How to improve the numerical reproducibility of hydrodynamics simulations: analysis and solutions for one open-source HPC software

Rafife Nheili

THÈSE

Pour obtenir le grade de

Docteur

Délivré par

UNIVERSITE DE PERPIGNAN VIA DOMITIA

Préparée au sein de l'école doctorale ED305
Et de l'unité de recherche DALI-LIRMM

Spécialité : Informatique

Présentée par **Rafife Nheili**

How to improve the numerical reproducibility of hydrodynamics simulations: analysis and solutions for one open-source HPC software

Soutenance prévue le 7 décembre 2016 devant le jury composé de

Jocelyne ERHEL Directrice de recherches, INRIA, Rennes	*Rapporteur*
Arnaud LEGRAND Chargé de recherches HDR, CNRS, Grenoble	*Rapporteur*
Nathalie REVOL Chargée de recherches, INRIA, ENS Lyon	*Examinatrice*
Jean-Marie CHESNEAUX Professeur, U. Pierre et Marie Curie, Paris	*Examinateur*
Matthieu MARTEL Professeur, U. Perpignan Via Domitia	*Examinateur*
Jean-Michel HERVOUET Chercheur Sénior HDR (émérite), EDF R&D, Chatou	*Membre invité*
Philippe LANGLOIS Professeur, U. Perpignan Via Domitia	*Directeur de thèse*
Christophe DENIS Chargé de formation et de recherches, ENS Cachan	*Co-directeur de thèse*

Acknowledgements

Je voudrais tout d'abord exprimer mes sincères remerciement à mon directeur de thèse, Philippe Langlois, pour son soutien, sa motivation et pour m'avoir apporté les connaissances nécessaires tout au long de mon doctorat. Il m'a toujours guidé dans la bonne direction et m'a aidé à surmonter de nombreux défis. J'associe à ces remerciement d'encadrement mon co-directeur Christophe Denis.

Mes sincères remerciements vont également à Jean-Michel Hervouet pour ses conseils judicieux qui m'ont aidés à avancer dans mes travaux, ainsi qu'à toute l'équipe d'open-Telemac d'EDF R&D Chatou.

Je remercie Jocelyne Erhel et Arnaud Legrand pour leur travail de rapporteur qu'ils ont du réaliser dans un délai très court, ainsi que Nathalie Revol, Jean-Marie Chesneaux et Matthieu Martel de participer au jury de cette thèse.

Je tiens à remercier les membres de l'équipe DALI, les permanents Bernard Goossens, David Parello, David Defour, Guillaume Revy, Christophe Nègre, ainsi que Sylvia Munoz et les doctorants Hugues de Lassus Saint-Geniès, Cathy Porada, Jean-Marc Robert, pour l'environnement agréable et le parfait esprit d'équipe. J'aimerais spécialement remercier mes deux collègues de bureau Chemseddine Chohra et Amine Najahi pour les moments agréables que nous avons passés ensemble, les discussions que nous avons eues, pour leur aide et leurs encouragements.

Ces trois années ont été financées par les programme Erasmus-Mundus Peace 2, l'Université de Perpignan et EDF R&D. Je tiens donc à remercier les organisateurs du programme Erasmus-Mundus Peace 2, en particulier Sandrine Canadas et Simona Brajou de l'Université de Perpignan. Je remercie surtout Simona, accompagnante des boursiers de ce programme, pour son investissement qui va souvent bien au-delà de sa mission.

Un grand merci à mes amis, en particulier Rawa et mon adorable petite communauté syrienne à Perpignan, pour leur soutien moral et leur présence à mes côtés qui m'ont permis de me sentir moins éloigné de mon pays.

Je suis extrêmement reconnaissante à mes parents et mes sœurs qui, dans leur situation terrible à Alep, ont cru en moi et m'ont donné la force dont j'avais besoin pour accomplir ce travail et surmonter les étapes difficiles que j'ai vécues. Ils me manquent beaucoup, tout comme ma ville bien-aimée.

Je dois ma plus profonde gratitude à mon mari Hassan et à mon adorable fils Adel. Je remercie Hassan pour son amour, sa compréhension et son grand soutien pour la réussite de ma thèse. Je voudrais m'excuser auprès d'Adel qui a connu des moments difficiles avec une maman doctorante !

J'offre ce travail à mes quatre chéris Hassan, Adel, Maman et Papa.

Rafife

Résumé en français

Extrait du guide à l'usage du candidat au Doctorat en vue de la soutenance de thèse (version du 25/06/2015) du collège doctoral de l'Université de Perpignan via Domitia : *La langue des thèses et mémoires dans les établissements publics et privés d'enseignement est le français, sauf exceptions justifiées par les nécessités de l'enseignement des langues et cultures régionales ou étrangères ou lorsque les enseignants sont des professeurs associés ou invités étrangers. La maîtrise de la langue française fait partie des objectifs fondamentaux de l'enseignement.(· · ·) Lorsque cette langue n'est pas le français, la rédaction est complétée par un résumé substantiel en langue française. Pour l'ED 305 une synthèse d'une vingtaine de pages en français est intégrée au manuscrit, mettant en évidence les apports principaux du travail de thèse.*

Introduction

Les barrages et les centrales exploitées en bord de mer ou de rivières interagissent avec l'environnement aquatique qu'il faut protéger et dont il faut se protéger. L'hydrodynamique est ainsi stratégique pour de nombreux services du secteur public et industriel. Par exemple, depuis plus de 20 ans EDF R&D développe openTelemac : un logiciel (libre) de simulation numérique de phénomènes hydrodynamiques. Les 300 000 lignes de ce code Fortran 90 résolvent, par la méthode des éléments finis, les équations de Saint-Venant qui décrivent les écoulements à surface libre en eaux peu profondes. Ces simulations calculent principalement la hauteur d'eau et sa vitesse en chaque point d'un maillage du domaine d'étude : une vallée, une côte, un bassin, *etc.* Les phénomènes de taille réelle génèrent des volumes de simulation très importants qui sont rendus possibles grâce au calcul parallèle. Il permet de traiter simultanément ces calculs en les distribuant entre plusieurs processeurs pour ainsi effectuer le plus grand nombre d'opérations en un temps le plus réduit possible. En pratique, le maillage du domaine est décomposé et distribué entre autant d'unités de calcul que nécessaire. Ses simulations, complexes

et de grandes tailles, ont par exemple permis d'éviter récemment une dépense de 12 millions d'euros pour l'étude de l'impact d'un ouvrage d'art en situation de crue. Les calculs de simulations sont réalisés avec les nombres flottants que nous allons maintenant présenter.

L'arithmétique des nombres flottants et ses propriétés

Ils permettent d'exprimer les réels sous la forme :

$$x = (-1)^s \cdot m \cdot \beta^e, \tag{1}$$

où $s \in \{0, 1\}$. La mantisse m est une suite d'entiers qui dépend de la base $\beta > 1$ ($0 \leq m_i < \beta$). Le nombre de chiffres, ou de bits quand $\beta = 2$, de cette mantisse indique la précision de x et est noté t. L'exposant e est représenté par w chiffres/bits, qui sert comme un facteur d'échelle pour le nombre flottant x. Une telle écriture ne représente exactement qu'un nombre fini de réels : pour les autres se pose le problème de leur approximation, de leur arrondi. Sur les ordinateurs, les nombres flottants respectent depuis 1985 la norme IEEE-754 qui réalise un compromis entre codage puissant et calculs efficaces.

Les formats IEEE-754

Le standard IEEE 754 définit les formats de représentation de nombres flottants. Ces formats ont évolué entre la version normalisée en 1985 [31] et la révision de 2008 [32]. Cette dernière introduit par exemple des flottants binaires et des flottants décimaux. Elle distingue aussi des formats dits d'échange et des formats dits étendus ou extensibles. Pour plus de détails, nous renvoyons le lecteur à [48] par exemple.

La norme définit entre autres les deux formats binaire *binary32* et *binary64*. Ils correspondent aux simple et double précisions de la version 1985 et sont à ce jour les plus couramment disponibles et utilisés en pratique. Ces flottants sont respectivement représentés en machine sur 32 et 64 bits. Nous mentionnons aussi le format *binary64 étendu* introduit dès la version 1985 pour prendre en compte les registres 80 bits des processeurs x86. Ces formats sont illustrés par la Table 1

TABLE 1: Les paramètres des 3 formats binaires principaux IEEE-754.

Paramètre	Format		
	binary32	*binary64*	*binary64 étendu*
Précision, t	24	53	≥ 64
e_{max}	+127	+1023	$\geq +16383$
e_{min}	-126	-1022	≤ -16382
Taille de l'exposant, w	8	11	≥ 15
Taille du format en bits	32	64	≥ 79

Les fonctions d'arrondi

Les fonctions d'arrondi, notées $\circ$, sont appliquées aux nombres flottants et leurs opérations. Elles sont nécessaires pour représenter un nombre ou le résultat d'une opération non représentables, comme par exemple les constantes π ou $1/10$, cette dernière s'écrivant en binaire avec le développement infini $0.0\underline{0011}00110011\cdots$. L'opération d'arrondi remplace donc un nombre réel par une approximation représentable en nombre flottant. Donc pour un réel non représentable $x \in \mathbb{R}$, $\circ(x) = \hat{x}$ est le nombre flottant $\hat{x} \in \mathbb{F}$ résultant de cette fonction d'arrondi. Bien sûr $\circ(\hat{x}) = \hat{x}$ si $\hat{x} \in \mathbb{F}$.

Les modes d'arrondi

Le standard IEEE-754 définit comment une valeur numérique est arrondie en nombre flottant en introduisant plusieurs modes d'arrondi. Nous énumérons ces quatre modes qui sont également illustrés à la Figure 1.

— l'arrondi au plus près :
- $RN(x) = \hat{x} \in \mathbb{F}$, où $|\hat{x} - x|$ est minimum et $\hat{x}$ est rendu unique par une stratégie de choix si besoin (cas où x est le milieu de deux flottants).

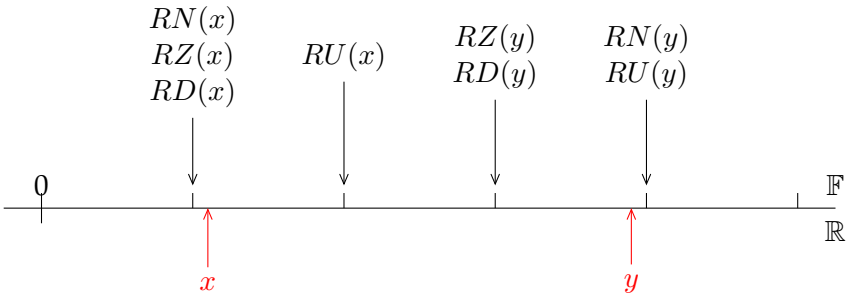
— l'arrondi vers le bas :
- $RD(x) = max_{\hat{x}\in\mathbb{F}}(\hat{x} \leq x)$, ainsi $\hat{x}$ est le plus grand nombre flottant qui est plus petit ou égal à x.

— l'arrondi vers le haut :
- $RU(x) = min_{\hat{x}\in\mathbb{F}}(\hat{x} \geq x)$, ainsi $\hat{x}$ est le plus petit nombre flottant qui est plus grand ou égal à x.

— l'arrondi vers vers 0 :
- $RZ(x) = RD(x)$ si $x \geq 0$,

- $RZ(x) = RU(x)$ si $x \leq 0$.

FIGURE 1: Les modes d'arrondi pour $x, y > 0$.



Faiblesses de l'arithmétique flottante

À chaque arrondi, nous perdons *a priori* de la précision qui correspond aux erreurs d'arrondi. Cette erreur d'arrondi est limitée par la précision arithmétique si t est le nombre de bits de la mantisse m de (1), il est classique de noter **u** la précision de travail qui vérifie pour le mode d'arrondi RN :

$$\mathbf{u} = 2^{-t},$$

et pour les autres modes d'arrondi :

$$\mathbf{u} = 2^{1-t}.$$

A cause de l'arrondi, les opérations arithmétiques des nombre flottants ne sont pas exactes. En général, le résultat d'un calcul informatique est différent de son résultat mathématique : si, en théorie $a + b = c$, on obtient en pratique $a + b = c + e$ où e est l'erreur d'arrondi de l'addition de a et b. Parfois, si ces deux opérandes sont trop différents (en ordre de grandeur, c'est-à-dire en exposant), alors la plus petite valeur sera absorbée par la plus grande et aura disparue du calcul. A cause de ces imperfections, l'associativité des opérations mathématiques n'est plus valide en arithmétique flottante :

$$(a + b) + c \neq a + (b + c).$$

Il en est du même de la distribution : $(a + b) \times c \neq a \times c + b \times c$. Et ceci est inévitable même sur l'ordinateur le plus puissant du monde.

Qu'est-ce que la reproductibilité numérique ?

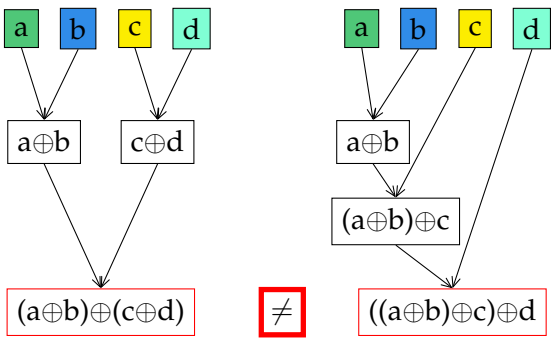
Un logiciel calcule de façon reproductible quand tous les bits de ses résultats sont identiques pour plusieurs exécutions successives, et ce même pour un nombre d'unités de calcul qui varie d'une exécution à l'autre. En effet, en calcul parallèle, les ordres de calcul sont souvent indéterminés et, à cause de la perte de l'associativité de l'addition flottante, les résultats peuvent être différents entre exécutions séquentielle (sur 1 seul processeur) et parallèle (sur plusieurs processeurs), et entre les exécutions parallèles elles-mêmes.
Un tel besoin est motivé pour faciliter le débogage, le test et la validation d'un code scientifique. Quand des exécutions non-reproductibles mènent à des résultats différents pour les mêmes entrées, l'élimination de bogue devient une tâche difficile où le développeur se pose la question : est-ce que ce bogue vient des erreurs d'arrondi ou d'un autre problème dans le programme ?
Il est important de noter que l'obtention de la reproductibilité ne correspond pas à améliorer, ni même à valider la précision des résultats car un résultat même reproductible peut être loin du résultat exact.
Dans un contexte de simulation numérique, le résultat exact est inconnu. En pratique, les développeurs des simulations ont confiance en leurs résultats séquentiels et les considèrent souvent comme les résultats de référence. C'est en particulier le cas pour le code auquel ce travail s'intéresse. Une mesure importante est donc de comparer les nouveaux résultats reproductibles à ces "références".
L'échec de la reproductibilité numérique a été signalée dans divers domaines d'application de la simulation HPC, comme en énergie [68], en météorologie [23], en dynamique moléculaire [63] ou en dynamique des fluides [55].

Les causes principales de la non-reproductibilité

Le parallélisme utilise simultanément plusieurs processeurs pour résoudre un problème. Ce problème est distribué vers ces processeurs.

MPI (Message Passing Interface) est une norme définissant une bibliothèque de fonctions utilisables avec les langages C, C++ et Fortran. Elle permet d'exploiter des ordinateurs distants ou multiprocesseurs par passage de messages [45]. En pratique les processeurs ne sont pas totalement indépendants les uns des autres, mais ont besoin de communiquer. Il y a deux types de communications, i) les communications point-à-point qui permettent à deux processeurs à l'intérieur d'un même communicateur d'échanger une donnée, ii) les communications collectives qui impliquent tous les processeurs d'un communicateur. Il est possible d'envoyer une même donnée à tous les processeurs, de découper un vecteur entre eux ou d'effectuer une opération (par exemple une addition) où chaque processeur contribuera. Cette dernière est nommée une réduction. Les modèles MPI sont par défaut non-déterministes. L'ordre d'arrivée des messages envoyés à partir de deux processeurs p_1 et p_2, à un troisième p_3 n'est pas défini. Donc MPI garantit seulement que deux messages envoyés par p_1 arriveront à p_2 dans l'ordre de l'envoi. La Figure 2 illustre comment une réduction parallèle peut mener à des résultats différents pour les mêmes entrées a, b, c, d et la même opération $a+b+c+d$. Et à cause des différents ordres de cette addition et de la non-associativité de l'arithmétique flottante, les résultats ne sont pas reproductibles, même pour deux exécutions successives avec un nombre de processeurs fixé.

FIGURE 2: La non-associativité de la réduction parallèle.

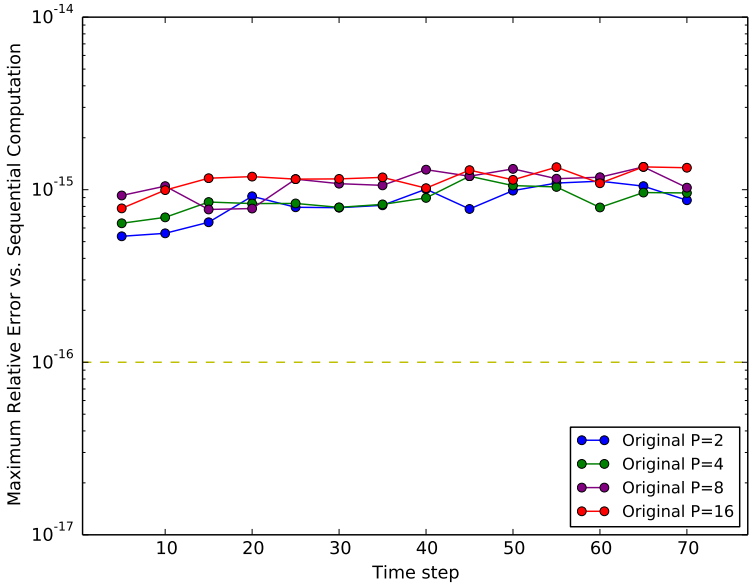


La non-reproductibilité d'openTelemac : Tomawac et Telemac-2D

Dans ce travail, nous nous intéressons à deux modules inclus dans openTelemac, Tomawac et Telemac-2D, que nous allons décrire dans ce qui suit.

— **Tomawac** Ce module est utilisé pour modéliser la propagation des ondes dans les zones côtières. Par la méthode des éléments finis, il résout une équation simplifiée pour la densité spectre-angulaire de l'action des vagues [25]. Ceci est fait pour des conditions stables. Il prend en compte plusieurs phénomènes physiques, *e.g.* des vagues générées par le vent, la réfraction sur le fond, la réfraction par les courants, *etc.*
A chaque nœud du maillage, il calcule la hauteur significative des vagues, la fréquence moyenne et la direction moyenne de l'onde, la fréquence de l'onde de crête, les courants induits par les vagues et les contraintes de rayonnement.
Il résout une équation de transport où une EDP de premier ordre est modifiée en EDO le long de ses courbes caractéristiques. Dans la pratique, ceci revient à résoudre un système linéaire diagonal. Nous étudions la non-reproductibilité du cas de test *Nice* inclus dans la distribution de Tomawac. Il simule l'impact sur la baie de Nice, de la houle de sillage générée par les navires grande vitesse qui relient la Corse au continent. La Figure 3 illustre les maximas des erreurs relatives entre l'exécution séquentielle et celles en parallèles sur p processeurs (axe des y), pour plusieurs pas de temps (axe des x), et pour les nombres de processeurs $p = 2, 4, 8, 16$. Les courbes colorées correspondent aux nombres de processeurs et la ligne pointillée représente la précision du calcul (*binary64*) avec $\mathbf{u} \approx 1.1 \times 10^{16}$.
Si il n'y avait pas d'erreur entre exécutions séquentielles et parallèles, les courbes colorées se chevaucheraient toutes sur la ligne pointillée. Si les résultats parallèles étaient au moins identiques, les courbes colorées se seraient chevauchées les unes les autres, ce qui n'est pas le cas.

FIGURE 3: Calcul original en virgule flottante.
Fréquence moyenne des vagues, module Tomawac,
cas de test *Nice*.



— **Telemac-2D** Ce module est un module d'hydrodynamique en 2D qui résout les équations de Saint Venant en utilisant la méthode des éléments finis pour un maillage d'éléments triangulaires. Il prend en compte plusieurs phénomènes, *e.g.* la propagation des ondes longues, le frottement sur le fond, la pression atmosphérique et le vent, *etc.*
Les résultats principaux à chaque nœud du maillage sont la hauteur de l'eau H et les deux composantes U, V de la vitesse. La méthode des éléments finis conduit à construire et résoudre un système linéaire qui est un mélange des trois sous-systèmes en H, U et V. Nous étudions la non-reproductibilité du cas de test *gouttedo* qui est inclus dans la distribution de ce module. Il s'agit d'une simulation en 2D de l'impact de la chute d'une goutte d'eau dans un bassin carré. Cette simulation utilise un maillage d'élément triangulaire (8978 éléments, 4624 nœuds) et plusieurs pas de temps de 0.2 seconde.

Les Figures 4, 5 et 6 illustrent la non-reproductibilité de la hauteur d'eau entre une exécution séquentielle et celles en parallèles avec $p = 2, 4, 8$, respectivement.

— Les résultats de gauche représentent à chaque nœud du maillage la valeur de la hauteur d'eau pour la simulation séquentielle.

— Celles du milieu correspondent aux résultats parallèles. Les points blancs sont les résultats qui diffèrent de ceux calculés en séquentiel. La perte de valeurs reproductibles augmente dans le temps, car les résultats dépendent du pas de temps précédent.

— Les résultats de droite présentent, à chaque nœud du maillage, l'erreur relative maximum entre une exécution séquentielle et celles en parallèles.

FIGURE 4: Les points blancs (milieu) sont les valeurs non-reproductibles de la hauteur d'eau entre les exécutions séquentielle (gauche) et avec 2 processeurs (milieu). À droite, l'erreur relative entre ces simulations.
Pas de temps : 1, 2, . . ., 7, 8 $\times$ 0.2 sec, module Telemac-2D, cas de test *gouttedo*.

