement de variable suivant :

$$\bar{f}_i^* = f_i^* + \frac{\delta t}{2\tau} (f_i^* - f_i^{*eq}) - \frac{\delta t}{2} \mathcal{S}_i^*. \quad (\text{B.3})$$

Le même changement de variable est utilisé pour les termes en \mathbf{x} et t :

$$\bar{f}_i = f_i + \frac{\delta t}{2\tau} (f_i - f_i^{eq}) - \frac{\delta t}{2} \mathcal{S}_i. \quad (\text{B.4})$$

En inversant cette dernière relation pour exprimer f_i en fonction de \bar{f}_i , on obtient :

$$f_i = \frac{2\tau}{2\tau + \delta t} \left(\bar{f}_i + \frac{\delta t}{2\tau} f_i^{eq} + \frac{\delta t}{2} \mathcal{S}_i \right). \quad (\text{B.5})$$

Avec les Eqs. (B.3) et (B.5), l'Eq. (B.2) devient

$$\bar{f}_i^* = \bar{f}_i - \frac{\delta t}{\tau + \delta t/2} \left(\bar{f}_i - f_i^{eq} + \frac{\delta t}{2} \mathcal{S}_i \right) + \delta t \mathcal{S}_i \quad (\text{B.6})$$

Si on définit un nouveau changement de variable

$$\bar{f}_i^{eq} = f_i^{eq} - \frac{\delta t}{2} \mathcal{S}_i, \quad (\text{B.7})$$

alors l'Eq. (B.6) est équivalente à

$$\bar{f}_i^* = \bar{f}_i - \frac{\delta t}{\tau + \delta t/2} (\bar{f}_i - \bar{f}_i^{eq}) + \delta t \mathcal{S}_i. \quad (\text{B.8})$$


Sans le changement de variable sur f_i^{eq} , l'Eq. (B.6) est équivalente à

$$\bar{f}_i^* = \bar{f}_i - \frac{\delta t}{\tau + \delta t/2} (\bar{f}_i - f_i^{eq}) + \frac{\tau \delta t}{\tau + \delta t/2} \mathcal{S}_i, \quad (\text{B.9})$$

où seul le facteur devant le terme source est modifié.

En introduisant le taux de collision sans dimension défini par $\bar{\tau} = \tau/\delta t$, l'Eq. (B.8) s'écrit finalement

$$\bar{f}_i^* = \bar{f}_i - \frac{1}{\bar{\tau} + 1/2} (\bar{f}_i - \bar{f}_i^{eq}) + \delta t \mathcal{S}_i, \quad (\text{B.10})$$

		Note Technique DES	Page 114/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

ou alternativement,

$$\bar{f}_i^* = \bar{f}_i - \frac{1}{\bar{\tau} + 1/2} (\bar{f}_i - f_i^{eq}) + \frac{\bar{\tau}\delta t}{\bar{\tau} + 1/2} \mathcal{S}_i. \quad (\text{B.11})$$

Dans les chapitres de cette documentation, l'Eq. (B.10)

est l'équation d'évolution utilisée dans les algorithmes LBM. Le changement de variable (Eq. (B.4)) conduit à calculer les moments d'ordre zéro par :

$$\mathcal{M}_0 = \sum_i \bar{f}_i + \frac{\delta t}{2} \sum_i \mathcal{S}_i \quad (\text{B.12})$$

B.1 Equivalence de l'Eq. 6.25 de [1] et de l'Eq. B.11

B.1.1 Démonstrations de l'équivalence

Dans Krüger *et al.* [1, p. 239], l'équation d'évolution pour \bar{f}_i s'écrit (Eq. 6.25) :

$$\bar{f}_i^* = \bar{f}_i - \frac{\delta t}{\lambda} (\bar{f}_i - f_i^{eq}) + \left(1 - \frac{\delta t}{2\lambda}\right) F_i \delta t \quad (\text{B.13})$$

où $\bar{f}_i^* \equiv \bar{f}_i(\mathbf{x} + \mathbf{c}_i \delta t, t + \delta t)$. On note que le temps de relaxation λ est homogène à un temps et il est défini sous 6.25 par

$$\lambda = \tau + \frac{\delta t}{2} \quad (\text{B.14})$$

On définit le taux de relaxation (sans dimension) par :

$$\bar{\tau} = \frac{\tau}{\delta t} \quad (\text{B.15})$$

En l'utilisant dans (B.14), on obtient :

$$\begin{aligned} \lambda &= \bar{\tau}\delta t + \frac{\delta t}{2} \\ &= (\bar{\tau} + 1/2) \delta t \end{aligned} \quad (\text{B.16})$$