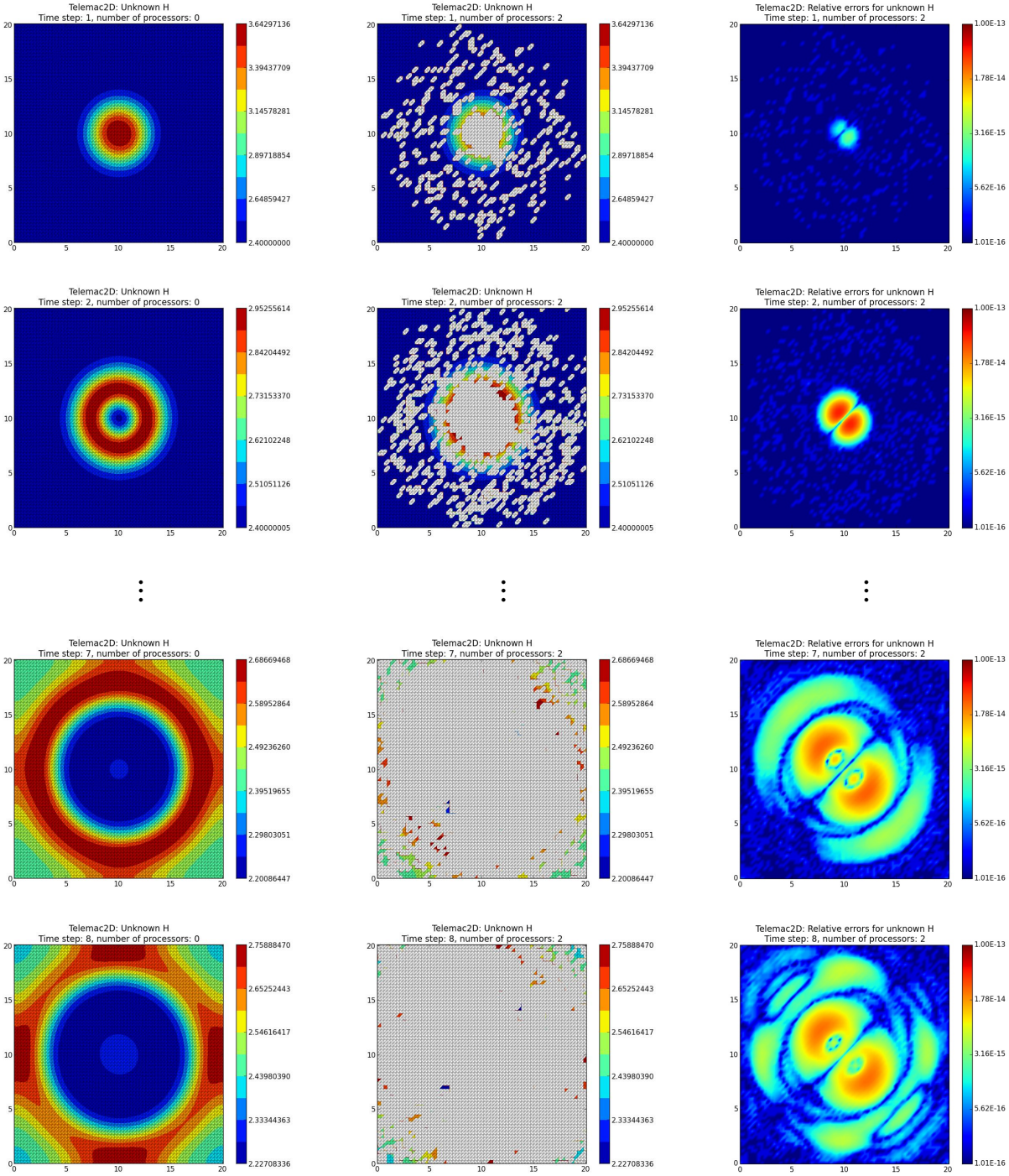


FIGURE 5: Les points blancs (milieu) sont les valeurs non-reproductibles de la hauteur d'eau entre les exécutions séquentielle (gauche) et avec 4 processeurs (milieu). À droite, l'erreur relative entre ces simulations.
Pas de temps : 1, 2, . . ., 7, 8 $\times$ 0.2 sec, module Telemac-2D, cas de test *gouttedo*.

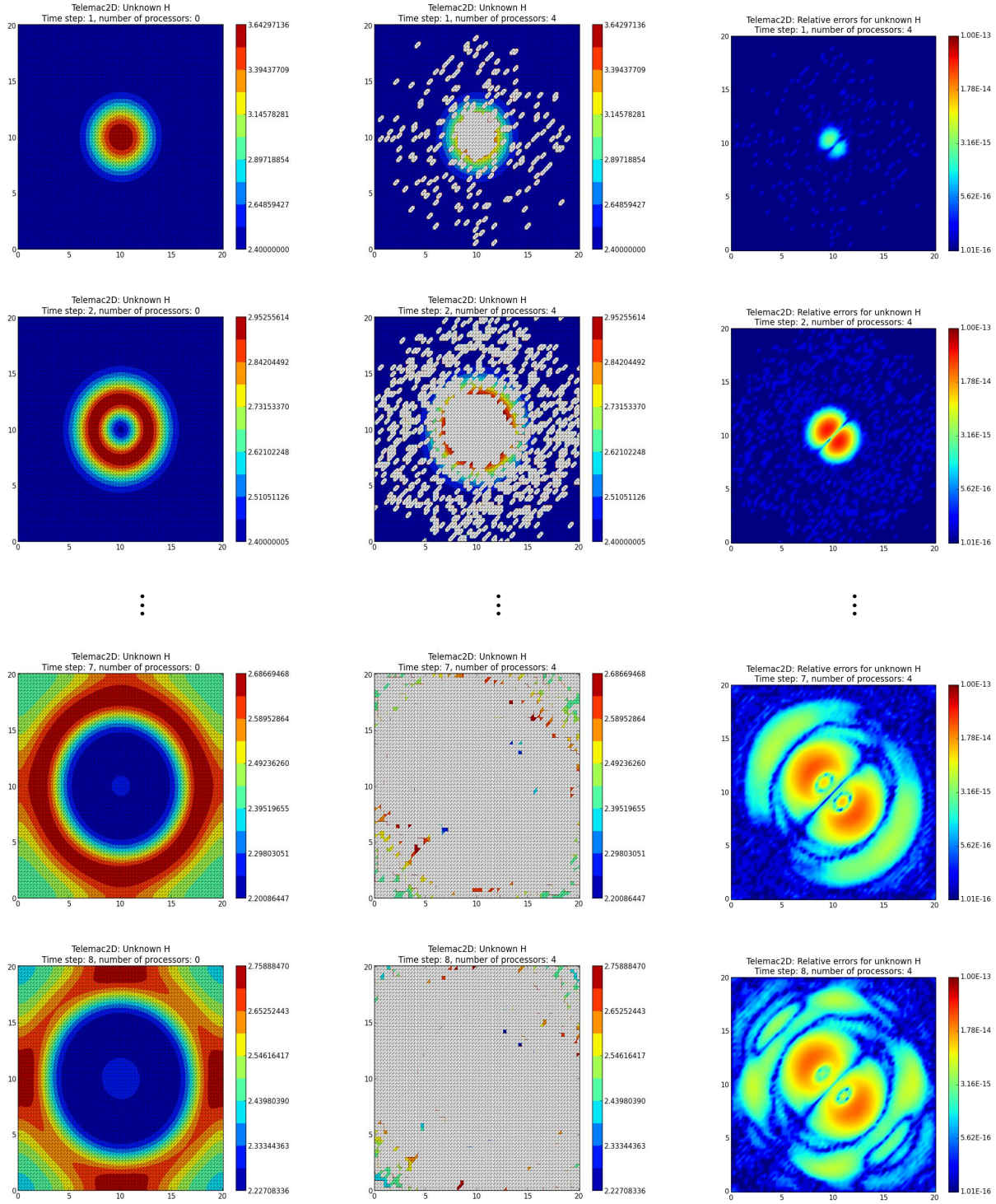
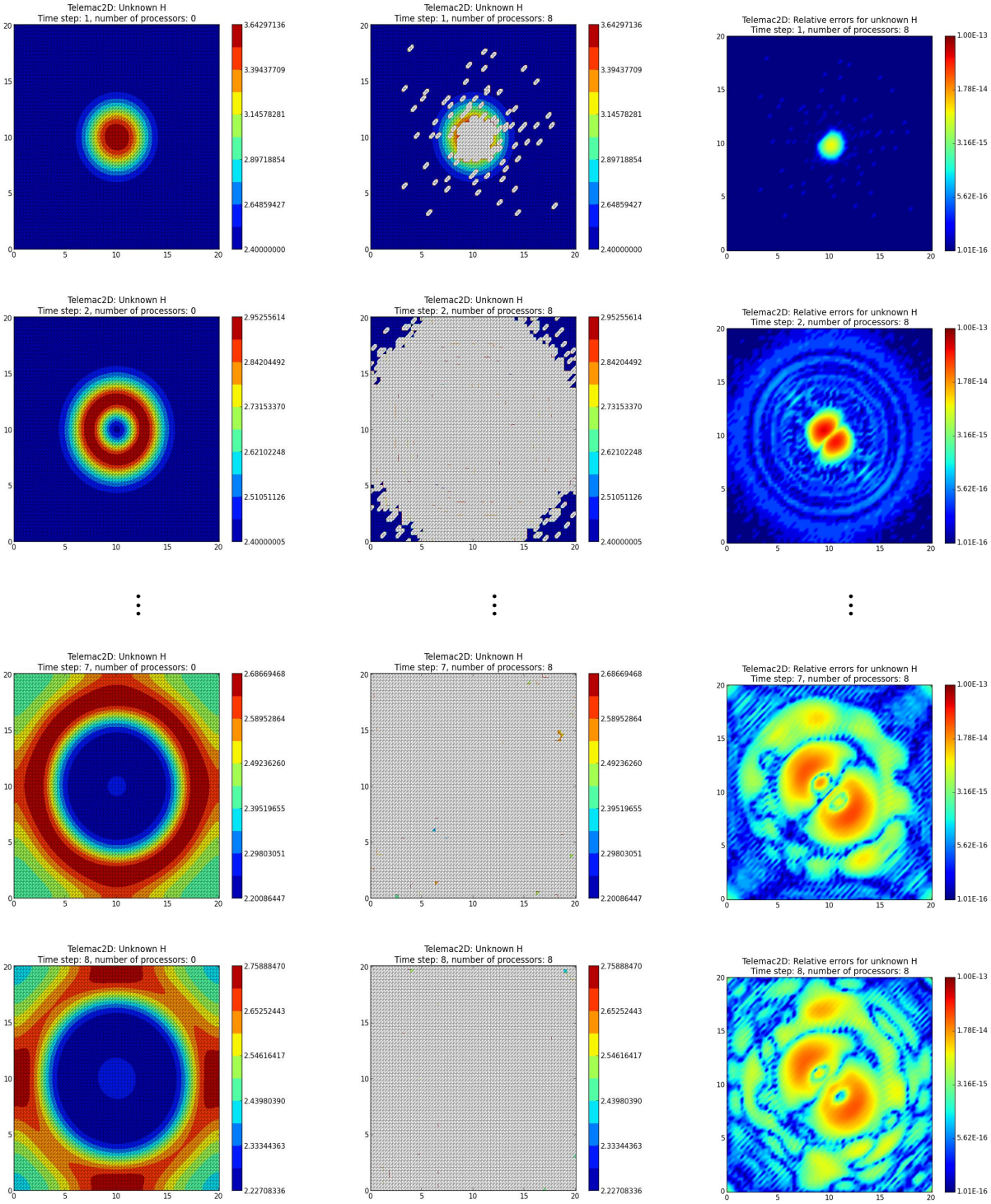


FIGURE 6: Les points blancs (milieu) sont les valeurs non-reproductibles de la hauteur d'eau entre les exécutions séquentielle (gauche) et avec 8 processeurs (milieu). À droite, l'erreur relative entre ces simulations.
Pas de temps : 1, 2, . . ., 7, 8 $\times$ 0.2 sec, module Telemac-2D, cas de test *gouttedo*.



Différentes méthodes pour obtenir la reproductibilité

Pour obtenir la reproductibilité numérique, nous devons identifier les sources qui produisent ce problème, puis ensuite appliquer aussi peu de modifications que possible afin de limiter leur sur-coût. Il existe plusieurs techniques pour produire des résultats reproductibles. Nous les présentons en deux groupes.

— **Obtenir la reproductibilité en améliorant la précision des résultats.** En arithmétique flottante, chaque opération élémentaire peut introduire une erreur d'arrondi qui est propagée au cours des calculs [48]. Par conséquence, l'erreur du résultat final peut même le rendre non significatif. Pour limiter cette perte de précision, les transformations sans erreur calculent les erreurs d'arrondi générées par chaque opération flottante. Le principe de ces transformations est que pour une opération $op \in \{+, -, *\}$ entre deux nombres flottants $\hat{a}$ et $\hat{b}$, il existe deux nombres flottants $\hat{x}$ et $\hat{y}$ qui vérifient :

$$\hat{a}\ op\ \hat{b} = \hat{x} + \hat{y}. \tag{2}$$

Ici $\hat{x} = \circ(\hat{a}\ op\ \hat{b})$, *i.e.* $\hat{x}$ est la partie arrondie du résultat, et $\hat{y}$ est l'erreur d'arrondi générée qui vérifie $|\hat{y}| \leq \frac{1}{2} ulp(\hat{x})$ [1].
La méthode de compensation utilise les transformations précédentes pour accumuler les erreurs d'arrondi dans un terme d'erreur tout au long du calcul, et finalement ajouter ce terme au résultat final. Un autre moyen d'améliorer la précision est d'utiliser, par exemple, les nombres double-double. Un nombre double-double x est représenté par une paire (x_h, x_l) de deux *binary64*. Les opérations double-double calculent avec deux fois plus de précision que la précision de travail [43]. Plusieurs études [65], [55], [23], ont comparé l'utilisation de la compensation avec les calculs en double-double, et il apparait que ces derniers engendrent un sur-coût plus important que la compensation.

— **Obtenir la reproductibilité indépendamment de la précision.** Ces techniques permettent d'éliminer les inconvénients du parallélisme qui conduisent à la non-reproductibilité numérique. Comme mentionné précédemment,

1. Définition de l'ulp en Relation 2.4, page 16

il s'agit de i) la différente propagation des erreurs d'arrondi entre les simulations quand le nombre de processeurs change et, ii) la réduction parallèle non-déterministe. Nous listons comme exemples trois techniques différentes.

— Les algorithmes de sommation reproductible proposés par Demmel et Nguyen [15] qui calculent un pré-arrondi des nombres flottants d'une façon identique pour les exécutions séquentielles et les exécutions parallèles.

— La transformation des nombres flottants en entiers qui garantit l'associativité de l'addition.

— Forcer un ordre spécifique de la réduction parallèle qui élimine les conséquences de la non-associativité.

Obtenir des simulations hydrodynamiques numériquement reproductibles

Nous avons rétabli la reproductibilité numérique au sein du logiciel open-Telemac pour des premiers cas simples, mais représentatifs, de ces simulations hydrodynamiques.

Reproductibilité de Tomawac

Dans ce cas, nous comparons trois méthodes : la compensation, l'algorithme de somme reproductible de Demmel et Nguyen et la conversion en entier. Ces trois méthodes sont testées sur le cas de test *Nice* où la source de non reproductibilité est uniquement l'assemblage éléments finis, et son complément, l'assemblage aux points d'interface dans le cas d'une exécution parallèle. En effet, le maillage du domaine est décomposé et distribué entre autant d'unités de calcul que nécessaire. Deux types de points apparaissent alors : ceux, internes, traités indépendamment par chaque processeur, et les points d'interfaces situés sur les frontières des sous-domaines et pour lesquels les processeurs doivent communiquer entre eux.

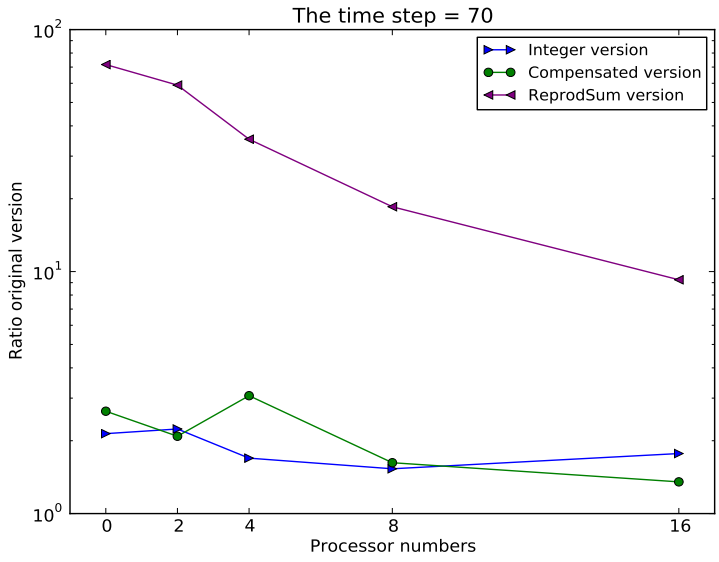
Le sur-coût de la reproductibilité de Tomawac

Nous comparons les trois méthodes reproductibles, ce qui va nous aider à choisir la meilleure méthode à utiliser pour un contexte plus compliqué.

En Figure 7, nous représentons le sur-coût de ces trois solutions où le nombre de processeurs varie ($p = 0$ pour une exécution séquentielle et $p = 2, 4, 8, 16$ pour celles en parallèle). L'axe des y est le ratio du sur-coût comparé à la version originale.

La solution compensée fournit des résultats plus précis avec un sur-coût de calcul raisonnable, il représente entre 2 à 3 fois le temps de calcul d'origine. La conversion en entier est aussi raisonnable en terme de sur-coût mais elle risque une perte de précision. L'algorithme de somme reproductible de Demmel et Nguyen est en général efficace, cependant dans ce contexte, il ajoute un sur-coût important (la courbe violette) qui varie de $\times 10$ à $\times 100$, et cela à cause des communications supplémentaires alors nécessaires.
Nous notons que la vitesse de l'implémentation parallèle n'apparait pas dans ces courbes parce que les parties mesurées sont dominées par toutes les communications entre les sous-domaines.

FIGURE 7: Tomawac reproductible : sur-coût des trois solutions (compensation, somme reproductible, conversion en entier) comparé au calcul original (cas de test *Nice*).



Reproductibilité de Telemac-2D

Dans ce cas plus complexe, nous appliquons la compensation qui était la plus efficace dans Tomawac. Dans ce module, la première source de non-reproductibilité est la propagation différente des erreurs d'arrondi générées aux nœuds d'interface. Nous notons que l'assemblage de ces points est implicitement appliqué dans plusieurs parties du calcul : phases de construction et de résolution du système linéaire.

Pour corriger, nous avons stocké et propagé ces erreurs tout au long du calcul, et enfin nous les utilisons pour compenser la valeur calculée après chaque étape de l'assemblage des nœuds d'interface. Ces corrections sont appliquées pour les simulations séquentielles et parallèles et ainsi obtenir la reproductibilité attendue.

La seconde source est la réduction dynamique de l'implémentation parallèle du produit scalaire dans les itérations de l'algorithme du gradient conjugué appliqué pour résoudre le système linéaire. Cet algorithme itératif est corrigé en appliquant un produit scalaire compensé qui fournit des résultats comme s'ils étaient calculés en deux fois la précision de travail. Le cas de test considéré *gouttedo* semble être bien conditionné avec des erreurs relatives entre les résultats séquentiels et ceux parallèles variant entre 10^{-15} et 10^{-13}. Cela nous permet d'estimer qu'une double précision de travail suffit pour calculer les résultats à la précision de l'arrondi de représentation. Ce qui conduit à des simulations reproductibles avec un seul niveau de compensation. Les Figures 8, 9, 10 illustrent des simulations maintenant reproductibles du cas de test *gouttedo*. Comparés aux originaux des Figures 4, 5, 6, il n'y a plus de point blanc qui apparait entre les exécutions séquentielles et parallèles quand le nombre p de processeurs varie, ici $p = 2, 4, 8$.

FIGURE 8: Reproductibilité numérique : aucun point blanc sur les valeurs de la hauteur d'eau entre les exécutions séquentielle (gauche) et à 2 processeurs (droite).
Pas de temps : 1, 2, …, 7, 8 $\times$ 0.2 sec, module Telemac-2D, cas de test *gouttedo*.

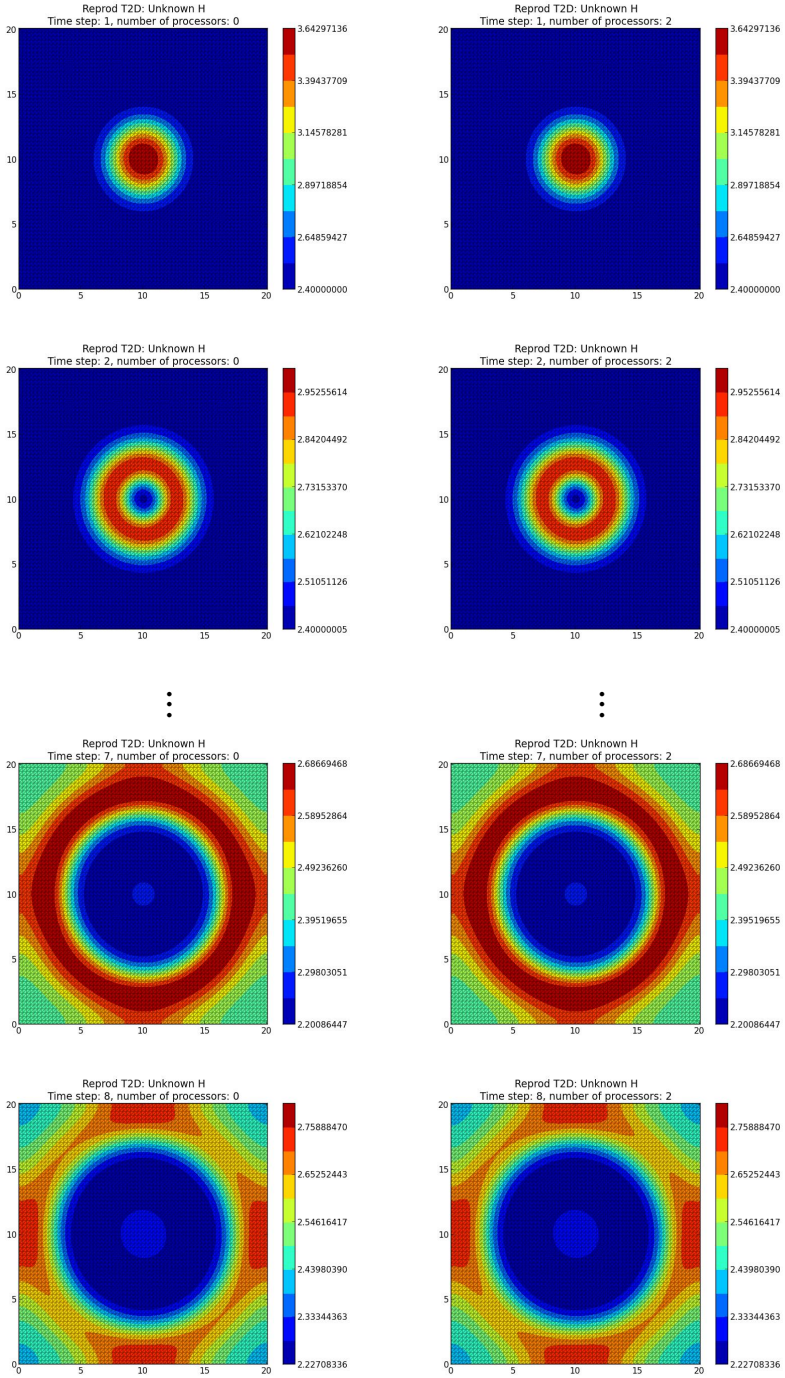


FIGURE 9: Reproductibilité numérique : aucun point blanc sur les valeurs de la hauteur d'eau entre les exécutions séquentielle (gauche) et à 4 processeurs (droite).
Pas de temps : 1, 2, …, 7, 8 $\times$ 0.2 sec, module Telemac-2D, cas de test *gouttedo*.

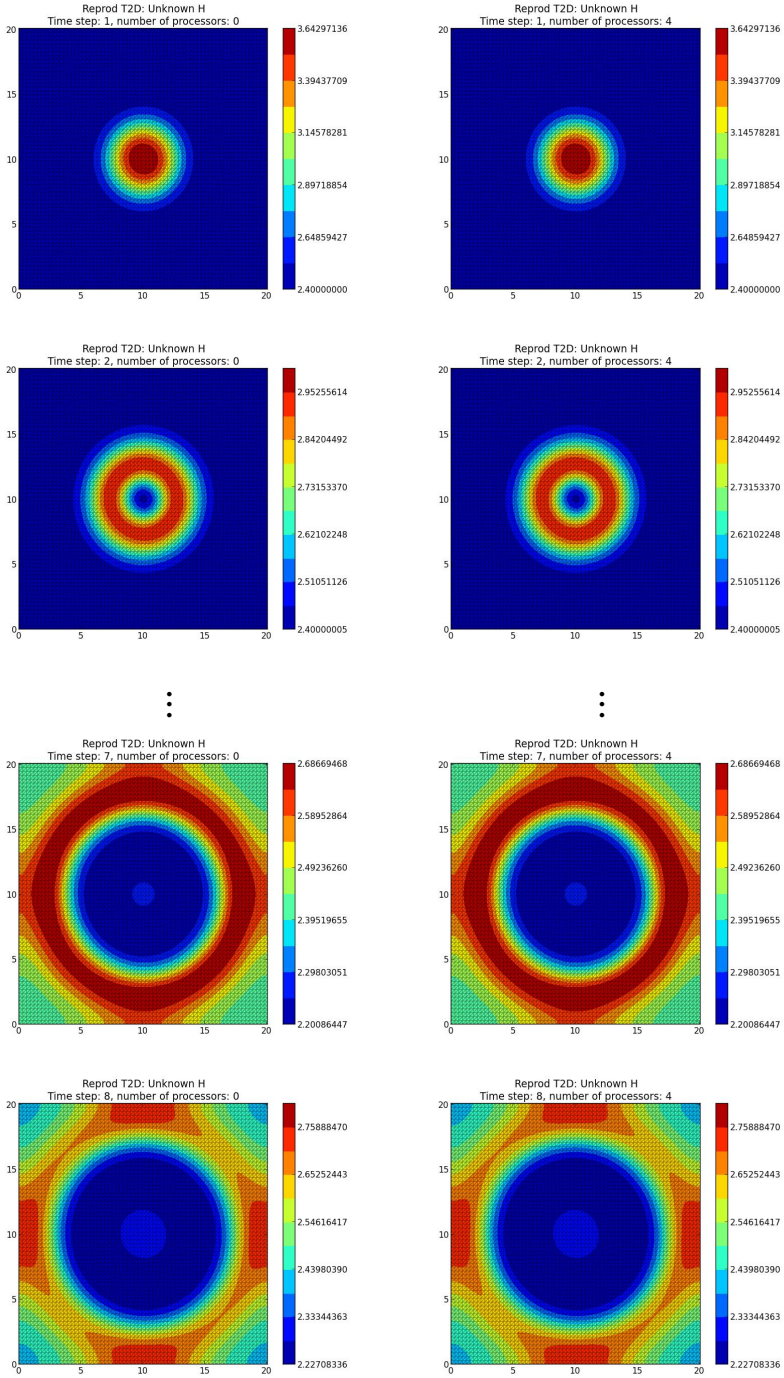
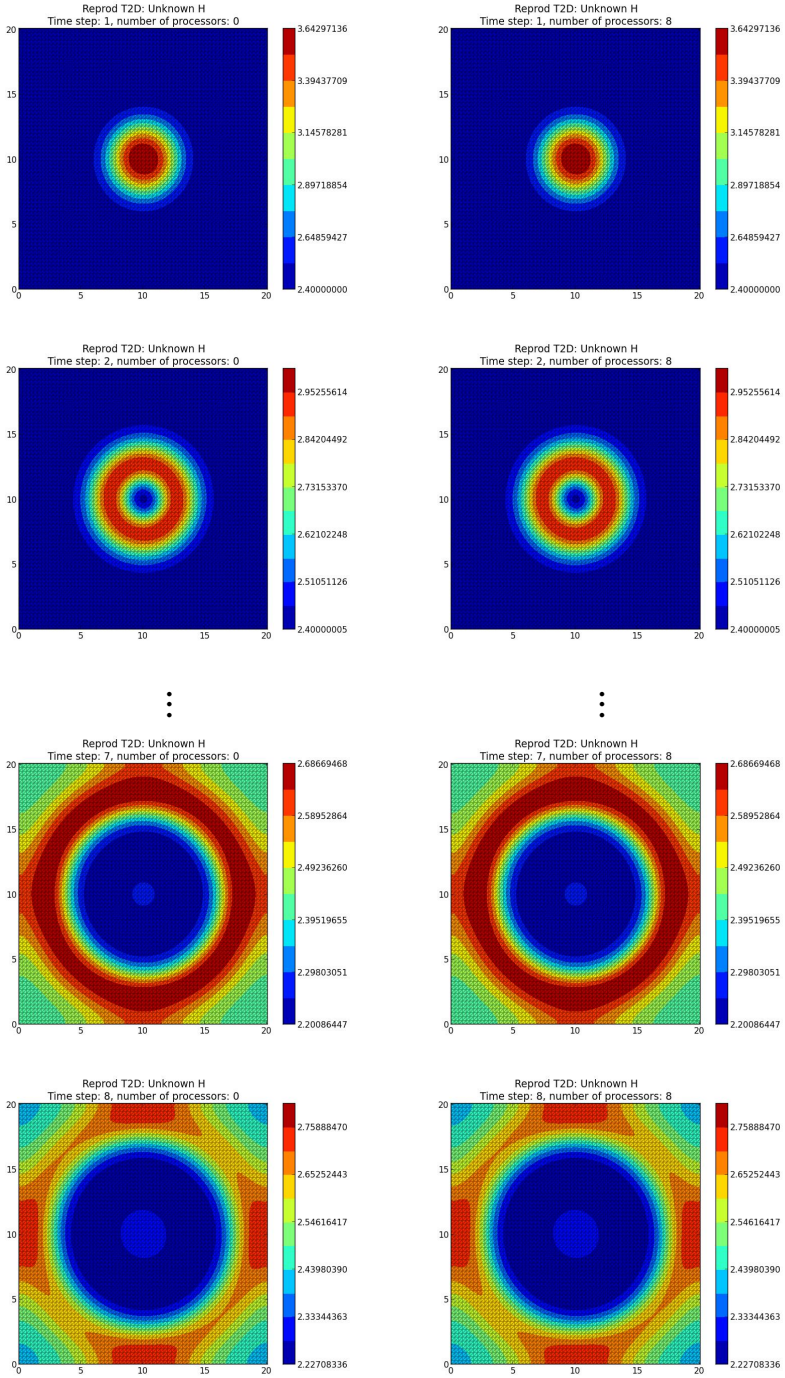


FIGURE 10: Reproductibilité numérique : aucun point blanc sur les valeurs de la hauteur d'eau entre les exécutions séquentielle (gauche) et à 8 processeurs (droite).
Pas de temps : 1, 2, …, 7, 8 $\times$ 0.2 sec, module Telemac-2D, cas de test *gouttedo*.



Le sur-coût de la reproductibilité de Telemac-2D

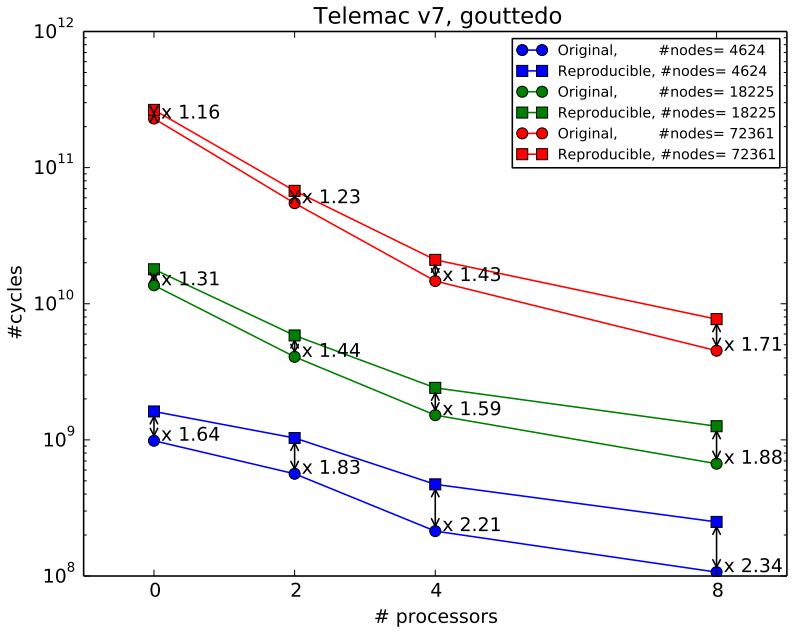
Nous mesurons le sur-coût de la reproductibilité sur la boucle du temps qui inclut toutes les modifications. Cela évite la prise en compte du grand taux de lecture/écriture de données de la simulation. Puisque le sur-coût dépend certainement de la taille de la simulation, nous comparons trois maillages différents avec 4624, 18225 et 72361 nœuds. La Table 2 présente l'importante augmentation du nombre de points d'interface quand le maillage est ainsi raffiné. Évidemment, cela correspond également à un coût de communication plus important.

TABLE 2: Le nombre de points d'interface (PI) pour 3 maillages différents quand le nombre de processeurs change.

PI	4624 nœuds	18225 nœuds	72361 nœuds
2 procs	72	143	280
4 procs	304	674	1368
8 procs	501	1152	2020

La Figure 11 présente le temps d'exécution, mesuré en cycles (axe des y), pour la version compensée reproductible (■) et l'originale en (●) quand le nombre de processeurs varie (axe des x). Il montre aussi le ratio entre ces deux versions où nous remarquons que la version compensée double plus ou moins le temps de calcul de la version originale.

FIGURE 11: Telemac-2D reproductible : Sur-coût du calcul reproductible (compensé) comparé à l'original (cas de test *gouttedo*).



Conclusion et perspectives

Les contributions présentées dans ce manuscrit ont fait l'objet des publications [40], [41], [49].

Nous avons étudié la non-reproductibilité numérique des simulations hydrodynamiques obtenues avec le code openTelemac qui est largement utilisé pour des applications industrielles et scientifiques. Ce code applique la méthode des éléments finis dont nous avons détaillé les étapes principales. Nous nous sommes plus particulièrement intéressé aux deux modules : Tomawac et Telemac-2D. Les principales questions auxquelles ce travail répond sont les suivantes. Quelles sont les sources de cette non-reproductibilité ? Comment retrouver la reproductibilité numérique en appliquant des techniques existantes dans ce code compliqué ? Quel est le coût de cette amélioration ? Dans Tomawac la seule source de non-reproductibilité est l'assemblage éléments finis.

Cette étape est appliquée uniquement sur les trois second membres du système diagonal qui est résolu. Nous avons récupéré leur reproductibilité numérique grâce à trois solutions existantes : la compensation [48], le récent algorithme de somme reproductible proposé par Demmel et Nguyen [15] et les transformations en entiers [51]. En comparant ces solutions, la technique de compensation apparait être la plus efficace et celle qui fournit des résultats précis. Ceci justifie notre choix d'appliquer la compensation dans le cas plus compliqué de Telemac-2D. Ce module construit un système linéaire à trois inconnues vectorielles H, U et V, où les trois sous-systèmes ne sont pas indépendants les uns des autres.
Dans ce cas, la première source de non-reproductibilité est causée par les différentes propagations des erreurs d'arrondis générées aux nœuds d'interfaces. Pour corriger cette étape, il fallait calculer, stocker et propager ces erreurs pour enfin les utiliser pour compenser chaque valeur calculée. Cette étape a été appliquée à chaque vecteur et au produit matrice-vecteur EBE dans la phase de résolution.
La seconde source est la réduction dynamique de MPI qui est appliquée pour calculer le produit scalaire global dans les itérations du gradient conjugué. Ce calcul est corrigé en appliquant un produit scalaire compensé qui fournit un résultat aussi précis que s'il avait été calculé avec une précision de travail doublée. Nous avons mesuré que le sur-coût en temps de calcul était également raisonnable.

Perspectives

Ce travail est une première étape vers la complète reproductibilité numérique du code openTelemac. Nous avons identifié et détaillé les sources de cette non-reproductibilité. Avec leur connaissance approfondie de code, les développeurs d'openTelemac sont optimistes : ils pensent qu'aucune autre source de non-reproductibilité demeure dans le code [26]. Cependant, nous n'avons corrigé que les fonctions utilisées pour les cas de tests étudiés. Donc, pour obtenir une complète reproductibilité numérique, la même méthodologie doit être appliquée au reste des calculs.

Comme mentionné, nous avons comparé l'efficacité de certaines méthodes

qui conduisent à la reproductibilité numérique. Cependant, d'avantage de comparaisons peuvent être appliquées sur d'autres méthodes existantes afin de valider la "meilleure" à utiliser. En effet, la faisabilité de ces tests est devenue plus facile après l'identification des sources de la non-reproductibilité numérique.

Les cas de test étudiés bénéficiaient d'un bon conditionnement, donc comme estimé, une précision de travail doublée a été suffisante pour atteindre la reproductibilité. Une étape importante serait de traiter des cas de test mal conditionné, afin d'analyser le niveau de précision requis puis comment et où l'appliquer.

Finalement, les corrections décrites ont remis en question certains choix d'implémentation du code qui étaient mal adaptés à de telles modifications. Pour cela, plusieurs aspects d'ingénierie logicielle peuvent être mises en place pour faciliter l'intégration de nos propositions dans le code complet.

Résumé de chaque chapitre

Chapitre 1. Les problèmes de reproductibilité des calculs à virgule flottante et dans la recherche scientifique

L'exigence de reproductibilité devient de plus en plus cruciale en Science. La reproductibilité est un terme utilisé dans des contextes différents, mais avec un sens commun : "L'obtention des mêmes résultats lors de la répétition d'une expérience". Dans ce chapitre, nous considérons deux contextes différents.

(i) **La reproductibilité numérique** concerne surtout les chercheurs en informatique car ce problème est dû aux particularités de l'arithmétique flottante. En effet, les distributions différentes d'un calcul parallèle peuvent produire des résultats numériques différents, ce qui est considéré comme un défaut du logiciel. Dans ce chapitre, nous expliquons les causes de ce problème et les motivations à le résoudre. Puisque l'arithmétique flottante est la source de la non-reproductibilité numérique, nous présentons ses principales particularités et leurs conséquences sur les calculs parallèles. En effet, l'arithmétique flottante a tendance à générer des erreurs d'arrondi lors de ses calculs. En conséquence même l'addition de nombres flottants est non-associative. Ces erreurs sont à l'origine de deux causes qui mènent à la non-reproductibilité numérique. La première est

que pour une exécution parallèle l'ordre des calculs est non-déterministe, et donc pour des opérations flottantes chaque ordre peut produire des résultats différents. La deuxième est que la propagation des erreurs d'arrondi au cours des opérations diffère quand le nombre de processeurs change car les calculs partiels sont différents, ce qui aussi peut produire aussi des résultats différents. Ce travail s'intéresse surtout à la reproductibilité des simulations informatiques. Pour cela nous rappelons leurs principes et nous introduisons la façon d'analyser et de récupérer leur reproductibilité.

(ii) **La reproductibilité de la recherche** concerne elle, les chercheurs des diverses disciplines scientifiques. Elle vise à valider le développement de la science elle-même en améliorant la façon dont les nouveaux résultats de recherche sont publiés. Nous soulignons l'importance de cette approche et nous présentons quelques outils utilisés pour réaliser une recherche reproductible. Par exemple, une bonne documentation des étapes de développement est un aspect important pour produire un travail reproductible. En informatique, la plus importante demande pour reproduire un travail est la disponibilité du code source, une bonne documentation et des outils pour qu'un chercheur extérieur n'ait pas du mal à l'exécuter et reproduire les résultats initialement publiés. Malheureusement, ceci est parfois limité par les droits de propriété et aussi parce que les développeurs préfèrent passer leur temps à développer plutôt que de gérer ces processus de partage.

Chapitre 2. Comment obtenir la reproductibilité numérique

Ce chapitre décrit et analyse certaines méthodes qui peuvent améliorer la reproductibilité numérique. Il présente également l'application de ces méthodes dans des contextes différents qui souffrent de non-reproductibilité. Comme mentionné, améliorer la reproductibilité des résultats n'est pas équivalent à améliorer leur précision. Cependant, nous distinguons deux familles de méthodes qui ont comme objectif d'obtenir des résultats reproductibles.

— **Obtenir la reproductibilité en améliorant la précision des résultats.** En arithmétique flottante, chaque opération élémentaire introduit une

erreur d'arrondi qui est propagée au cours des calculs [48]. Par conséquence, l'erreur du résultat final peut finalement le rendre non significatif. Pour limiter cette perte de précision, les transformations sans erreur calculent les erreurs d'arrondi générées par les opérations flottantes élémentaires. Le principe de ces transformations est que pour une opération $op \in \{+, -, *\}$ entre deux nombres flottants $\hat{a}$ et $\hat{b}$, il existe deux nombres flottants $\hat{x}$ et $\hat{y}$ qui vérifient :

$$\hat{a}\ op\ \hat{b} = \hat{x} + \hat{y}. \tag{3}$$

Ici $\hat{x} = \circ(\hat{a}\ op\ \hat{b})$, *i.e.* $\hat{x}$ est la partie arrondie du résultat, et $\hat{y}$ est l'erreur d'arrondi générée qui vérifie $|\hat{y}| \leq \frac{1}{2} ulp(\hat{x})$.
La méthode de compensation utilise les transformations précédentes pour accumuler tout au long du calcul les erreurs d'arrondi dans un terme d'erreur et finalement compenser le résultat final avec ce terme.

Un autre moyen d'améliorer la précision est d'utiliser, par exemple, les nombres double-double. Un nombre double-double x est représenté par une paire (x_h, x_l) de flottants 64 bits. Les opérations double-double calculent en deux fois plus que la precision de travail [43]. Plusieurs travaux ont comparés l'utilisation de la compensation avec les calculs en double-double, et il apparait que ces derniers engendrent un sur-coût plus important que la compensation, [65], [55], [23].

— **Obtenir la reproductibilité indépendamment de la précision.** Ces techniques éliminent les inconvénients du parallélisme qui conduisent à la non-reproductibilité numérique. Comme mentionné précédemment, il s'agit de la propagation des erreurs d'arrondi qui diffère quand le nombre de processeurs change et parce que la réduction parallèle est non-déterministe. Nous listons comme exemple trois techniques différentes.

  — Les algorithmes de sommation reproductible proposés par Demmel et Nguyen [15] calculent des pré-arrondi identiques des nombres flottants pour les exécutions séquentielles et les exécutions parallèles.

  — La transformation des nombres flottants en entiers garantit l'associativité de l'addition.

— Forcer un ordre spécifique de la réduction en parallèle qui élimine les conséquences de la non-associativité.

Chapitre 3. La propagation des erreurs d'arrondi dans une simulation par éléments finis

Dans ce chapitre, nous décrivons les principaux calculs de la méthode des élément finis qui apparaissent dans les étapes de construction et de résolution du système linéaire. Cette présentation est liée à l'implémentation de la résolution par éléments finis d'openTelemac. Ce chapitre vise à détailler les calculs d'openTelemac qui sont nécessaires pour expliquer les sources de la non-reproductibilité. Nous décrivons le processus éléments finis en séquentiel et en parallèle pour les modules Tomawac et Telemac-2D. Finalement nous identifions et détaillons les sources de leurs non-reproductibilité.