En remplaçant dans Eq. (B.13), on obtient :

$$\bar{f}_i^* = \bar{f}_i - \frac{1}{\bar{\tau} + 1/2} (\bar{f}_i - f_i^{eq}) + \left(1 - \frac{\delta t}{2(\bar{\tau} + 1/2)\delta t}\right) F_i \delta t$$

En simplifiant le facteur devant F_i , on obtient :

$$\left(1 - \frac{\delta t}{2(\bar{\tau} + 1/2)\delta t}\right) = \frac{2\bar{\tau}}{2\bar{\tau} + 1} = \frac{\bar{\tau}}{\bar{\tau} + 1/2}$$

Finalement, on obtient :

$$\bar{f}_i^* = \bar{f}_i - \frac{1}{\bar{\tau} + 1/2} (\bar{f}_i - f_i^{eq}) + \frac{\bar{\tau}}{\bar{\tau} + 1/2} F_i \delta t \quad (\text{B.17})$$

qui correspond à l'Eq. (B.11).

B.1.2 Lien avec la formulation qui introduit \bar{f}_i^{eq}

L'Eq. (B.13) avec λ défini par l'Eq. (B.14) s'écrit :

$$\bar{f}_i^* = \bar{f}_i - \frac{\delta t}{\tau + \frac{\delta t}{2}} (\bar{f}_i - f_i^{eq}) + \left(1 - \frac{\delta t}{2\tau + \delta t}\right) F_i \delta t$$

En manipulant le dernier terme du membre de droite, on obtient :

$$\begin{aligned} \bar{f}_i^* &= \bar{f}_i - \frac{\delta t}{\tau + \frac{\delta t}{2}} (\bar{f}_i - f_i^{eq}) - \frac{\delta t}{2(\tau + \delta t/2)} F_i \delta t + F_i \delta t \\ &= \bar{f}_i - \frac{\delta t}{\tau + \frac{\delta t}{2}} (\bar{f}_i - f_i^{eq}) - \frac{\delta t}{(\tau + \delta t/2)} \frac{\delta t}{2} F_i + F_i \delta t \end{aligned}$$

En factorisant les second et troisième termes on obtient :

$$\bar{f}_i^* = \bar{f}_i - \frac{\delta t}{\tau + \frac{\delta t}{2}} \left(\bar{f}_i - f_i^{eq} + \frac{\delta t}{2} F_i \right) + F_i \delta t$$

Si on pose :

$$\bar{f}_i^{eq} = f_i^{eq} - \frac{\delta t}{2} F_i$$


Alors on obtient :

$$\bar{f}_i^* = \bar{f}_i - \frac{\delta t}{\tau + \frac{\delta t}{2}} (\bar{f}_i - \bar{f}_i^{eq}) + F_i \delta t$$

Encore une fois si on pose $\bar{\tau} = \tau/\delta t$ alors :


$$\bar{f}_i^* = \bar{f}_i - \frac{1}{\bar{\tau} + 1/2} (\bar{f}_i - \bar{f}_i^{eq}) + F_i \delta t$$

qui est l'équation d'évolution LBE utilisée dans ce document.

		Note Technique DES	Page 115/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