Une exécution parallèle nécessite une décomposition du domaine en sous-domaines qui seront chacun exploités par des processeurs différents. Une décomposition efficace devrait tenir compte d'une bonne répartition des éléments finis entre les processeurs et doit réduire au minimum le nombre de nœuds (sommets de l'élément) partagés par plusieurs sous-domaines qui ont besoin de communications entre les processeurs. En effet, il y a des nœuds situés à l'interface entre les sous-domaines et qui sont partagés entre plusieurs processeurs. Ils sont appelés des *nœuds d'interface* ou des *points d'interface*.
Chaque sous-domaine est responsable de la construction et de la résolution de son propre système d'éléments finis mais quand un nœud d'interface appartient à plusieurs sous-domaines, certaines communications seront nécessaires afin de construire ou de résoudre le système global.

Les principales sources de non-reproductibilité

— **L'assemblage des nœuds d'interface.** Dans une exécution parallèle, les nœuds d'interface sont assemblés (au sens de l'assemblage par éléments finis) pour calculer la valeur globale de ce nœud. Cette valeur assemblée doit être la même que la valeur obtenue en séquentiel mais cette égalité n'est pas réalisée à cause des erreurs d'arrondi. Cet assemblage est la principale différence entre les résolutions séquentielle et parallèles, et affecte la reproductibilité numérique. En effet, la propagation incontrôlée

des erreurs d'arrondi produit des résultats différents lorsque le nombre de sous-domaines change. En effet, un nœud interne peut devenir un nœud d'interface dans une décomposition donnée, mais pas dans une autre.

— **Le produit scalaire.** Dans une simulation parallèle, le produit scalaire du domaine entier est calculé partiellement par chaque sous-domaine, puis il est sommé sur tous les sous-domaines par une communication collective. Cette réduction dynamique, produite par MPI, a un ordre non-déterministe. Donc pour les mêmes entrées, les résultats peuvent être différents. Cela conduit à la non-reproductibilité pour 2 exécutions successives, même si le nombre de processeurs est fixé. En plus, quand le nombre de sous-domaines varie, la propagation de l'erreur varie (les produits scalaires partiels varient), ce qui rend le résultat final différent aussi.

Chapitre 4. Vers des simulations reproductibles dans openTelemac

Dans ce chapitre, nous définissons comment modifier les calculs pour obtenir la reproductibilité numérique de deux cas de test dans Tomawac et Telemac-2D.

— **La reproductibilité de Tomawac.** Dans ce cas, nous comparons trois méthodes : la compensation, l'algorithme de somme reproductible de Demmel et Nguyen et la conversion en entier. Ces trois méthodes sont testées sur le cas de test *Nice* où la source de non reproductibilité est uniquement l'assemblage éléments finis et son complément, l'assemblage aux points d'interface en cas d'exécution parallèle.
Dans ce contexte, nous pouvons résumer leur faisabilité comme suit. La solution compensée fournit des résultats plus précis sans d'importants sur-coûts de calcul. La conversion en entier n'introduit pas non plus un grand sur-coût mais cette solution risque d'introduire une perte de précision. L'algorithme de somme reproductible de Demmel et Nguyen est en général efficace. Cependant dans ce contexte, il introduit un sur-coût important à cause des communications supplémentaires nécessaires.

— **La reproductibilité de Telemac-2D.** Dans ce cas plus complexe, nous appliquons la compensation qui est apparue comme étant la plus efficace

dans le cas précédent.

Dans ce module, la première source de non-reproductibilité est la propagation non-déterministe des erreurs d'arrondi générées pendant les calculs des nœuds d'interface. Nous rappelons que cette étape est implicitement présente dans plusieurs parties du calcul : phases de construction et de résolution du système linéaire. Pour corriger les valeurs, il faut calculer, stocker et propager ces erreurs, pour enfin les utiliser pour compenser la valeur calculée après chaque assemblage des nœuds d'interface. Ces corrections sont appliquées sur les simulations séquentielles et parallèles pour obtenir la reproductibilité attendue.

La seconde source est le calcul du produit scalaire global dans les itérations du gradient conjugué (étape de résolution). Il est corrigé en appliquant un produit scalaire compensé qui produit des résultats comme s'ils étaient calculés en deux fois la précision de travail.

Le cas de test considéré, *gouttedo*, semble être bien conditionné avec des erreurs relatives entre les résultats séquentiels et ceux parallèles variant entre 10^{-15} et 10^{-13}. Cela nous permet d'estimer qu'une double précision de travail suffit pour calculer les résultats à la précision de l'arrondi de représentation. Ce qui conduit à des simulations reproductibles avec un seul niveau de compensation. Cette approche est raisonnable en terme de sur-coût du temps d'exécution.

Aucun sur-coût significatif est mesuré pour l'ensemble de la simulation reproductible par rapport à l'original, à cause de la domination du coût du processus de lecture/écriture des données. Par contre, le cœur du calcul prend environ deux fois plus de temps dans la version reproductible.

Chapitre 5. Les aspects de l'implémentation

Après avoir obtenu la reproductibilité des cas présentés, l'objectif est d'obtenir une reproductibilité complète, *i.e.* dans tous les modules d'openTelemac. La technique de compensation doit être intégrée dans tous les calculs qui diffèrent entre les exécutions séquentielle et parallèle. Pour faciliter cette tâche, ce chapitre vise à être un complément technique utile qui explique les aspects de cette implémentation.

Nous décrivons la méthodologie de suivi des calculs du problème concerné pour identifier les sources de la non-reproductibilité. Cette étape permet de détailler où les erreurs d'arrondi diffèrent entre simulations séquentielle et parallèles.
OpenTelemac dépend de sa bibliothèque d'éléments finis BIEF. Celle-ci inclut plusieurs sous-programmes en FORTRAN 90 qui fournissent la structure de données, les phases de construction et de résolution de la simulation. Presque toutes nos modifications ont été limitées aux sous-programmes de cette bibliothèque. Nous expliquons quatre types de modifications sur : i) la structure des données, ii) les opérations algébriques et les phases iii) de construction et iv) de résolution éléments finis.

Les modifications sont transparentes pour les utilisateurs. Ceux-ci choisissent uniquement s'ils veulent lancer un calcul original ou reproductible en modifiant un mot clé du fichier de cas de test "ASSEMBLAGE EN ELEMENTS FINIS" (ou "FINITE ELEMENT ASSEMBLY" en anglais). Ce mot correspond à la variable Fortran `MODASS` et prend respectivement les valeurs 1, 2 et 3 pour le mode original, entier et compensé.

Il était inévitable de manipuler trois composantes d'openTelemac : la bibliothèque BIEF, la bibliothèque parallèle et le module Telemac-2D qui comprennent respectivement 493, 46 et 192 sous-programmes. Les modifications pour obtenir la reproductibilité ont été limitées à environ 30 sous-programmes qui appartiennent principalement à BIEF.

Nous pensons que détailler l'implémentation des modifications est important pour la continuité de ce travail, bien que ce chapitre nécessite un peu de connaissance du code openTelemac. Il contribue aussi à la reproductibilité de notre démarche et de nos contributions. L'intégration de ces modifications est toujours en cours. Il est prévu que ces modifications de code soient disponibles dans la prochaine version diffusée d'openTelemac.

Contents

OpenTelemac Notations

In this document	In openTelemac code	Description
nel	NELEM	number of elements in the mesh
el	IELEM	the number of one element
ndp	NDP	number of node in on element
idp	IDP	the local number of one element vertex
np	NPOIN	number of nodes in the mesh
IKLE	IKLE	the connectivity table
A_1, A_2, A_3	AM1,AM2,AM3	the linear system matrices
C_1, C_2, C_3	CV1,CV2,CV3	the linear system second members

Chapter 1

Introduction

Non-reproducibility issue

Most of scientific domains are becoming computational today, as for instance in healthcare, biomedical and biosciences, climate and environmental changes, multimedia processing, design and manufacturing of advanced materials, geology, astronomy, chemistry, physics, and of course financial systems [60]. High performance computing of large and complex numerical simulations are so performed. Parallel processing that distributes these large amount of computations to several computing units is widely used to reach efficient running times. But this performance gain may leads to some numerical reproducibility failures which are revealed as an application drawback.
The failure of numerical reproducibility is due to the finite precision of computer arithmetic. The floating-point arithmetic specified by the IEEE-754 standard is the most common way of dealing with real numbers on a computer. In this standard the rounding error of each basic operation ($+, -, \times, /, \sqrt{\ }$) satisfies one upper bound that corresponds to the best possible accuracy for the available machine precision. Nevertheless, a sequence of these operations propagates each rounding error operation over the whole computation until the error in the final result may blur it, and even changes it into a non-significant value. Rounding error leads to non-associative floating-point additions. Parallel applications often implement a reduction operation, which consists to apply a chosen operation ($+$, $\times$, max, $\cdots$) over all the partial results returned by the computing units in order to compute a global result. Nevertheless, this reduction computes in a non-deterministic order that may corrupt the result of a parallel computation from one run to another.

Numerical reproducibility is to get the same result when running a computation several times and even with different numbers of computing units. Such a need is motivated to facilitate the debug, the test and the validation of scientific codes. When non-reproducible executions yield different results for the same inputs eliminate bugs became a fussy task: does these different results come from rounding errors or from another bug in the program?

In this work, we are interested in a HPC simulation in computational fluid dynamics in 1D-2D-3D, mostly in 2D applications. Its resolution uses the finite element method. We study the openTelemac suite which is one industrial scaled software [51] originally introduced by the Laboratoire National d'Hydraulique et Environnement (LNHE), a part of the R&D group of Électricité de France. It is now an integrated set of open source Fortran 90 modules, with more than 300,000 lines of code issue from a 20 years of international collaboration. More than 4000 users and a hundred of developers (engineers, scientists, $\cdots$) are registered on its official web site.

OpenTelemac simulations suffer from non-reproducibility. For instance, the 2D-simulation of the *Malpasset* dam break (433 dead people and huge damage in 1959) is performed with a finite element resolution of the Saint-Venant equations. This test case is solved by the Telemac-2D module; unknowns are the water depth (H) and its velocity (U,V). The parallel resolution uses a subdomain decomposition of a triangular element mesh of 26000 elements and 53000 nodes.
Table 1.1 exhibits the non-reproducibility at some randomly chosen points when varying the number of computing units between: 1, 64 and 128 processors. Most of the parallel results share nothing more than the order of magnitude compared to the sequential result.

TABLE 1.1: Reproducibility failure of the *Malpasset* test case

	The sequential run	a 64 procs run	a 128 procs run
depth H	0.3500122E-01	0.2748817E-01	0.1327634E-01
velocity U	0.4029747E-02	0.4935279E-02	0.4512116E-02
velocity V	0.7570773E-02	0.3422730E-02	0.7545233E-02

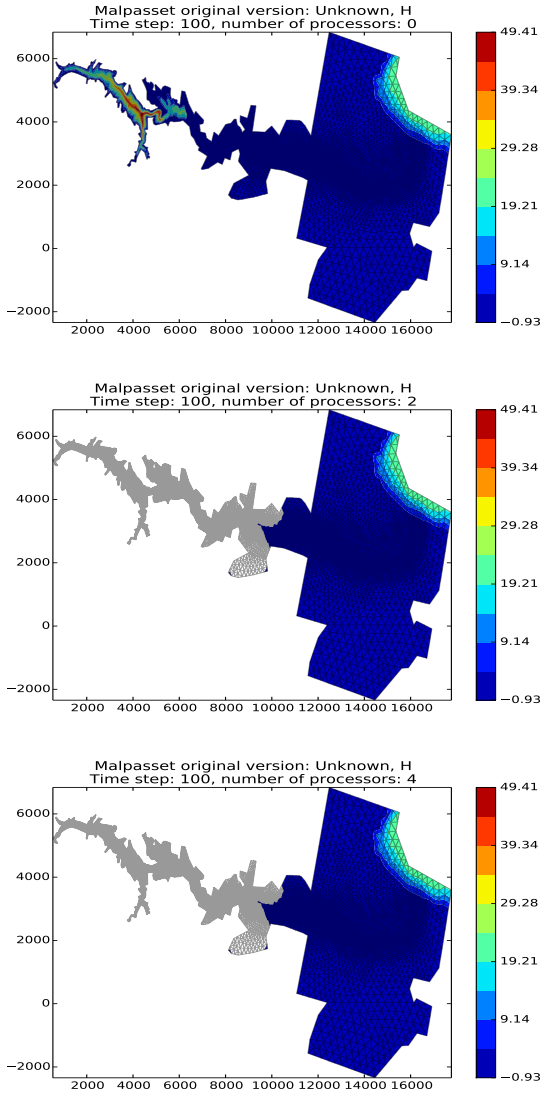
Figure 1.1 exhibits the non-reproducible values of the computed water

depth between the sequential execution and the parallel ones after 100 simulation time steps for the whole domain.

— The first plot shows the water depth values computed by the sequential simulation.

— The second and the third plots are related to the parallel results with 2 and 4 processors, respectively. Gray spots exhibit the mesh node values that are different from the sequential ones.

FIGURE 1.1: *Malpasset*: white spots are non-reproducible water depth values. The sequential run (left) compared to a 2 processors run (middle) and a 4 processors run (right). Time steps: 100×0.2 sec.



Numerical reproducibility failures have been reported in other dynamic fluid computations [55] and also in various other application domains of HPC simulation like in energy [68], dynamic weather science [23] or dynamic molecular [63] for instance. Before entering in more details, we emphasized two points related to any numerical reproducibility study.

— In some works, reproducibility is not recovered but only improved. The first case means obtaining bitwise identical results for parallel computation when the number of processors varies. The second one means that the number of different bits are decreasing between the parallel results as in [55], [68], [23].

— When results become reproducible, they do not necessary turn accurate, since a reproducible result may be far away from the exact one.

It exists several techniques to yield numerical reproducibility. They can be presented in two groups.

— ***Improving the result accuracy.*** These techniques recover reproducibility by computing one accurately rounded value of the result in both the sequential and the parallel executions. This kind of approach is for instance provided by error-free transformations, compensation or expansions. Compensation techniques improve the result accuracy by rewriting the algorithms to introduce targeted error-free transformations. *A posteriori* error analysis proves that the compensated result is as accurate as if the original algorithm is performed in twice the working precision. Expansion techniques simulate more computing precision using error-free transformations for every arithmetic operation of the computation. Let us mentioned that both expansions and compensated techniques can be iterated to double, triple, $\cdots$, multiply k-times the computing precision.

— ***Eliminating the causes of the non-reproducibility.*** These techniques force the addition associativity which is the main culprit of the non-reproducibility issue. A first proposed way is to freeze the order of additions in the parallel reduction. It is also possible to convert floating-point numbers to integer values which have an associative addition. A recent solution is to achieve a pre-rounding step of the floating-point numbers designed

to be similar in both the sequential and the parallel executions, such that it eliminates the non-deterministic propagation of the rounding errors.

Contributions

The contributions of this work are detailed in the chapters 4, 5 and 6 and have also been published in [40], [41], [49]. This work aims to analyze the numerical non-reproducibility issue in openTelemac and to fix it with the more suitable way. The main steps are to identify the sources of this non-reproducibility (in the code), to then propose, implement and analyze efficient methods that eliminate this drawback.

OpenTelemac computes a finite element simulation. A linear system is built by recovering the finite element values to express them as nodal values. This process is named the finite element assembly. Parallelism is implemented thanks to domain decomposition where each sub-domain is managed (here) by one computing unit. This process introduces inner and interface nodes. The latter belong to a common boundary between several sub-domains and their computations are shared between several computing units. In order to provide a global result, sub-domains exchange their local node contributions to accumulate them (in a static order in openTelemac). This step is named the interface node assembly. It is one of the main significant differences between the sequential and the parallel resolutions and it mainly affects the numerical reproducibility. Indeed, uncontrolled propagation of rounding errors and different results are produced when the sub-domain number changes. One inner node may become an interface one in a given decomposition while remaining an inner node in another decomposition. To recover the reproducibility, all node computations must be corrected, *i.e.* both inner and interface ones. Another source of non-reproducibility is the computation of the dot products in the linear system resolution. In a parallel simulation, the dot product that corresponds to the whole domain is first partially computed by each sub-domain, then accumulated by a collective communication over all the sub-domains. In practice, the collective reduction is often dynamic and so produces non-deterministic orders. In this case, for the same inputs, the results may differ because of the non-associativity of the floating-point addition. This problem also occurs for successive runs for the same number of computing units.

We detail how we recover the reproducibility in two modules of openTelemac: Tomawac and Telemac-2D.

— **Tomawac.** This module is used to model wave propagation in coastal areas. In this module a diagonal system is built and solved. So the finite element assembly appears to be the only source of the non-reproducibility. Modifications are so limited to this step. We will compare three methods that allow us to recover the reproducibility. We analyze their implementation difficulty and their extra-cost. We evaluate compensation [48], Demmel and Nguyen reproducible sums [15] and integer conversion techniques [51]. In this case we can summarize that the compensated solution provides both the more efficient and the more accurate results. Demmel and Nguyen reproducible sums are penalized by a significant communication extra-cost compared to the others.

— **Telemac-2D.** This module is a 2D hydrodynamics module which solves the Saint Venant equations using the finite element method for a computational mesh of triangular elements. We also recover the numerical reproducibility of a test case using compensation techniques. In this module, a system linear is built by finite element assembly and several algebraic operations. Then it is solved with the conjugate gradient algorithm. Because of several optimizations these two phases are merged: the assembly procedure appears even in the solving phase. We identify the following sources of non-reproducibility.

  — In the building phase: the finite element assembly and the algebraic operations are not reproducible due to different rounding error propagations when the decomposition differs.

  — In the solving phase: the dot product produces different results due to the MPI dynamic reduction. The matrix-vector product is also a non-reproducible source, because it includes a finite element assembly due to the EBE storage of the system matrix.

  To reduce the rounding error propagation, we compute and store these errors during the finite element assembly. We update them over the algebraic operations and finally use them to compensate the computed value after every interface node assembly. This process is also applied in the solving phase to correct the matrix-vector product.

On another hand, the MPI dot product is corrected in sequential with the compensated algorithm Dot2 which computes an accurate compensated result. For the parallel execution, we implement pdot2 that starts to compute the dot products and the generated rounding errors locally in each sub-domain. Then these local pairs are exchanged and accumulated using compensated Sum2 in each sub-domain.
The compensation is applied in both the parallel and the sequential simulations to yield the expected reproducibility of these two cases. This approach is measured to be reasonable in term of running time extra-cost: compared to the original one it introduces no significant extra-cost to the complete reproducible simulation when the read/write data process are taken into account.

Outline

— **Chapter 1. Reproducibility in floating-point computation and in scientific research.** This chapter details the causes of the numerical reproducibility failure. It introduces the main features of floating-point arithmetic and detail the effects of this arithmetic to parallel computing. It explains why numerical reproducibility is required and the way to analyze it specially in computing simulations.
This chapter also discusses the importance of reproducibility in scientific research as a way of how research results have to be published to remain reproducible. This process aims to validate and facilitate the development of the researches.

— **Chapter 2. How to recover numerical reproducibility.** This chapter describes some methods that may enhance numerical reproducibility. A first category improves the result accuracy. Compensation and expansion methods belong to this set. A second one includes two different ways. In one hand some methods force the associativity of the addition, either by transforming the floating-point numbers into integer or by setting the order of the parallel reduction. Other methods achieve similar pre-rounding steps to the floating-point numbers in both the sequential and the parallel executions such that addition results do not depend anymore of the evaluation order.

This chapter also presents applications of these methods in different computation contexts which suffer from non-reproducible results.

— **Chapter 3. The rounding error propagation in finite element simulations.** This chapter describes the main steps of a finite element method. It is specially related to openTelemac's modules: Telemac-2D and Tomawac. We exhibit the non-reproducible components and then identify the sources of these failures. The first source is the finite element assembly when parallel domain decomposition is used. This process introduces different rounding error propagations between sequential and parallel runs. In Telemac-2D a second source is due to the computation of a global dot product which introduces a non-deterministic order of additions when the number of computing units varies.

— **Chapter 4. Toward reproducible simulations within openTelemac.** This chapter details the modifications to the openTelemac computation that we introduce to recover the reproducibility of the studied modules Tomawac and Telemac-2D.

Three methods are compared for Tomawac: compensation, transformation into integers and reproducible sums. We detail their implementation and analyze the corresponding running time extra-cost. We find in this context of finite element computation that the compensation is the more efficient way.

Consequently, this latter method is applied to the more complex Telemac-2D test case. We compensate in both the building and the solving phases of the linear system. These phases include several non-reproducible sources as, the finite element assembly, algebraic operations, and the conjugate gradient algorithm. Indeed, this latter computes non-reproducible dot products and matrix-vector products.

— **Chapter 5. How to implement reproducibility in openTelemac.** This chapter describes the methodology that we used to identify the non-reproducibility sources by tracking the computation sequence. Then it details implementation aspects of the modifications we introduced in openTelemac. There are four types of modifications with respect to the computing context: data structure, algebraic operations, building phase and solving phase. This chapter provides a technical point of view that

will be useful in the future to continue toward the full reproducibility of hydrodynamics simulation with all openTelemac modules.

Chapter 2

Reproducibility in floating-point computation and in scientific research

The reproducibility requirement becomes more and more crucial in Science. Reproducibility is a term used in various contexts but with a common sense: "Obtaining the same results when repeating an experiment". In this chapter, we consider two different contexts: i) the numerical reproducibility and ii) the reproducibility of research. The first one mainly concerns the computer science researchers, because this issue is due to floating-point arithmetic peculiarities. Indeed, different computing distributions of a parallel computation may yield different numerical results, which is considered as a software drawback.
Reproducibility of scientific researches concerns, or should concern, every scientist in various disciplines. It aims to validate scientific contribution by improving the way that new research results are published. Hence, reproducible research contributes to develop the science itself.
In Section 2.1 we present the main features of floating-point arithmetic and their effects when computing arithmetic sequences in parallel. We detail in Section 2.2 the motivations and the reasons of numerical reproducibility by explaining how the floating-point arithmetic becomes the main culprit of this issue. We also recall the principles of a computer simulation to introduce how we will be able to analyze and recover its reproducibility, which is the motivation of this thesis. Finally, Section 2.3 is dedicated to the reproducibility of research, where we highlight its importance and several tools which are useful

succeed a reproducible research.

2.1 Main features of floating-point arithmetic

The computer memory is limited so it cannot store infinite precision numbers. Hence floating-point (FP) numbers approximate real numbers. The most common representation for floating-point numbers with computer is the IEEE-754 norm standardized in 1985 [31] and revised in 2008 [32]. This standard defines the floating-point formats and the behavior of their arithmetic operations. It also defines the format conversion rules, some special values, the rounding modes and the accuracy of the basic arithmetic operations.
The IEEE-754 standard aims to obtain predictable and portable programs which produce identical results when running on different machines.

2.1.1 Floating-point numbers representation

Floating-point representation is based on scientific notation. Basically a floating-point number x is a signed product of a fixed size integer and a rational number. It is represented as:

$$x = (-1)^s \cdot m \cdot \beta^e, \tag{2.1}$$

where $s \in \{0, 1\}$. The mantissa m is a string of integer digits which depend on the radix $\beta > 1$ ($0 \leq m_i < \beta$). The exponent e, represented by w bits, acts as a scaling factor for the floating-point number x. The number of mantissa digits is the precision of the floating-point number and is denoted t in this manuscript.

Relation (2.1) leads to several possible representations of x. So a set of extra rules, named normalization, are introduced to have only one representation for a number in a given precision.
In binary radix ($\beta = 2$), the digits $m_i \in \{0, 1\}$, therefore mantissas are normalized by setting the first digit m_0 to 1. This way to save one bit in encoding is named the implicit bit convention or the hidden bit convention. This bit is stored implicitly. Figure 2.1 details each component for the binary case.

FIGURE 2.1: Binary FP representation

sign, 1 bit	exponent, w bits	mantissa, t bits
$s \in \{0, 1\}$	$e_{min} \leq e \leq e_{max} \in \mathbb{Z}$	$m = 1.m_1 m_2 \dots m_{t-1}\ ;\ 0 \leq m_i < 2$

If no normalization occurs, a number is called a subnormal or denormalized number. It is any non-zero number with a magnitude smaller than the smallest normal number. The significand of a subnormal number always has the form $m = 0.m_1 m_2 \dots m_{t-1}$. This leads to a gradual loss of precision when the result of an arithmetic operation is not exactly zero but is too close to zero to be represented by a normalized number. If $x, y \in \mathbb{F}$ are different but very close numbers, the computed value of $x - y$ equals 0 if normalized number are only represented, while if subnormal numbers are available, $x - y \neq 0$ and is a subnormal number.

IEEE-754 formats

The IEEE-754 standard defines the floating-point representation formats. These formats have evolved between the standardized version in 1985 [31] and the 2008 revision [32]. The latter introduced, for instance, binary and decimal floating-point numbers. It also defines two kinds of formats that are named interchange formats and extended or extendable precision formats. For details we refer to [48], for instance.

The standard defines the basic formats in binary, *binary32* and *binary64*. They correspond to the simple and double precision of the 1985 version and are the mostly available and used in practice. These floating-point numbers are respectively represented with 32 and 64 bits. We also mention the extended format *binary64* which is introduced since the 1985 version to take it into account the 80-bit registers in the x86 processors. These formats are summarized in Table 2.1

2.1.2 Rounding function

The rounding function, denoted $\circ$, is applied to floating-point numbers and their operations. It is needed to represent one number or one operation result which can not be exactly written as a floating-point number, e.g. the

TABLE 2.1: Parameters of three IEEE-754 binary formats.

Parameter	Format		
	binary32	*binary64*	extended *binary64*
Precision, t	24	53	≥ 64
e_{max}	+127	+1023	$\geq +16383$
e_{min}	-126	-1022	≤ -16382
Exponent width, w	8	11	≥ 15
Format width in bits	32	64	≥ 79

constants π or $1/10$, the latter is written in binary as an infinite development $0.0\underline{0011}00110011\cdots$. The rounding operation replaces these real values by an approximating floating-point value. For a non-representable number $x \in \mathbb{R}$, $\circ(x) = \hat{x}$ denotes the floating-point number $\hat{x} \in \mathbb{F}$ resulting from the rounding. Of course $\hat{x} = \hat{x}$ when $\hat{x} \in \mathbb{F}$

Rounding modes

The IEEE-754 standard defines how any numerical value is rounded to a floating-point number by introducing several rounding modes. In the following we enumerate the four rounding modes, and illustrate them in Figure 2.2. The first mode is the default one and the others are called directed rounding.

— round-to-nearest:
- $RN(x) = \hat{x} \in \mathbb{F}$, where $|\hat{x} - x|$ is minimal and $\hat{x}$ is unique.
- If the number falls midway between $\hat{x}_r$ and $\hat{x}_l$,
where $|\hat{x}_r - x|$=$|\hat{x}_l - x|$, the so called *round tie to even* rule applies the returned number is the one with the even integral significant. This rule is the default one [1].

— round down or round towards $-\infty$:
- $RD(x) = max_{\hat{x}\in\mathbb{F}}(\hat{x} \leq x)$, where $\hat{x}$ is the largest floating-point number less than or equal to x.

— round up or or round towards $+\infty$:
- $RU(x) = min_{\hat{x}\in\mathbb{F}}(\hat{x} \geq x)$, where $\hat{x}$ is the smallest floating-point number greater than or equal to x.

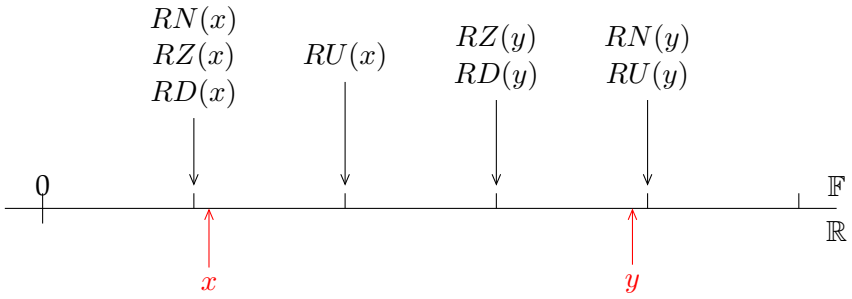
— round toward 0:
- $RZ(x) = RD(x)$ if $x \geq 0$,

1. The 2008 revision of the IEEE-754 also introduces a Round Tie to Away mode that delivers the largest magnitude value.

- $RZ(x) = RU(x)$ if $x \leq 0$,

where $\hat{x}$ is the closest floating-point number to x that is no greater in magnitude than x.

FIGURE 2.2: The rounding modes for $x, y > 0$.



The IEEE-754 standard forces the *correct* rounding for the five basic operations (addition, subtraction, multiplication, division and square root). "Correct rounding" means that the returned result is computed as an infinitely precise result, that is then rounded to the floating-point number according to the chosen rounding mode. At each rounding, you *a priori* lose some accuracy which corresponds to the rounding error. This rounding error is bounded by the arithmetic precision. It is classic to denote **u** this working precision that verifies $\mathbf{u} = 2^{-t}$ for the RN rounding mode and $\mathbf{u} = 2^{1-t}$ for the other modes.

2.1.3 Relative errors and the ulp function

Whatever we work about accuracy or reproducibility of floating-point computation, we are interested in the rounding errors, or more generally in approximation errors. Let $\hat{x} \in \mathbb{F}$ be an approximated value of $x \in \mathbb{R}$. We distinguish two errors:

- the absolute error:

$$Err_{abs} = |\hat{x} - x|, \tag{2.2}$$

- the relative error:

$$Err_{rel} = \frac{|\hat{x} - x|}{|x|}, \quad \text{if} \quad x \neq 0. \tag{2.3}$$

When a result $\hat{x}$ is compared to the exact value x, Relation (2.3) measures the accuracy of the result $\hat{x}$. In real-life applications as in industrial ones, the exact result is unknown. So we have to compare the value $\hat{x}$ to a reference one. For the reproducibility issue in parallel computations, a common practice is to

consider that the sequential computation yields **the** reference result. Of course such choice is arbitrary since all these results remain approximations as long as no accuracy validation has been provided.
The major problem of floating-point computation is the rounding error propagation which occurs within a sequence of computations. Although one isolated operation returns the best possible result as the generated error is bounded by $\mathbf{u}$ or $\mathbf{u}/2$ with respect to the rounding mode. We have:

— for directed roundings: $Err_{rel} < \mathbf{u}$,

— for nearest rounding: $Err_{rel} \leq \mathbf{u}/2$.

A sequence of calculations may lead to significant errors due to the accumulation of every single rounding error. It is well known that a final result of several operations may be far from the exact value, see [48] for details and numerous entries on the subject.

The term unit in the last place (ulp) refers to the weight of the last bit of the mantissa. Its definition may vary from one source to another. A review of these definitions is proposed in [47]. The ulp of a floating-point number $\hat{x}$ is defined by the Relation (2.4):

$$ulp(\hat{x}) = \begin{cases} 2^{e-t}, & \text{if round-to-nearest,} \\ 2^{e+1-t}, & \text{otherwise.} \end{cases} \tag{2.4}$$

If $\hat{x} > 0$, then ulp($\hat{x}$) is the gap between $\hat{x}$ and the next larger floating point number. If $\hat{x} < 0$, ulp($\hat{x}$) is the gap between $\hat{x}$ and the next smaller floating point number (larger in absolute value). Ulps are of interest for measuring and describing the computation accuracy.

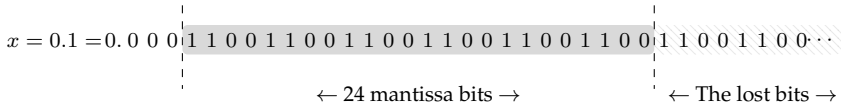
2.1.4 Sources of errors in floating-point arithmetic

The floating-point arithmetic may lead to incorrect results because of the rounding errors. The worst cases appear in the absorption and cancellation phenomena. Other errors may come from overflows or underflow cases. All these behaviors sometimes create catastrophic bugs as reported in [66], [2]. We define these terms in the following.

Rounding errors

These errors are introduced by the mostly unavoidable roundings presented in the previous section. They represent the main source of errors in floating-point computation.
For instance, the real number $x = 0.1$ has an infinite floating-point representation. Figure 2.3 shows how the rounding of x will lose bits, that become the rounding error within the floating-point representation of x.

FIGURE 2.3: The binary 32 representation of the real number $x = 0.1$.



Overflow and underflow

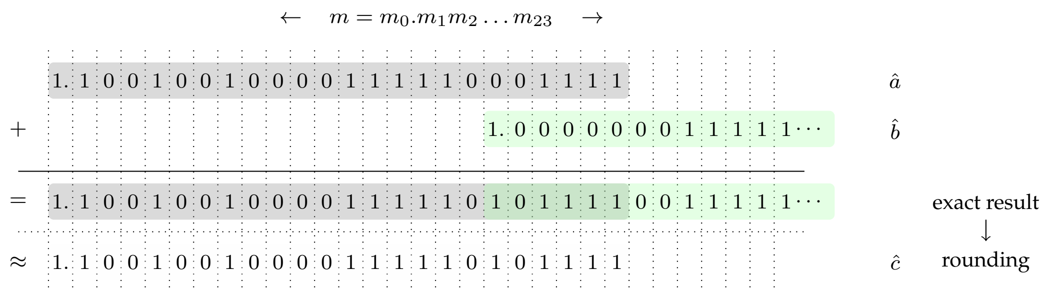
Like all finite representation formats of numbers, it is possible to code numbers only between a minimum and a maximum limit. If one operation produces a value larger than the maximum representable floating-point number, then the result will be the special value $\inf$. The inability to compute beyond the maximum value of the format is named overflow. Note that once one operation returns a special value, all other operations using this value as a parameter will produce a special value: $\inf$ value, or NaN for example in the case of multiplying by zero.
Conversely, underflow occurs when a value is the smaller than the smallest representable floating-point number in the used format. In this case, the selected rounding mode sets the corresponding consequences. One of the risk caused by this problem is the possibility that the value is rounded to zero. That may produce a cascade of cancellations (through a suite of operations) or the return of the special value $\inf$ in the context of a division.

Absorption and cancellation

Absorption is due to the rounding of one addition between two floating-point numbers having very different orders of magnitudes. Let $\hat{a}$ and $\hat{b}$ be two floating-point numbers such that $\hat{a}$ is much bigger than $\hat{b}$. A part of $\hat{b}$ will be lost in the result of the floating-point addition $\circ(\hat{a} + \hat{b})$. This phenomenon is

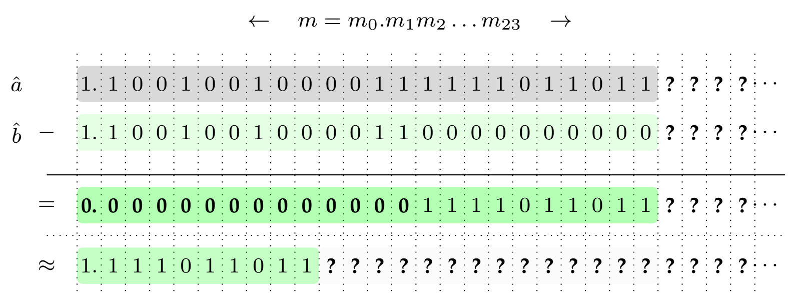
illustrated in Figure 2.4. It is named a catastrophic absorption when all the bits of $\hat{b}$ are lost during the addition. In this case, the result equals to $\hat{a}$ and $\hat{b}$ has been fully absorbed by $\hat{a}$.

FIGURE 2.4: Absorption when adding $\hat{a}$ to $\hat{b}$ where $\hat{a} \gg \hat{b}$. A part of $\hat{b}$ mantissa is lost.



Cancellation occurs during the subtraction of two very close numbers in absolute value (or the addition of same kind of values but of opposite signs). In each operation the result is normalized by adding zeros in the mantissa as the most significant bits are canceled. The result can be very different from the exact one and it may cause a huge relative error. This phenomenon is illustrated in Figure 2.5. We talk about catastrophic cancellation when it remains only a few significant digits in the result.

FIGURE 2.5: Cancellation of the subtraction of two very close and uncertain numbers $\hat{a}$ and $\hat{b}$. The re-normalization of the result adds uncertain zeros to the right of the mantissa because the bits on the left canceled.



These two phenomena are not severe when operands are exact. In this case, IEEE-754 correct rounding applies and so rounding errors are bounded by the precision **u**. On the contrary, cancellation catastrophic for close operands that suffer from uncertainty. This uncertainty may come from previous operations

and the worst case is the absorption one. The catastrophic loss of this sequence of absorption then cancellation is illustrated by the following example that also exhibits the non-associativity of floating-point addition.

Non-associativity in floating-point arithmetic

The floating-point addition is not associative because of rounding errors. So changing the addition order may produce different results $\circ(\circ(a+b)+c) \neq \circ(a+\circ(b+c))$. This phenomena is a consequence of the limited precision and of the range of the IEEE floating-point representation. Hence, depending on the computation order, the propagation of rounding errors differs to yield to different results.
For instance, the following computation illustrates the non-associativity problem:

$$\circ(\circ(-1+1)+\mathbf{u}) = \mathbf{u}, \quad \text{while} \quad \circ(\circ(1+\mathbf{u})-1) = 0.$$

In the first evaluation order, the first two numbers cancel each other out, such that $\mathbf{u}$ is added with 0 to result $\mathbf{u}$. Whereas in the second order, $\mathbf{u}$ is absorbed by 1, where $1 \gg \mathbf{u}$ and it remains the operation $1-1$ that exactly returns 0.

The next example illustrates that floating-point operations can overflow or not, depending on the order of evaluation. As in Fortran 90, we denote MAX_DOUBLE the maximum value that is representable by a double precision floating-point number. The mathematically equal expressions

$$(MAX\_DOUBLE - MAX\_DOUBLE) + MAX\_DOUBLE, \text{ and}$$
$$(MAX\_DOUBLE + MAX\_DOUBLE) - MAX\_DOUBLE,$$

are differently processed: there is no overflow as well as no rounding error in evaluating the first expression, but an overflow clearly occurs in the second one.
We detail in the next section how these limitations of floating-point arithmetic generate non-reproducible computations.

2.2 Numerical reproducibility

High performance computing (HPC) for large scale numerical simulations relies on parallelism to distribute the computations on several computing units (or processors), which reduces the running time of computations.
In this section we describe how parallel executions produce different computation orders either when the number of computing units changes or not. That may produce different results because of the finite precision of computer arithmetic (Section 2.1).

Hence numerical reproducibility aims to get the same result when a scientific computation is run several times with different numbers of computing units. Numerical reproducibility is a requested feature for several reasons. Firstly, it facilitates the debugging and the testing of the code. Effectively, it is not obvious to fix a bug nor to test a code when the results differ from one run to another. Secondly, many simulations go through a validation and verification (V&V) process to obtain a legal agreement [5]. The verification process determines if the programming and the computational implementation of the conceptual model is correct. The validation process determines if the computational simulation agrees with physical reality. It demonstrates that the accuracy of the simulation results are close to what can be measured through physical experimentations. Hence numerical reproducibility is a requirement in the validation step.

Some methods are proposed to avoid this numerical non-reproducibility issue; we detail them in Chapter 3. To choose the more suitable method, we take into account the context of the problem and the corresponding extra-cost.

2.2.1 The parallelism effects on the numerical reproducibility

Parallelism at a glance

Parallelism consists on using simultaneously several processors, later say p, to solve a single problem. The problem to solve is distributed by a specified method toward these processors. HPC users may require weeks of run to solve one application. For that, parallel computation is increasingly used for its effectiveness in reducing the processing time. Ideally, using p processors

divides by p the time to solve the same problem with only one processor. In practice, such an optimal ratio rarely applies.

Concurrent programming languages, libraries, APIs and parallel programming models have been created for programming parallel computers. They are divided into classes based on the assumptions they make about the underlying memory architecture: shared memory (that will not be addressed here) and distributed memory. Shared memory programming languages communicate by manipulating shared memory variables, as in P-Threads [53] and OpenMP [12] which are the two most widely used. Distributed memory communication relies on message passing, as the most widely used MPI (Message Passing Interface) [45].
MPI is standardized and portable. The standard defines the syntax and the semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran. In practice the processors are not independent of each other, they need to communicate. For that, there are two types of communications.

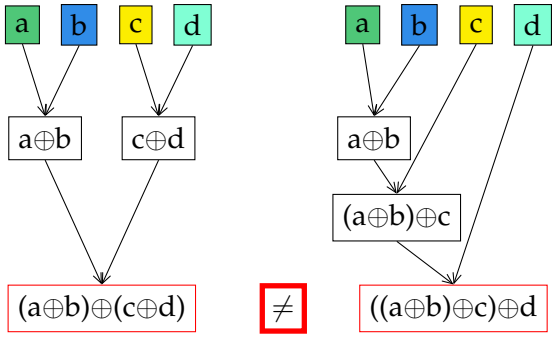
1. Point-to-point communication which occurs between two processors, the receptor and the transmitter.
2. Collective communication specifies various types of collective operations, such as all-to-all, all-to-one, one-to-all and others. Processors exchange their local results in order to obtain a global result. The last step toward the global result is usually named a reduction step. The all-to-one MPI_REDUCE or the all-to-all MPI_ALLREDUCE are most used collective operations. They compute the global value from the local ones corresponding to a required operation: it could be a summation, a maximum, a minimum or even a user-defined function.

In this work, the most important aspect of MPI is that it is non-deterministic by default: the arrival order of messages sent from two processors p_1 and p_2, to a third one p_3 is not defined. MPI only guarantees that two messages sent from p_1 to p_2 will arrive in the sent order. It is the programmer's responsibility to ensure that a computation is deterministic when this is required.

The effects on the numerical reproducibility

Parallel computation in our problem scope (finite element simulations) is introduced by domain decomposition where each computing unit solves its own sub-domain. The sub-domains compute their local contributions, exchange and summed them to obtain the global value of the whole domain. In this process two causes may lead to non-reproducible results.
First, parallel computation often introduces non-deterministic collective communications which leads to different order of the computations. Due to the non-associativity of the floating-point arithmetic, results are not reproducible. In this case, there is non-reproducibility for two successive runs even when the number of sub-domains does not change.
Figure 2.6 illustrates how a reduction differs for the same inputs a, b, c, d and the same operation $a+b+c+d$, because of the different orders of the addition.

FIGURE 2.6: A non-associative parallel reduction



Secondly, when the number of sub-domain varies, the computation of the local contributions differs because of the different propagations of rounding errors. In this case, even if the global value is summed in a static order, this will differ when the number of sub-domain changes (as it is the case in open-Telemac in Chapter 4). However, in this case successive runs with a fixed number of sub-domains remain reproducible.

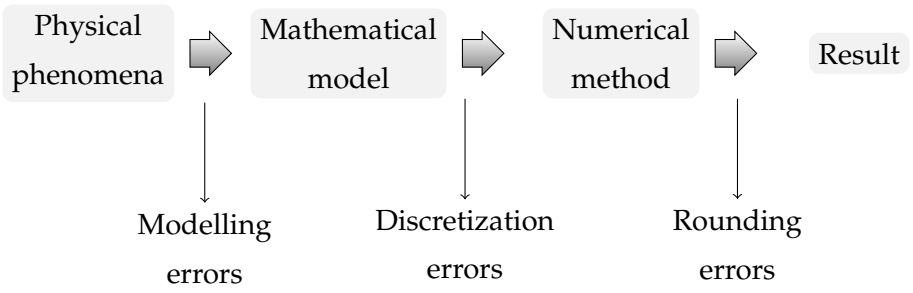
2.2.2 Non-reproducibility in numerical simulations

Computer simulation [69] is the mainstream technique of representing the real world by a computer program. It is a useful part of mathematical modeling of many natural systems, as in physics, astrophysics, climatology, chemistry and biology, human systems in economics, psychology, social science,

and engineering. Computer simulation is the main way to provide predictive information for complex phenomena. However accurate predictions are hard to be provided, since several approximations occur before getting the desired results [8]. As illustrated in Figure 2.7, these approximations appear in several steps. It is difficult to describe the true physics, so inevitable approximations occur during the building of its mathematical model. In the discretization of the problem, many approximations happen because of the transformation of the continuous functions, models and equations into discrete counterparts. Finally, approximations that occur during the numerical method are the rounding errors already introduced in the beginning of this chapter. In this thesis, we focus on these approximations. Indeed, this kind of approximation causes two issues: the loss of accuracy and the non-reproducibility of the results. We work on the non-reproducibility of a hydrodynamics simulation aiming to identify the sources of this issue and to correct them.