Bibliographie

- [1] T. Krüger, H. Kusumaatmaja, A. Kuzmin, O. Shardt, G. Silva, E. Viggien, The Lattice Boltzmann Method : Principles and Practice, Graduate Texts in Physics, Springer, 2016.
- [2] H. C. Edwards, C. R. Trott, D. Sunderland, Kokkos : Enabling manycore performance portability through polymorphic memory access patterns, Journal of Parallel and Distributed Computing 74 (12) (2014) 3202 – 3216. doi:10.1016/j.jpdc.2014.07.003.
- [3] Compatibilities of Kokkos library. Web link accessible on 12 May 2020 : <https://github.com/kokkos/kokkos/wiki/Compiling>.
- [4] Logiciel de visualisation Paraview <https://www.paraview.org/>.
- [5] W. Verdier, P. Kestener, A. Cartalade, Performance portability of lattice Boltzmann methods for two-phase flows with phase change, Computer Methods in Applied Mechanics and Engineering 370 (2020) 113266. doi:10.1016/j.cma.2020.113266.
- [6] A. Younsi, A. Cartalade, On anisotropy function in crystal growth simulations using lattice boltzmann equation, Journal of Computational Physics 325 (2016) 1 – 21. doi:<https://doi.org/10.1016/j.jcp.2016.08.014>.
- [7] W. Verdier, Modèle à champ de phase pour les systèmes ternaires diphasiques, Tech. Rep. 2021-67858/A, CEA, dES-ISAS-DM2S-STMF-LMSF (2021).
- [8] T. Boutin, W. Verdier, A. Cartalade, Grand-potential-based phase-field model of dissolution/precipitation : Lattice boltzmann simulations of counter term effect on porous medium, Computational Materials Science 207 (2022) 111261. doi:<https://doi.org/10.1016/j.commatsci.2022.111261>.
- [9] H. Huang, M. Sukop, X.-Y. Lu, Multiphase Lattice Boltzmann Methods. Theory and Application, Wiley & Sons, 2015.
- [10] Q. Li, K. Luo, Q. Kang, Y. He, Q. Chen, Q. Liu, Lattice Boltzmann methods for multiphase flow and phase-change heat transfer, Progress in Energy and Combustion Science 52 (2016) 62 – 105. doi:10.1016/j.pecs.2015.10.001.
- [11] D. Anderson, G. McFadden, A. Wheeler, Diffuse-interface methods in fluid mechanics, Annual Reviews of Fluid Mechanics 30 (1998) pp. 139–165. doi:10.1146/annurev.fluid.30.1.139.
- [12] N. Provatas, K. Elder, Phase-Field Methods in Materials Science and Engineering, Wiley-VCH, 2010.
- [13] I. Singer-Loginova, H. M. Singer, The phase field technique for modeling multiphase materials, Reports on Progress in Physics 71 (2008) 106501. doi:10.1088/0034-4885/71/10/106501.
- [14] A. Cartalade, Modèles à champ de phase et équations fractionnaires simulés par méthodes de Boltzmann sur réseaux, thèse d'Habilitation à Diriger des Recherches, Université Paris-Sud (2019). doi:10.13140/RG.2.2.10705.07529.
- [15] S. Alliance Service Plus, Manuel utilisateurs du cluster ORCUS, Tech. Rep. 2020-66362/B, CEA, DES/ISAS/DM2S/DIR (2021).

		Note Technique DES	Page 116/116
		Réf : STMF/LMSF/NT/2022-70869	
		Date : 21/11/2022	Indice : A
	LBM_saclay : code HPC multi-architectures sur base LBM. Guide du développeur.		

- [16] W. Verdier, Phase-field modelling and simulations of phase separation in the two-phase nuclear glass $\text{Na}_2\text{O}-\text{SiO}_2-\text{MoO}_3$, Ph.D. thesis, Thèse CEA–Institut Polytechnique de Paris (2022).
- [17] T. Boutin, Modèle champ de phase de dissolution d'un solide à 3 composants : application à l'altération de verres de stockages borosilicatés, Tech. Rep. 2021-70883/A, CEA, dES-ISAS-DM2S-STMF-LMSF (2022).
- [18] A. Fakhari, M. Rahimian, Phase-field modeling by the method of lattice Boltzmann equations, Physical Review E 81 (2010) 036707. [doi:10.1103/PhysRevE.81.036707](https://doi.org/10.1103/PhysRevE.81.036707).
- [19] A. Karma, W.-J. Rappel, Phase-field method for computationally efficient modeling of solidification with arbitrary interface kinetics, Physical Review E 53 (4) (1996) R3017–R3020. [doi:10.1103/PhysRevE.53.R3017](https://doi.org/10.1103/PhysRevE.53.R3017).
- [20] A. Karma, W.-J. Rappel, Quantitative phase-field modeling of dendritic growth in two and three dimensions, Physical Review E 57 (4) (1998) pp. 4323–4349. [doi:10.1103/PhysRevE.57.4323](https://doi.org/10.1103/PhysRevE.57.4323).
- [21] A. Cartalade, A. Younsi, M. Plapp, Lattice boltzmann simulations of 3D crystal growth : Numerical schemes for a phase-field model with anti-trapping current, Computers & Mathematics with Applications 71 (9) (2016) 1784–1798. [doi:10.1016/j.camwa.2016.02.029](https://doi.org/10.1016/j.camwa.2016.02.029).
- [22] S. Chen, G. Doolen, Lattice boltzmann method for fluid flows, Annual Reviews of Fluid Mechanics 30 (1998) pp. 329–364. [doi:10.1146/annurev.fluid.30.1.329](https://doi.org/10.1146/annurev.fluid.30.1.329).
- [23] F. Dubois, Equivalent partial differential equations of a lattice Boltzmann scheme, Computers & Mathematics with Applications 55 (7) (2008) 1441 – 1449, mesoscopic Methods in Engineering and Science. [doi:10.1016/j.camwa.2007.08.003](https://doi.org/10.1016/j.camwa.2007.08.003).