A question arises. Why do we want, for instance, to reproduce a fluid dynamic phenomena which is very influenced by uncontrolled physical parameters like wind, temperature, friction *etc.* Effectively, reproducing the phenomenon itself is not always possible in real life because of these uncontrolled parameters. But this parameters are set in a computational simulation, so the non-reproducibility is a numerical issue which has to be corrected, at least for the already presented motivation: debug, validate, legal agreement.

FIGURE 2.7: Various errors via numerical simulation



As computing power increases towards exascale, more complex numerical simulations are performed in various domains. Many cases of non-reproducibility have been reported, mostly where parallel sums or dot products are computed. For examples, in the context of finite difference or finite volume calculations, the authors in [55] report a non-reproducibility issue causing by a parallel sum.

The same problem is identified in climate simulations [23] and in a power state estimation application [68]. In [17], a non-reproducible parallel dot product is reported in a program which simulates sheet metal forming based on a finite element method. Two reasons of the non-reproducibility are defined in a Monte Carlo code which simulates the time-dependent interaction of photons and matter [21]. The first is the non-deterministic order of operations. The second is that the particles may not get the same pseudo-random number stream, this latter not being a numerical issue. It is solved by giving each particle its own random number generator state, and seeding these states in a manner that is independent of the number of domains. Even for single precision, in a molecular dynamics simulation using GPU [63], a failure of reproducibility appears due to a global summation which calculates the sum of a large set of numbers with a high variance in magnitude. In Chapter 3, we will exhibit the mostly used solutions that are proposed to correct these numerical reproducibility issues.

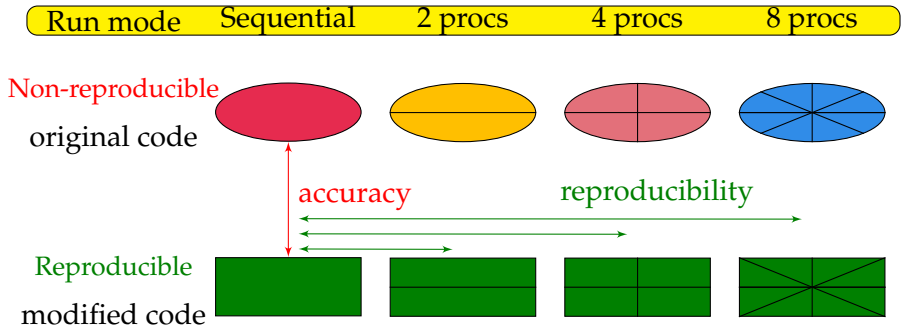
How to measure reproducibility in a parallel simulation

Figure 2.8 illustrates the main aspects we need to consider. The first row shows how the results of a code (in different colors) are not reproducible when the number of processors varies. To recover reproducibility we have to carefully identify the sources which produce the non-reproducibility and to then apply as few as possible modifications to limit their extra-cost. The main difficulty in this work was to identify these sources.
A convincing modified reproducible code should satisfied two criteria:
i) bit-wise *identical* result for every p-parallel run and for every $p \geq 1$, as showed in the second row in Figure 2.8 where the results share the same color.
ii) the reproducible results must be within a reasonable range of differences compared to the original sequential simulation ones. This measure is important for the code developers who, as already mentioned, trust their sequential results as a reference.
Figure 2.8 illustrates these two precedent measures.

- Reproducibility is measured as the relative error (2.3) between the modified sequential simulation and the parallel ones (in green).

– The measure of the accuracy is the relative error between the original sequential simulation and the modified one (in red).

FIGURE 2.8: The two measures of a convincing modified reproducible code



Finally the extra-cost of the modifications to recover reproducibility is an important measure, specially in simulation which computes at "real-time" the desired results. In fact, these modifications are not attractive if they reduce the parallelism benefits. Unfortunately, the accuracy of this measure is not reliable and even not reproducible. Indeed, the execution time can vary, for the same data input and the same execution environment. This uncertainty has many causes: spoiling events like background tasks, concurrent jobs or OS interrupts [42]. Modern processors include parts built to run with a non-deterministic behavior: instruction scheduler, branch predictor, cache management are examples of components implementing randomized algorithms. External conditions such as the room temperature also modify the clock frequency which invalidates second-based timings. In our work, as significant as possible measures will be obtained and presented in Chapter 5.

2.3 Reproducible research

The reproducibility of the research is the ability of a research study to be duplicated, either by the same researcher or by someone else. Reproducibility is one of the main principles of any scientific method. This ability is necessary for validating any research results. For instance, a bio-pharmaceutical company reported that it can not reproduce 47 of 53 "landmark" cancer papers [4]. This company may got different results because the experiments

were not performed in the same way as the original study. That opens the door to doubt about which result was correct: "the published or the repeated one". To prevent this kind of problems, scientists in several domains ask the question "How to produce a reproducible research". For guarantee the reproducibility of the research results, an independent researcher should be able to reproduce the experiment, under the same conditions, and achieves the same results. This procedure ensures that, the results are constructed on an accurate and reliable method. These independent researchers must not know the original paper, results, or its authors, eliminating chances for bias in testing.

Two often confusingly terms, reproducibility and replicability, are compared in [18],[3], [20], [67]. They are defined as: the reproducibility requires changes while replicability avoids them. In other words, reproducibility aims to provide the same scientific conclusions when experimental conditions may vary to some degree, while replicability is the ability to produce identical results under identical conditions. Indeed, these two ideas are requested in each research. The authors also give practical guidance for achieving both replicability and reproducibility in experimental science. They explain that a new research should be provided with sufficient information, specially technical ones in a "section of materials and methods", that allow experiments to be repeated and provide reproducible results. Moreover, researchers should cited the number of times that their experience was repeated. More the provided information is complete and accurate, reproducing the results will be easier .

In nature sciences, reproducing an experience can be difficult due to expensive laboratory equipment and complex process. Whereas in applied computer science, it is generally estimated that it should be sufficient to download the code sources and the data, compile it and run it. The authors in [11], explain that is not as easier. They published a study that includes 613 published works, and ask the questions: "Is the source code available, and does it build?". From the 613 published works, only 231 have the source code available and only 102 runs. View of the importance of the availability of the research source to obtain a reproducible work, this statistic is disappointing for the advancement of science.

There are many ignored aspects about methodology and tracking in the

computer science community, despite their importance for authors and reviewers as a principle in the conduct and the validation of research. For instance, a well documentation of the development steps is an important aspect to produce a reproducible work. It helps both authors and reviewers to understand and reminder all the steps of the concerned study to reproduce or to improve it. Moreover, the accessibility of a source code is essential for both reproducibility and development, but it is sometimes limited by the property right and sometimes developers prefer to spend their time to develop rather than handle the sharing process. For instance, when an author publishes figures and tables of an experimental result, it is better to publish also the generating scripts. That increases the confidence of reviewers and avoids to doubt about the dataset or about the meaning of this figure if the caption is not clear. At the author's side, it simplifies the modifications and corrections often suggested by reviewers since every figure is easily regenerated.

For the previous reasons, the authors of [62] motivate to use a combination of existing tools through the development work, that remains useful for a reproducible research. These tools are well-known, lightweight and open-source technologies. They provide reasonable reproducibility warranties without taking away too much flexibility from the users, by offering good code modification isolation. The authors propose to share the source code and data in a Git repository to simplify how reconstruct the experimentation setup. In Git, the results are in an experimentation history which are branched with their corresponding source code and dataset. Finally, they propose to write the papers in Org-mode to lead to a completely reproducible work. The Org-mode document contains, along with the text, all the analysis scripts and the raw data that can be inspected by reviewers.

In [58], the authors describe a suitable file for publication which stores together the dataset, the program code, and the presentation to allow the reproduction of published results. Obviously these approaches require an expertise on the used tools. It is also unclear if it would scale for multiple users working simultaneously, doing code modifications and experiments in parallel.

Obtaining or improving the reproducibility of results is one of the big challenges of the current research landscape. A survey was conducted in 2015

during the Euro-Par conference [30]. The questionnaire, which specifically targeted the parallel computing community, contained questions in four different categories: general questions on reproducibility, the current state of reproducibility, the reproducibility of the participants' own papers, and questions about the participants' familiarity with tools, software, or open-source software licenses used for reproducible research. The survey participants also think that the majority of the results presented in papers that they receive for review are unlikely to be reproducible. The survey also showed that scientists need to be better informed what the different open-source licenses actually mean and which licenses are allowed to be applied by their research institutions. Last, they found evidence that many scientists are not familiar with software for literate programming and with scientific workflows, which can potentially help to improve the reproducibility of articles. The survey revealed that the majority of the voters believe that the state of reproducibility needs to be improved in the domain of parallel and high performance computing, this context of reproducibility is discussed in the previous section.

The objective of producing reproducible research results is very important for the development of science. Indeed, it requires extra work and effort throughout the development, but with the right tools it becomes a useful work routine.

2.4 Conclusion

This chapter was dedicated to define of the main terms used in this thesis. We introduce the numerical reproducibility and its importance, specially in computer simulation. We exhibit the essential of IEEE-754 floating-point arithmetic and how this arithmetic could cause loss of accuracy. We have also introduced some concepts to analyze and measure this accuracy. We detailed the effects of parallelism and how it can introduce reproducibility failures. We also define the important measures which we use to illustrate our final reproducible results in Chapter 5.

Finally, we highlight the importance of the reproducible research issue where scientists must consider to apply an effort to support in the development of the science.

Chapter 3

How to recover numerical reproducibility

In this chapter we explain and analyze some methods that aim to improve the numerical reproducibility. We also present their application in different computing contexts which suffer from reproducibility failures. Indeed, as already mentioned, eliminating this numerical failure is an important issue for many industrial codes to debug them, to check their correctness and to validate their results. However, it is not equivalent to improve, nor even to validate the accuracy of their results since a reproducible result may be far away from the exact one.

Several methods aim to improve numerical reproducibility [34]. In this context, we choose to distinguish two types. In Section 3.1, we describe a first type of methods which compute an accurately rounded value of the result in both the sequential and the parallel executions. We focus on compensated and expansions techniques that both use error-free transformations. A second type, presented in Section 3.2, guarantees the numerical reproducibility without necessarily improving the accuracy of the computation. The idea here is to even force the associativity of the addition, or to achieve a similar prerounding of floating-point numbers in both the sequential and the parallel executions. For all these methods, we discuss the efficiency in terms of accuracy and run time extra-cost through the study of previous works, some being applications to reproducibility problems.

3.1 Reproducibility thanks to accuracy improvement

In floating-point arithmetic, each elementary operation may introduce a rounding error which is propagated over the computations. Consequently, the error in the final result may blur it, as even changes it into a non-significant value. To limit the loss of accuracy compensation, expansions or arbitrary precision libraries are useful existing methods. Applying them in a thoughtful way may yield to obtain the reproducibility of the results.

Compensation techniques improve the accuracy of one result performing light and very targeted extra computations (error-free transformations) at the working precision.
The multiple precision floating-point library (MPFR) [64] is a portable and reliable library written in C for arbitrary precision arithmetic on floating-point numbers. It extends most of the important features of the IEEE-754 standard (correct rounding, special values, exceptions) to binary floating-point numbers that encompass their own precision. It also provides some mathematical functions defined for the same kind of arbitrary precision numbers.
Expansion libraries stand between compensation and MPFR. It simulates FP arithmetic of k-times the working precision thanks to a systematic use of error-free transformations. Hence it provides at least as accuracy improvement as the compensation techniques and allows the users to go beyond: from double-double to quad-double number for instance.

3.1.1 Error-free transformations

The error-free transformations (EFT) compute the rounding errors generated by some elementary operations in floating point arithmetic. The principle of these transformations is that, for an elementary operation $op \in \{+, -, *\}$ of two floating-point numbers $\hat{a}$ and $\hat{b}$, it exists two floating-point numbers $\hat{x}$ and $\hat{y}$ that verify:

$$\hat{a}\ op\ \hat{b} = \hat{x} + \hat{y}. \tag{3.1}$$

Here $\hat{x} = \circ(\hat{a}\ op\ \hat{b})$, *i.e.* $\hat{x}$ is the rounded part of the result, and $\hat{y}$ is the generated rounding error that verifies $|\hat{y}| \leq \frac{1}{2} ulp(\hat{x})$. It can also be respectively named the high-order and low-order parts of the result.

In the following we present error-free transformations of addition and multiplication. These algorithms only rely on floating-point operations in the round-to-nearest (RN) mode.

Addition and subtraction

Algorithm 3.1 is one of the error-free transformations of the addition. It was introduced by Dekker in 1971 [14]. Fast2Sum computes the high x and low y parts of the sum of the two floating-point numbers a, b where $|a| \geq |b|$. It requires $3flop$ (floating-point operations) and if necessary, a preliminary test to ensure that $|a| \geq |b|$.

Algorithm 3.1 [x,y]=Fast2Sum(a,b), $|a| \geq |b|$

$x = RN(a + b)$
$z = RN(x - a)$
$y = RN(b - z)$

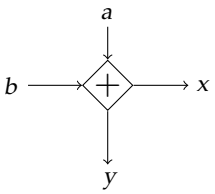


FIGURE 3.1: Fast2Sum symbol

Algorithm 3.2 was earlier mentioned by Moller in 1965 [46] and also proposed by Knuth [37] in 1969. Like Fast2Sum, 2Sum computes the high and the low parts of the sum but in $6flop$. Despite its $flop$ extra-cost compared to Fast2Sum, 2Sum may be preferable because it does not need a branching test nor require a preliminary knowledge of the operand order of magnitude [38]. Of course, these two algorithms produce the same result.

Algorithm 3.2 [x,y]=2Sum(a,b)

$x = RN(a + b)$
$a' = RN(x - b)$
$b' = RN(x - a')$
$\delta_a = RN(a - a')$
$\delta_b = RN(b - b')$
$y = RN(\delta_a + \delta_b)$

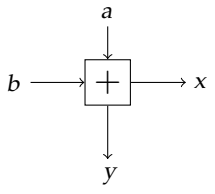


FIGURE 3.2: 2Sum symbol

In [48], Muller and *al.* note that on modern processors the penalty due to a wrong branch prediction when comparing a and b costs much more than

$3flop$ additional. On modern architectures, it is possible to perform independent operations in parallel, hence the depth of the dependency graph of the instructions of the algorithm is an important criterion. For instance, 2Sum performed two operations in parallel, therefore its depth is 5. Also Fast2Sum does not work for $\beta > 3$, whereas 2Sum works in any radix. This is of interest when using a decimal system even if the authors of [38] illustrate possible counterexamples for using Fast2Sum in radix 10.

Multiplication

The error-free transformation of the multiplication, 2Product, was introduced by Dekker in 1971 [14]. It starts by calling the Split algorithm proposed by Veltkamp [48]. Algorithm 3.3 splits the inputs a and b into their high-order and low-order parts, respectively a_h, b_h and a_l, b_l. The k value should verify $k < t$, where t is the precision of a. Thus the a_h significand fits in $t - k$ digits and the a_l significand fits in k digits, verifying that $a_h + a_l = a$ exactly. The value of k is respectively 16 and 27 for IEEE *binary32* and *binary64*.

Algorithm 3.3 *Veltkamp's split*: [a_h,a_l]=Split(a)

Require: $c = \beta^k + 1$

$\gamma = RN(c \times a)$

$\delta = RN(a - \gamma)$

$a_h = RN(\gamma + \delta)$

$a_l = RN(a - a_h)$

2Product returns the two floating-point numbers $x = RN(a \times b)$ and the generated error y. It requires $17flop$.

Algorithm 3.4 [x,y]=2Product(a,b)

$[a_h, a_l] = \text{Split}(a)$

$[b_h, b_l] = \text{Split}(b)$

$x = RN(a \times b)$

$t_1 = RN(-r_1 + RN(a_h \times b_h))$

$t_2 = RN(t_1 + RN(a_h \times b_l))$

$t_3 = RN(t2 + RN(a_l \times b_h))$

$y = RN(t_3 + RN(a_l \times b_l))$

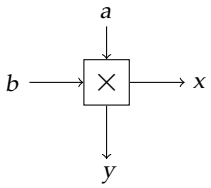


FIGURE 3.3: 2Product symbol

Another EFT for the product exists when a fused multiply and accumulate operator (FMA) is available [48]. FMA was introduced in 1990 on the IBM RS/6000 processor to facilitate correctly rounded software division and to make some calculations (especially dot products and polynomial evaluations) faster and more accurate. Algorithm 3.5 requires only 2 FMA to return both $RN(a\times)$ and its generated error.

Algorithm 3.5 [x,y]=2MultFMA(a,b)

$x = FMA(a, b, 0)$

$y = FMA(a, b, -x)$

Error-free transformations can be generalized to vectors or matrices for instance. In [57], [56] Rump and *al.* propose to transform a vector v where each element v_i is transformed exactly into a high-order part q_i and a low part r_i thanks to M. Algorithm (3.6) returns $T = \sum_{i=1}^{n} q_i$ which is the exact sum of the high-order parts q_i, and r which is the vector of the low-order parts r_i. The exact accumulation of the low-order parts r_i and T is the sum of the vector v: $T + \sum_{i=1}^{n} r_i = \sum_{i=1}^{n} v_i$.

Algorithm 3.6 [T,r] = ExtractVector(M,v)

Require: $M = 2^k, k \in \mathbb{Z}$ and $M \geq \sum_{i=1}^{n} |v_i|/(1 - 2n\mathbf{u})$

$T = 0$

for i = 1 **to** n **do**

$q_i = RN(RN(M + v_i) - M)$

$r_i = RN(v_i - q_i)$

$T = RN(T + q_i)$

end for

These error-free transformations are widely used to provide more accuracy in floating-point arithmetic. Many compensated algorithms have been developed using these transformations to compute accurate sum or dot product of floating-point vectors [50]. With several combined operations, the rounding errors are accumulated, then compensated in the final result to provide an accurate result.

3.1.2 Compensated algorithms

Compensated algorithms provide k times more accuracy while computing in the working precision **u**. The principle of such algorithms is to accumulate (in an error term) the errors of each elementary floating-point operation throughout the flow of computations. This error term represents the accumulation of all the committed rounding errors across the computation. It is finally compensated to the final rounded result to correct it.

Computing a twice more accurate result

Rump, Ogita and Oishi propose Algorithm 3.7 in 2005 [50]. This cascaded summation, Sum2, is a compensated algorithm which approximates the sum of a vector using Algorithm 3.2. It requires $7(n-1)flop$.

Algorithm 3.7 res = Sum2(a)

$s_1 = a_1, \sigma_1 = 0$
for i=2 **to** n **do**
  $[s_i, q_i]$ =2Sum(s_{i-1}, a_i)
  $\sigma_i = RN(\sigma_{i-1} + q_i)$
end for
$res = RN(s_n + \sigma_n)$

They also propose Algorithm 3.8, which is a compensated dot product, that uses Algorithms 3.2 and 3.4 to compute a twice more accurate result. Dot2 requires $25n - 7flop$, or $10n - 7flop$ if a FMA is used.

Algorithm 3.8 res = Dot2(a,b)

$[r, \epsilon]$ =2Product(a_1, b_1)
for i = 2 **to** n **do**
  $[p, \pi]$ =2Product(a_i, b_i)
  $[r, \sigma]$ =2Sum(r, p)
  $\epsilon = RN(\epsilon + RN(\sigma + \pi))$
end for
$res = RN(r + \epsilon)$

We mentioned several times that these compensated algorithms provide twice more accurate results. We now give more explanations about this important property focusing on the summation case (the dot product or polynomial evaluation cases are similar for instance). Higham in [28] defines the term

$$\gamma_n = \frac{n\mathbf{u}}{1 - n\mathbf{u}},$$

as a concise way of handling the error analysis bounds. Using correct rounding, classic backward rounding error analysis [28] bounds the error in the computed value $\hat{s}$ of $s = \sum_{i=1}^{n} x_i$ as:

$$|\hat{s} - s| \leq \gamma_{n-1} \sum_{i=1}^{n} |x_i|. \tag{3.2}$$

Here $\hat{s}$ is computed with, for instance, the naive accumulation algorithm ($s = s + x_i$) in the working precision $u = 2^{-t}$. Rump and his co-authors prove in [50] that their compensated summation, summing in precision $\mathbf{u}$, actually returns a floating-point number $\hat{s}_c$ that verifies:

$$|\hat{s}_c - s| \leq \mathbf{u}|\sum x_i| + \gamma_{n-1}^2 \sum |x_i|. \tag{3.3}$$

Rewriting (3.2) or (3.3) to exhibit the relative error of the classic or the compensated sum, respectively $|\hat{s} - s|/|s|$ and $|\hat{s}_c - s|/|s|$, allows us to identify the scaling factor $\sum |x_i| / |\sum x_i|$. This factor is known to be the condition number of $\sum x_i$. In the compensated bound (3.3), it only applies to the second order term in $\mathbf{u}^2$, *i.e.* in twice the working precision. Hence for condition number smaller than $1/\mathbf{u}$, $|\hat{s}_c - s| \lesssim \mathbf{u}$, which means that the computed sum is almost maximally accurate at precision $\mathbf{u}$. For larger condition number, the accuracy of the ill-conditioned sum is dominated by the second term in (3.3).
Similarly the condition number for the dot product is:

$$2 \times \sum |a_i \cdot b_i| / |\sum a_i \cdot b_i|.$$

When these conditioning are larger than $1/\mathbf{u}$, one needs more than twice the working precision to remain accurate. Other levels of compensation can be applied, like detailed in the next section.

Computing k times more accurate result

When the vectors are ill-conditioned a twice working precision is not sufficient. Algorithms 3.10 and 3.11 respectively called SumK and DotK have been proposed to solve these cases [50].

Algorithm SumK starts with calling Algorithm 3.9 which was proposed by Kahan [35]. VecSum is a "distillation algorithm" which transforms a vector a into another vector of an identical sum where $a_n = RN(\sum a_i)$, but with a conditioning improved by a factor **u**. VecSum is also one EFT for the vector sum.

Algorithm 3.9 a = VecSum(a)

for i=2 **to** n **do**
  $[a_i, a_{i-1}]$ =2Sum(a_i, a_{i-1})
end for

SumK repeats this distillation $k-1$ times, followed by a final recursive summation. The result is improved as if k times the working precision was used. Clearly we note that for $k = 2$, Sum2 and SumK are identical.

Algorithm 3.10 res = SumK(a,k)

for K=1 **to** k-1 **do**
  a =VecSum(a)
end for
for i=1 **to** n-1 **do**
  $c = RN(c + a_i)$
end for
$res = RN(a_n + c)$

To compute a compensated dot product that provides k times more accurate result, Ogita and *al.* [50] propose Algorithm 3.11 which is derived from SumK and Dot2.

Algorithm 3.11 res = DotK(a,b,K)

$[p, r_1]$ =2Product(a_1, b_1)
for i = 2 **to** n **do**
  $[h, r_i]$ =2Product(a_i, b_i)
  $[p, r_{n+i-1}]$ =2Sum(p, h)
end for
$r_{2n} = p$
$res =$ SumK$(r, k-1)$

3.1.3 Expansions

Expansions implement multi-precision computation by representing a number with n-components of floating-point numbers. Expansions of length 2 were initially proposed by Dekker in 1971 [14]. In 1997, Shewchuk proposes algorithms of summation and product between expansions taking the benefit of the IEEE-754 standard properties [59]. More recently, Daumas proposes faster and more integrated solution for the product between expansions [13].

One expansion is a representation of a real number x by a sequence of n-components $(x_1, x_2, \ldots, x_n)$ of floating-point numbers where:

— the value of the expansion x is the exact, not rounded, sum of its components,

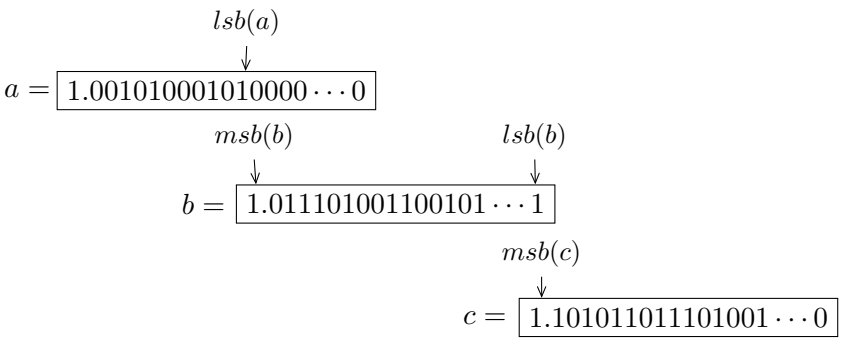
— a component x_i may be equal to zero,

— the components are ordered by magnitude,

— the sub-sequence of non-zero components must be non-overlapping. The two floating-point values x_i, x_{i+1} are non-overlapping, if the least significant nonzero bit of x_i is more significant than the most significant nonzero bit of x_{i+1}.

Figure 3.4 illustrates an overloading instance. The lbs and the msb of the component refer to the 'least significant bit' and the 'most significant bit' respectively. For a non-overloading of two non-zero components x_i and x_{i+1}, it is sufficient that $lsb(x_i) > msb(x_{i+1})$.

FIGURE 3.4: The a and b components are non-overlapping while the b and c ones are overlapping



As the number of components of an expansion increases, the representation accuracy increases also but its arithmetic becomes more costly. In the following, we focus on expansions with two components that are usually named double-double numbers.

Double-doubles

The double-double number x is represented by a pair (x_h, x_l) of *binary64* numbers. Double-double operations compute in twice the working precision, as demonstrated in [43]. These algorithms necessitate a re-normalization phase to make sure that the two components (x_h, x_l) do not overlap in the result. This phase uses Fast2Sum, (Algorithm 3.1). Dekker proposes algorithms to compute the sum and the product of two double-double numbers [14]. More accurate ones are integrated in the QD_Lib [27], QD stands for quad-double where every number is represented by a 4-length expansion. It also provides algorithm DD_2Sum and DD_2Product that compute respectively the sum and the product of two double-double numbers (see Algorithms 3.12 and 3.13, respectively).

Algorithm 3.12 $[res_h, res_l]$ = DD_2Sum(a_h, a_l, b_h, b_l)

$[r_h, r_l]$ =2Sum(a_h, b_h)
$[s_h, rs_l]$ =2Sum(a_l, b_l)
$c = RN(r_l + s_h)$
$[u_h, u_l]$ =Fast2Sum(r_h, c)
$w = RN(s_l + u_l)$
$[res_h, res_l]$ =Fast2Sum(u_h, w)

Algorithm 3.13 $[res_h, res_l]$ = DD_2Product(a_h, a_l, b_h, b_l)

$[r_h, r_l]$ =2Product(a_h, b_h)
$r_l = RN(r_l + (a_h \times b_l))$
$r_l = RN(r_l + (a_l \times b_h))$
$[res_h, res_l]$ =Fast2Sum(r_h, r_l)

In these double-double algorithms, the intermediate computed results are represented with two floating-point numbers. Contrary to the compensated methods, the errors of the computation are not accumulated in any term. The results are consistently represented in twice the working precision.

The double-double algorithms can be applied automatically, by simply overloading the basic operations. Algorithm 3.14 computes the sum of a floating-point vector a, *i.e.* a_i is not a double-double number. Sum_DD uses Algorithm 3.12 and requires $10(n-1)flop$.

Algorithm 3.14 res_h = Sum_DD(a)

$s_h = a_1$
$s_l = 0$
for i = 2 **to** n **do**
  $[s_h, s_l]$ =DD_2Sum$(s_h, s_l, a_i, 0)$
end for

3.1.4 Compensation *versus* double-double

The rounding error analysis and corresponding error bounds of expansion algorithms are presented in [48]. We have already presented the accuracy behavior of compensated algorithms. In term of accuracy, both the $k = 2$ compensation and the double-double methods provide accurate results, *i.e.* as if computed in twice the working precision.
In term of running time extra-cost, several studies compare these two methods. We now mention some of them. In [44] and [39], the authors find that the compensated algorithm for the polynomial evaluation with the Horner schema runs at least twice as fast as the double-double one. This is due to the additional re-normalization phase of the double-double algorithms (avoided in the compensated ones). This phase penalizes the performance because it sequentializes the computation and reduces the exploitation of the instruction

level parallelism by the processor. Moreover, in [65], the author compares the accurate sum of a vector performed with the compensated algorithm Sum2 and the double-double one Sum_DD (Algorithms 3.7 and 3.14, respectively). The loop costs $7flop$ in Sum2 and $10flop$ in Sum_DD. A relatively close runtime might be expected with this difference. However it is not the case in practice. A significant difference is measured, where the compensated algorithm is 3-4 times faster than the double-double one. The author presents Table 3.1 to show the execution time of these two algorithms when varying the size n of the vectors. This significant difference is explained by the fact that Sum2 takes more benefit of the instruction level parallelism than Sum_DD, more details are in [65].

TABLE 3.1: Execution time in microseconds for the algorithms Sum2 and Sum_DD [65].

n	10^2	10^3	10^4	10^5	10^6
Sum2	0,2	2,3	23	243	2579
Sum_DD	0,8	7,9	78	789	8174
Sum_DD/Sum2	4	3,43	3,39	3,25	3,17

Compensation and double-double methods are essentially different in two points.

1. The required knowledge in numerical analysis and computer arithmetic for designing or applying compensation methods is higher than that for using double-doubles. Indeed, the application of the later can be done by simply overloading the basic floating-point operations along any given numerical algorithms.
2. Compensated algorithms have a better instruction level parallelism and therefore offer a better performance than the double-double ones.

To conclude and as reported in L.Thevenoux PhD thesis [65], double-doubles lead to automatic and easy to implement solutions. Compensation ones are more efficient but must be applied manually and should required a bit of numerical expertise.

3.1.5 More accuracy up to recover numerical reproducibility

Compensated and double-double methods have been actually applied and compared in some computation cases to recover numerical reproducibility.
For instance, a code of finite volume calculations is considered in [55]. A parallel global sum provides non-reproducible results because of the non-deterministic order of reduction. The application of compensation and double-double methods successfully recovers the reproducibility. At the contrary using the 80 bits length long-double precision only improves it. The parallel compensation of the global sum is computed by using a (sum,correction) pair as a C data type. MPI_Allreduce provides a customized reduce operation which is here defined as an error-free transformation of the sum. A MPI_Datatype is used to encapsulate the local sum and the local correction term. The authors compare the memory and computational extra-cost to exhibit that the long-double technique is the least costly: it benefits from the hardware implementation of 80 bits length, against simulated 128 bits in the other methods. The full 128-bit double-double is 60 times as expensive due to being implemented in software. The greatest improvement in reproducibility is with the compensation techniques that benefit from an additional cost of around half-a-percent of the total running time.
A similar study is proposed in a climate simulation in [23]. In this case, the authors resume that in theory the running time of the compensation method is better than the double-double one. But they observe that the global summation takes a very small part of the code compared to the overall execution (less than 1%). So, applying these two accurate methods does not influence the whole running time.
Another study to recover the reproducibility in a molecular dynamics simulation using GPU is published in [63]. The authors describe a significant performance improvements when single precision floating-point arithmetic is used, but which introduces a loss on accuracy and reproducibility of the results. They resolve it by using a numeric type `float2`, called "composite precision floating-point numbers", and defined as:

```
struct float2
float x; //x2.value
float y; //x2.error
```

```
x2;
...  //all simulation computations
float x2 = x2.x + x2.y;
```

The conversion from `float2` structures back to float structures is performed by adding the value and the error terms as in the compensation technique. The pair [`x,y`] is computed by redefining the floating-point operations $(+, -, \times)$ which perform the calculation as well as keep track of inherent error. That allow the authors to combine double precision accuracy with single precision performance, in order to improve reproducibility.

3.2 Reproducibility regardless of the accuracy

Contrary the previous section, we now describe some methods that do not improve the accuracy to recover the numerical reproducibility. They eliminate the parallelism drawbacks which lead to numerical non-reproducibility. As mentioned, these main drawbacks are the non-deterministic parallel reduction and the different propagation of the rounding errors when the number of processors change. We also end this subsection describing some works that apply these methods to recover the numerical reproducibility in several computation cases.

3.2.1 Demmel and Nguyen reproducible algorithms

In 2013, Demmel and Nguyen have introduced algorithms to compute reproducible sums of a vector v of size n, *i.e.* bitwise identical results independently on the summation order [15]. Their solutions derive from Rump-Ogita-Oishi's AccSum and FastAccSum algorithms [56]. One new error free transformation algorithm, like Algorithm 3.6, computes the exact sum of the high-order parts of a vector relying on a large value $M = 2^k$ that depends on the conditioning of the sum and its length: $M \geq \sum_{i=1}^{n} |v_i|/(1-2n\mathbf{u})$. But to obtain a reproducible sum, the high-order parts must be reproducible as well as M. Demmel and Nguyen propose Algorithm 3.15, where $M = 2^{\lceil \log_2(\delta_1) \rceil}$ is computed depending on:
i) $m = max(|v_i|)$ which is reproducible, and
ii) $\delta = RN(n \cdot m/(1-2n\mathbf{u}))$.

More accuracy can be obtained by a pre-rounding step that returns K shrunks over the entries such that the floating-point sum of each shrunk remains exact. A shrunk width depends on the maximum absolute value of the entries (as m) and their number (as n). That allows the user to increase the accuracy if the application requires it, but this reduces the efficiency of the method.

A parallel implementation of ReprodSumK and FastReprodSum algorithms introduces two reductions:

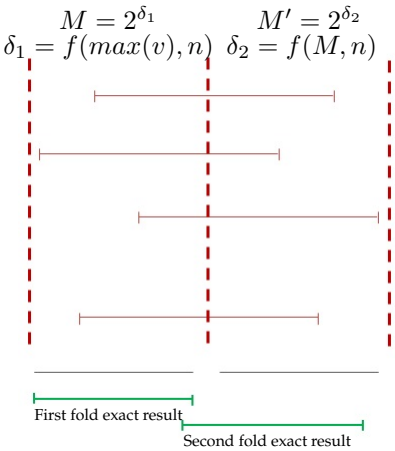
— max: the computing units exchange their local $max(|v_i|)$ to obtain the global one

— sum: the computing units exchange also their local sums which are computed in the same pre-rounding, to obtain a reproducible global sum.

Implementation is efficient in absolute but we will explain in Section 5.1.2 how it appears to be expensive in our finite element application mainly, because of the two reductions.

Algorithm 3.15 T = ReprodSumK(v,k)

$m = max_i(|v_i|)$
$\delta_1 = RN(n \cdot m/(1 - 2n\mathbf{u}))$
$M_1 = 2^{\lceil \log_2(\delta_1) \rceil}$
for f = 1 **to** k-1 **do**
  $[T_f, v] =$ ExtractVector(M_f, v)
  $\delta_{f+1} = RN(n \cdot (2\mathbf{u} \cdot M_f)/(1 - 2n\mathbf{u}))$
  $M_{f+1} = 2^{\lceil \log_2(\delta_{f+1}) \rceil}$
end for
$T_k = 0$
for i = 1 **to** n in any order **do**
  $q_i = RN(RN(M_k + v_i) - M_k)$
  $T_k = RN(T_k + q_i)$
end for
$T = 0$
for i = 1 **to** k **do**
  $T = RN(T + T_i)$
end for



Later, Demmel and Nguyen propose in [16] an improved "1-Reduction" algorithm. It is similar to Algorithm 3.15 in the principle of performing a pre-rounding step to return K shrunks, but with an advantage that avoids having to compute the global maximum absolute value. A set of separation boundary values (M_i): is statically defined (with respect to the exponent range of the FP numbers and the length of the sum) and so is shared by all computing units. Each computing unit uses its own local boundary (which is the largest one to the right of the leading bit of the local maximum absolute value) and perform the same pre-rounded accumulations as previously. Since each processor can have a different boundary, and only intermediate values corresponding to the same boundary are summed. In parallel execution, this algorithm benefits from only one reduction for the summation of partial sums with their own boundary.

3.2.2 Integer arithmetic is reproducible

The main source of non-reproducibility is the non-associativity of the floating-point addition. Integer addition is associative, so in [21] and [10], they started from this principle and transform the floating-point additions to integer ones. In practice, the double precision numbers are mapped to 64 bits integer numbers before the addition operation. A transformation of the form $IV = Q \times V$ is performed to map a floating-point V to the IV integer thanks to a scaling factor Q that depends on both the floating-point and integer ranges. This latter one must be not too large because large double precision value will overflow the integer range, nor too small because small ones will be flushed to zero.
In general, this method increases the computing error because the integer conversion reduces the accuracy below the double precision's one. The degree to which it is reduced depends on the choice of the scaling factor and on the range of the floating-point entries. This will be detailed in Section 5.1.3.

3.2.3 Deterministic parallel reduction

Another method that eliminates the non-associativity of the parallel addition is to force a specific computation order in the reduction.
In [68], Villa and *al.* observe the convergence rate of a conjugate gradient calculation in a Power State Estimation (PSE) simulation. This convergence criterion

uses the Euclidean norm of the residual vector, $||v|| = \sqrt{\sum_{i=1}^{n} v_i^2}$. This norm computes a parallel dot product for the whole domain and the source of the non-reproducibility arises from the non-deterministic reduction of the partial dot product. They compare two methods that may recover the reproducibility. The first one is the double-double method already detailed in Section 3.1.3. In this application, the double-double dot product improves but does not completely recover the reproducibility. It also leads to an important extra-cost. The second method is a custom tree-based parallel reduction that deterministically performs reductions. In each level, a single thread accumulates sequentially j different values, where j is a fixed chosen value. Since the "shape" of the reduction tree always depends on the constant j, the reduction is deterministic and so provides reproducible results. Here is an example of the shape of the tree reduction when $j = 2$:

$$
\begin{array}{lcccccccc}
\text{Level 1} \rightarrow & a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 \\
\text{Level 2} \rightarrow & a_0 + a_1 & & a_2 + a_3 & & a_4 + a_5 & & a_6 + a_7 & \\
\text{Level 3} \rightarrow & a_0 + a_1 + a_2 + a_3 & & & & a_4 + a_5 + a_6 + a_7 & & & \\
\text{Level 4} \rightarrow & a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 & & & & & & &
\end{array}
$$

The running time of this method varies as the j value changes. A suitable range values of j produces an acceptable running time, but smaller or larger values reduce the performance. By comparing the two methods, they conclude that the deterministic tree scheme is of course reproducible but potentially less accurate than the double-double's one whereas introducing an extra-cost significantly less than the double-double approach.

In a simulation of sheet metal forming based on a finite element method [17], Diethelem studied the non-reproducibility of a parallel dot product in the conjugate gradient method. As the previous study, he proposes a deterministic order of reduction. The procedure introduces a parameter p^* that defines a number of virtual processors. The actual number of processors has to be $p \leq p^*$. The parallelism is achieved as if p^* processors are available. The overall task is always distributed across p^* virtual processors, while the p physically processors have to do the work of one virtual processor. When it completes this sub-task, it does the work of another virtual processor if necessary, and so on. In this way, they estimate to ensure the reproducibility of the partial results.

For the final result they always assemble the partial p^* in the same order. In other words, the order of the accumulation always depends on p^* but the work is actually done by p processors.
For instance of a dot product where $n = 8$, and a chosen $p^* = 4$. The partial sums are: $s_1 = x_1y_1 + x_2y_2$, $s_2 = x_3y_3 + x_4y_4$, $s_3 = x_5y_5 + x_6y_6$, $s_4 = x_7y_7 + x_8y_8$. By varying $p = 2, 3 < p^*$, the partial sums are always the same and for the final result they always assemble the partial p^* in the same order, $s = s_1 + s_2 + s_3 + s_4$.

s_i		physical procs. p_1	physical procs. p_2
virtual procs.	p_1^*	s_1	
	p_2^*		s_2
	p_3^*	s_3	
	p_4^*		s_4

s_i		physical procs. p_1	physical procs. p_2	physical procs. p_3
virtual procs.	p_1^*	s_1		
	p_2^*		s_2	
	p_3^*			s_3
	p_4^*		s_4	

The disadvantages of this method are: i) the important loss of performance because it is not always possible to distribute the virtual processor's work to the physically available processors in a balanced way. This may lead that certain physical processors become idle while others are still busy doing their computations. ii) The restrict of a chosen fixed p^* where the application can not benefit from larger number of processors.

This technique has the advantage that no modification to the code is needed once this technique is integrated. But a drawback is the high extra-cost that can be discussed by comparing the parallelization benefit to have a best execution time [7] .

3.3 Conclusion

In this chapter we present several methods that improve parallel floating-point computation up to recover the numerical reproducibility. We distinguished two types of approaches: those that increase the accuracy until the remaining errors become smaller than the working precision and the others that provide one potentially inaccurate but associative floating-point addition.

We summarize that both the compensated and the expansions methods can provide the same result accuracy. However, the first one benefit from a lower extra-cost but are more complicated to apply. On another hand, providing a parallel deterministic order of reduction leads to the great advantage that no modification to the code is needed once this technique is integrated. Nevertheless this technique only corrects the order of the parallel reductions. Since the non-deterministic propagation of rounding errors in the local sums may still forbid reproducibility of the reduced results even if a static order is used, this solution does not completely solve the non-reproducibility issue. This important drawback will be detailed in Chapter 4. The Demmel and Nguyen's algorithm (ReprodSum) provide multi-level of accuracy while being reproducible every time. One drawback is the necessity to two parallel reduction, which may remain relatively costly as we will measure it in Section 5.1.2. The recent algorithm "1-Reduction" is not applied in this work but remains an interesting perspective. Integer conversions are not costly but may produce significant loss of accuracy if the range of values is relatively large, which is *a priori* the case in HPC simulations.

In this chapter, we mentioned HPC applications of many scientific domains that report failures of the numerical reproducibility of their results. Most of the previously cited papers analyze the non-reproducibility due to the non-deterministic order of the parallel reduction, while in practice it is not the only source of non-reproducibility. In the following chapters, we will focus on the non-reproducibility of a finite element simulation software, namely in the openTelemac suite. It is interesting to mention now Smith and Margetts work [61]. They analyze the non-reproducibility issue in a rather abstract point, by studying the influence of the parallel global sum reduction in iterative solvers in a finite element simulation. Nevertheless, they also exhibit that another source of non-reproducibility is due to domain decomposition. While each sub-domain solves a partial part of the domain, some nodes are shared between them. The same analyze is mentionned in [9] concerning openTelemac. We will explain that is due to the different rounding errors propagation to these nodes when the number of processors changes (Chapter 4). Then we detail how to manage the computation of these shared nodes to correct them (Chapter 5) and so recover the expected numerical reproducibility.

Chapter 4

The rounding error propagation in finite element simulations

In this chapter we introduce and describe the main steps of a finite element method (FEM). This presentation is definitely related to openTelemac's implementation of a finite element resolution. Other FE software will share the same principles but certainly present significant implementation differences. This chapter aims to detail the openTelemac computations which are necessary to explain the sources of the reproducibility failures.
It is organized as follows. Section 4.1 introduces the openTelemac suite, specially the two studied modules, Telemac-2D and Tomawac. In Section 4.2 and Section 4.3, we describe the the building and solving phases that are the main steps of a finite element method. These computation suffer from non-reproducibility and are corrected in the next chapter. In Section 4.4 we present how parallelism is processed in this finite element method and in Section 4.5 we exhibit the non-reproducible resolutions with the two modules Telemac-2D and Tomawac. Finally in Section 4.6, we identify the sources of these reproducibility failures and detail the effects of the FP arithmetic and of the parallelism on these computations.

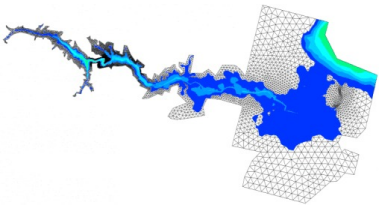
4.1 The openTelemac suite

4.1.1 An introduction of openTelemac

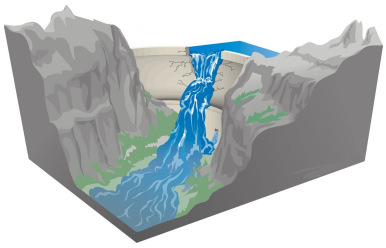
Since 1987, the Research and Development (R&D) Department of Electricité de France (EDF) launched the openTelemac project [51]. The objective was to develop a hydro-informatics system dedicated to free surface flows. This

project is of considerable strategic interest: dams, thermal and nuclear power stations on seaboard or river banks interact with an aquatic environment and have to be supervised to protect people. The *Malpasset* dam failures, for instance, is an example of a real disaster to avoid. It was an arch dam on the Reyran River, Figure 4.1b, located approximately 7 km north of Fréjus on the French Riviera (Côte d'Azur), in the Var department, southern France. This tragedy occurs on December 1959, killing 423 people in the resulting flood and the damage amounted to a total of 68 million dollars.

FIGURE 4.1: *Malpasset* dam failure.



(a) The *Malpasset* dam mesh illustrates the flow of water after 2200 seconds.



(b) The *Malpasset* dam model.

OpenTelemac is an integrated set of open source software written in Fortran 90. It consists in more than 300,000 lines of code issue from a 20 years of international collaboration. More than 4000 users are registered on its official web site [51]. OpenTelemac includes a finite element library BIEF (BIbliothèque d'Elements Finis in French), which provides the data structure, especially the types of matrix storage. BIEF also includes methods for solving advection equations, diffusion equations, linear system inversion with different types of preconditioning. BIEF can be used to carry out all the conventional operations like dot product, matrix-vector product, adding vectors, *etc.* OpenTelemac consists in several modules, each one solves a specific problem. The studied ones in this thesis are Telemac-2D and Tomawac, where their systems are built as detailed in the following.

The Telemac-2D module

This module is a 2D hydrodynamics module which solves the Saint Venant equations using the finite element method for a computational mesh of triangular elements. It takes into account several phenomena, *e.g.* propagation of

long waves, bottom friction, atmospheric pressure and wind, *etc.* The main results at each node of the computational mesh are the depth of water H and the two velocity components U, V. Finite element method leads to build and solve a general sparse linear system which is a mixed of three sub-systems:

$$\begin{pmatrix} A_{hh} & A_{hu} & A_{hv} \\ A_{uh} & A_{uu} & 0 \\ A_{vh} & 0 & A_{vv} \end{pmatrix} \begin{pmatrix} H \\ U \\ V \end{pmatrix} = \begin{pmatrix} B_h \\ B_u \\ B_v \end{pmatrix} \tag{4.1}$$

The sub-members in System (4.1) are computed from physic algebraic equations, by taking into account all the physical condition inputs. The zero submatrices correspond to the coupling absence between the two velocity components.

Telemac-2D simulations depend on many physical and numerical parameters which are chosen by the user. For instance in our work, the treatment of the linear system is linked to the pseudo wave equation. This treatment simplifies the linear system combining three unknowns by eliminating the velocity from the continuity equation at the discrete level. So it decouples water depth and velocity in System (4.1) to obtain a simplified one:

$$\begin{pmatrix} A_1 & 0 & 0 \\ 0 & A_2 & 0 \\ 0 & 0 & A_3 \end{pmatrix} \begin{pmatrix} H \\ U \\ V \end{pmatrix} = \begin{pmatrix} C_1 \\ C_2 \\ C_3 \end{pmatrix}, \tag{4.2}$$

where A_2 and A_3 become diagonal matrices using the lumped-mass method. The matrix A_1 and the three second members are defined by the following algebraic transformations :

$$\begin{aligned} A_1 &= A_{hh} - A_{hu}A_2^{-1}A_{uh} - A_{hv}A_3^{-1}A_{vh}, \\ C_1 &= B_h - A_{hu}A_2^{-1}B_u - A_{hv}A_3^{-1}B_v, \\ C_2 &= B_u - A_{uh}H, \\ C_3 &= B_v - A_{vh}H. \end{aligned} \tag{4.3}$$

More details of these transformations are presented in [25]. It is important to note here that the two velocity second members C_2 and C_3 depend on the H unknown values. Hence, this procedure mixed the building and the resolution

phases. The new System (4.2) is solved in two steps: H is first computed by applying the conjugate gradient method to $A_1 H = C_1$. Then U, V derive from H thanks to A_2 and A_3 diagonal systems.

The Tomawac module

This module is used to model wave propagation in coastal areas. By means of a finite element method, it solves a simplified equation for the spectro-angular density of wave action. This is done for steady-state conditions. It takes into account several physical phenomena, *e.g.* wind-generated waves, refraction on the bottom, refraction by currents, *etc.* At each node of the computational mesh, it calculates the significant wave height, the mean wave frequency, the mean wave direction, the peak wave frequency, the wave-induced currents and the radiation stresses.
It solves a transport equation where a first order of partial differential equations (PDE) is changed as ordinary differential equations (ODE) along its characteristic curves [25]. In practice, this turns to solve a diagonal linear system:

$$Ax = b, \tag{4.4}$$

where A is a diagonal matrix. Only the second member b is built from the finite element assembly, as we will detail in Section 4.2.2.

4.2 The Finite Element Method

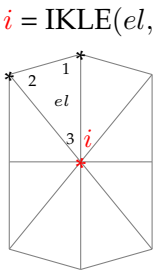
The finite element method (FEM) is a numerical technique to compute approximate solutions to boundary value problems for PDE [52]. FEM subdivides a continuous domain into smaller, simpler parts, called finite elements. It represents the studied domain (which is often complex) by a simple geometry of elements. These elements may be of numerous geometrical shapes. In this thesis, we always consider triangular elements. No overlap nor gap between two neighbor elements, *i.e.* with a common border, exists. This condition is described as: $\bigcup_{el=1}^{nel} \Omega_{el} = \Omega$ and $\Omega_j \bigcap \Omega_{j'}$ is only a vertex for $j \neq j', (j, j') \in \{1, \cdots, nel\}$, where Ω is the continuous domain and Ω_{el} is a discrete domain composed of nel elements el.
The main steps of the finite element method are the following.

1. *Discretize the continuous studied domain.* The domain is divided into finite elements of simple shapes. The finite element mesh is typically generated by a pre-processor program.
2. *Select interpolation functions.* These functions are used to interpolate the field variables over the element.
3. *Assemble the element equations.* To find the global equation system for the whole studied domain, the element equations are assembled on every domain node. The principle of this step is detailed in Section 4.2.2.
4. *Solve the global equation system.* The finite element global equation system is typically sparse, symmetric and positive definite. Direct and iterative methods can be used to solve it. We consider the preconditioned conjugate gradient in Section 4.3.1. The unknowns at the mesh nodes are the solution of the system.

4.2.1 The local and global descriptions of a finite element

Figure 4.2 illustrates a triangular mesh where the domain is split into elements $el = 1, \ldots, nel = 8$. Each element has ndp vertices depending on its shape with a local number. The triangular mesh here satisfies $ndp = 3$ and $idp = 1, 2, 3$. Each node (which is also an element vertex) has a global number $i = 1, \ldots, np$ over the whole mesh, here $np = 9$.

FIGURE 4.2: A triangular finite elements description



The connection between the local number idp in one element el and the global number i in the whole mesh is called the element connectivity [54]. Practically, the mapping between these two specific numberings is implemented as a connectivity table IKLE. We obtain the global number i of a node from the local number idp in the element el by the connectivity table as,

$$i = \text{IKLE}(el, idp).$$

The numerical solutions are computed on every node from the values in the elements by the finite element assembly step.

4.2.2 Finite element assembly

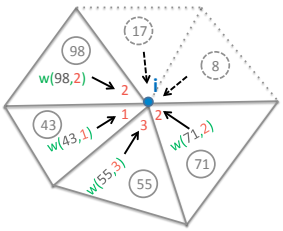
The main step in a finite element method is the finite element assembly that builds the linear system. It recovers the finite element values to express them as nodal values. In other words, this process builds the global vector V of size np by accumulating the elementary contributions W_{el} for every element el in the mesh that contains the node i. The assembly of node i computes:

$$V(i) = \sum_{el=1}^{nel} W_{el}(i). \tag{4.5}$$

Algorithm 4.1 codes the finite element assembly: the implementation difficulty of this step comes from the management of the node numbering. As already described several numberings are introduced for a given node and tables map from one to another. The assembly loops over the domain for each element vertex $idp = 1, \ldots, ndp$, and then over each element $el = 1, \ldots, nel$. According to these two informations, the IKLE table returns the global number i and the corresponding elementary contribution which has to be accumulated in i.

Algorithm 4.1 Assembly step of the vector V in each node i from its elementary contributions W_{el}.

```
for idp = 1 to ndp do
    for el = 1 to nel do
        i = IKLE(el,idp)
        V(i) = V(i) + W(el, idp)
    end for
end for
```



This double loop defines an indirect accumulation, *i.e.* two consecutive iterations of the inner loop do not apply to the same i value. In practice, this indirection forbids to know the whole assembled vector V until the end of the two nested loops. This will complicate how to efficiently apply reproducible summation algorithms that need to know the whole vector before accumulating, *e.g.* ReprodSumK as we will describe in Section 5.1. This assembly is applied to the elementary vector, the diagonal of any matrix and in the EBE matrix-vector product as we will detail in Sections 4.2.3 and 4.2.4.

4.2.3 The EBE matrix storage

The discretization of a partial differential equation by the finite element method typically leads to large sparse matrices, *i.e.* matrices with very few non-zero entries. So it is important to define a suitable data structure for an efficient implementation. Generally, the main step of a finite element simulation is the assembly technique which is described in the previous Section 4.2.2. It yields a linear system that will be solved. In this thesis, the conjugate gradient method processes this resolution. This classic method does not change the matrix, it only requires a matrix-vector product. Hence the data structure should provide an efficient matrix-vector product. The element-by-element (EBE) storage [24] realizes it with a good efficiency and stores the matrix as vectors.
In a mesh with nel elements and np nodes, a matrix M is written as:

$$M = D + \sum_{el=1}^{nel} X_{el}, \tag{4.6}$$

where D of size np is the assembled diagonal and X_{el} are the elementary contributions of the extra-diagonal values. The diagonal terms are no more stored at the elementary level but are assembled with Algorithm 4.1. The matrix has $ndp*(ndp-1)$ extra-diagonal entries, which are 6 for a triangular element [25]. Each term in X_{el} depends on the 3 local nodes of the element el ($idp = i_1, i_2, i_3$):

$$X_{el} = \begin{pmatrix} \dots & X_{i_1 i_2}(el) & X_{i_1 i_3}(el) \\ X_{i_2 i_1}(el) & \dots & X_{i_2 i_3}(el) \\ X_{i_3 i_1}(el) & X_{i_3 i_2}(el) & \dots \end{pmatrix} = \begin{pmatrix} \dots & X(el,1) & X(el,2) \\ X(el,3) & \dots & X(el,4) \\ X(el,5) & X(el,6) & \dots \end{pmatrix}. \tag{4.7}$$

So in practice, each X_{el} is stored as a vector of size 6:

$$X_{el} = [X(el,1), \dots, X(el,6)].$$

The size and the numbers of vector X_{el} depends on the element shape, *i.e.* on ndp, and the different tranformations are implemented in openTelemac [25].

The advantage is that for a triangular mesh, the storage of the assembled matrix is reduced to $6nel + np$ coefficients by assembling only the diagonal instead of $np \times np$ coefficients when assembling the whole matrix with no

special storage of sparse matrix.

4.2.4 The EBE matrix-vector product

This EBE storage provides an efficient matrix-vector product thanks to its data structure. The result RES of the product of the matrix M by the vector V of size np, satisfies the following equality:

$$RES = D \circ V + \sum_{el=1}^{nel} X_{el} \cdot V_{el}. \tag{4.8}$$

The first operand is a vector R_1 of size np resulting from the Hadamard product (componentwize product):

$$R_1(i) = D(i) \times V(i), \quad i \in \{1, \cdot, np\}. \tag{4.9}$$

The second operand is calculated in two steps.

1. Firstly, the multiplication of the extra-diagonal terms and the vector V_{el} with a mapping *via* the connectivity table from the global numbering i and the local one $i_1, i_2, i_3 \in el$ yields:

$$X_{el}.V_{el} = [X(el,1).V_{i_2}, X(el,2).V_{i_3}, X(el,3).V_{i_3}, X(el,4).V_{i_1}, X(el,5).V_{i_1}, X(el,6).V_{i_2}].$$

2. Secondly, each term of the previous result has to be assembled to a global vector R_2 of size np, in the corresponding node i, by the FE assembly step, *i.e.* applying Algorithm 4.1.

Finally, the two vectors are added to obtain the final result (4.8) of the matrix-vector product:

$$RES = R_1 + R_2.$$

4.3 The linear system resolution

We denote $Ax = b$ the linear system to be solved. Its components have been built by applying the algebraic operations to consider all the physical phenomena and the finite element assembly. Vector x stores the unknowns and b is the second member, both are of size np. A is a sparse matrix which is stored in $6nel + np$ thanks to the EBE storage, as defined in Section 4.2.3. For this type of storage a direct resolution is not adapted because it would need

the knowledge of the assembled values of the matrix. On the contrary, the iterative methods uses the matrix only to perform matrix-vector products which are efficiently computed with an EBE storage, as presented in Section 4.2.4. Here the conjugate gradient (CG) is the chosen iterative method. The number of iterations depends on the system dimension and is reduced by matrix preconditioning. Here a diagonal preconditioning is applied. More details are now introduced.

4.3.1 The conjugate gradient method

The conjugate gradient, historically, was developed as a direct method. Then it became one of the most popular iterative technique for solving sparse symmetric positive definite (SPD) linear systems. In a direct method, the unknowns x are found when the residual of the system verifies: $r = Ax - b = 0$. When using it as an iterative method, a convergence criteria is tested during the iterations. In numerical simulation, it is preferable to choose a relative accuracy ϵ with respect to the norm of the second member b. In openTelemac the user chooses the ϵ value which is 10^{-4} in *gouttedo*. This will ensure that the solution does not depend on the initialization of the unknowns. The convergence test in the m-th iteration is [1]:

$$\frac{\|Ax^m - b\|}{\|b\|} \leq \epsilon. \tag{4.10}$$

An initial x^0 is chosen; for each iteration the residual decreases ($r^{m+1} < r^m$) and the solution is updated as $x^{m+1} = x^m - \rho^m d^m$. That according to the descent direction d^m and the step length ρ^m. The CG algorithm is presented by Algorithm 4.2

1. Iteration index are denoted as upper exponents: x^m.

Algorithm 4.2 The conjugate gradient algorithm

Entries:	matrix A, second member b
Initialization:	vector x^0, the stopping criterion (4.10)
Output:	vector x
Initialization:	$r^0 = \boldsymbol{Ax^0} - b$, $d^0 = r^0$, $\rho^0 = \dfrac{(r^0, r^0)}{(d^0, \boldsymbol{Ad^0})}$
	$x^1 = x^0 - \rho^0 d^0$
Iterate:	until stopping criterion:
	$r^m = r^{m-1} - \rho^{m-1}\boldsymbol{Ad^{m-1}}$
	$d^m = r^m + \dfrac{(r^m, r^m)}{(r^{m-1}, r^{m-1})} d^{m-1}$
	$\rho^m = \dfrac{(r^m, d^m)}{(d^m, \boldsymbol{Ad^m})}$
	$x^{m+1} = x^m - \rho^m d^m$

The matrix-vector product is distinguished in a bold style ($\boldsymbol{Ad^m}$) and the dot product in parentheses (as (r^m, d^m)). Theoretically, the CG computes the exact solution (regardless of rounding errors of the floating point numbers) for a number of iterations smaller than the system size. However, a lot of time it does not converge as fast as desired due to the ill-conditioning of the system. For that, preconditioning techniques modifies the original system to an equivalent one with a better conditioning. The diagonal preconditioning, or the Jacobi preconditioner is a good way for EBE matrix. It is applied before the resolution by modifying only the diagonal of the matrix, which is assembled. The diagonal matrix D is formed such as, $D_{ii} = 1/\sqrt{|A_{ii}|}$ and $D_{ij} = 0$.
The new system is then written as : $DAD\ D^{-1}x = Db$.
The result of the DAD operation is a matrix where the diagonal is only made of ones. So, x is calculated easily once the unknown of the new system ($D^{-1}x$) is computed.

4.4 The parallelism management

4.4.1 The domain partition

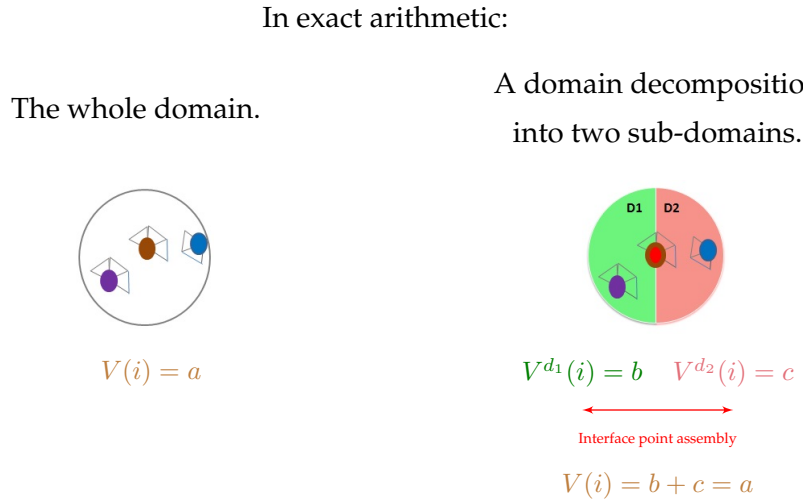
A parallel execution requires to partition the domain. An efficient partition should take into account the partitioning into blocks in order to ensure a good

distribution of the elements between the computing units. As well, this partition has to minimize the number of nodes shared by several sub-domains that need communications between the computing units.
The domain partition in openTelemac is provided by the METIS software [36]. It is a multi-level graph partitioning algorithm that works by applying one or more stages. Each stage reduces the size of the graph by collapsing vertices and edges, partitions the smaller graph, then maps back and refines this partition of the original graph. The partitioning is done without overlapping, edge to edge, so each element belongs to a unique sub-domain. Nevertheless, there are nodes (vertex of element) that are shared at the interface between sub-domains; they are called *interface nodes* or *interface points*.

4.4.2 Interface node assembly: the exact case

Each sub-domain is responsible on the assembling and the solving of its own finite element system. Furthermore, when an interface node belongs to several sub-domains, some communications will be required in order to build or solve the global system.
Figure 4.3 illustrates one node i in the domain which becomes an interface point in the decomposition into two sub-domains, for instance. The value of the node i is $V(i) = a$, while in the parallel case it is known from the two contributions V^{d_1} and V^{d_2} that comes respectively from the partial values at the interface node i on sub-domain d_1 and d_2. Let us write for instance: $V^{d_1}(i) = b$ and $V^{d_2}(i) = c$. The two sub-domains communicate to exchange their value and accumulate them to obtain the global value $b + c = a$. This equality is verified in exact arithmetic but we will detail in Figure 4.8 how it is not the case in floating-point arithmetic.

FIGURE 4.3: The transformation of one node i into an interface point in the domain decomposition into two sub-domains. A communication and a reduction are necessary to obtain the global value of i.



4.5 Non-reproducible results in parallel simulations

Because of the effects of the parallelism with the floating-point arithmetic, openTelemac results suffer from numerical non-reproducibility. We illustrate this problem in each studied modules, Telemac-2D and Tomawac.

4.5.1 Non-reproducibility of Telemac-2D

We studied the non-reproducibility of the *gouttedo* test case, which is available in the distribution of the Telemac-2D module. It is a 2D-simulation of a water drop fall in a square basin. This simulation uses a triangular element mesh (8978 elements, 4624 nodes) and simulates several time steps of 0.2 sec. Figures 4.4, 4.5 and 4.6, exhibit the non-reproducible results of the water depth between the sequential execution and the parallel ones, for $p = 2, 4, 8$ sub-domains, respectively.

— The left plot shows the computed water depth values in the sequential simulation. (One domain, no domain decomposition)

— The middle plot is related to the parallel results. White spots exhibit the mesh nodes that differ from the sequential ones. The loss of reproducible

values increases as the simulation runs in the time scale because the results depend on the previous time step.

— The right plot shows, on each point of the mesh, the maximum relative error between the sequential execution and the p-parallel ones.

We also remark that the relative error increases in the time scale. These values are interesting to *a posteriori* evaluate the stability (conditioning) of the given problem.

In the simulations with $p = 2, 4, 8$ subydomains, the relative errors vary between 10^{-15} and 10^{-13}, that means that from 1 to 3 decimals are different in the result values between sequential and parallel runs. That aims to estimate that a twice working precision can compute one accurate rounding of the results and so will produce reproducible results.

FIGURE 4.4: *gouttedo* : white spots are non-reproducible water depth values between the sequential run (left) and a 2 processors run (middle). Right: relative error map between these two simulations. Time steps: $1, 2, \ldots, 7, 8 \times 0.2$ sec.

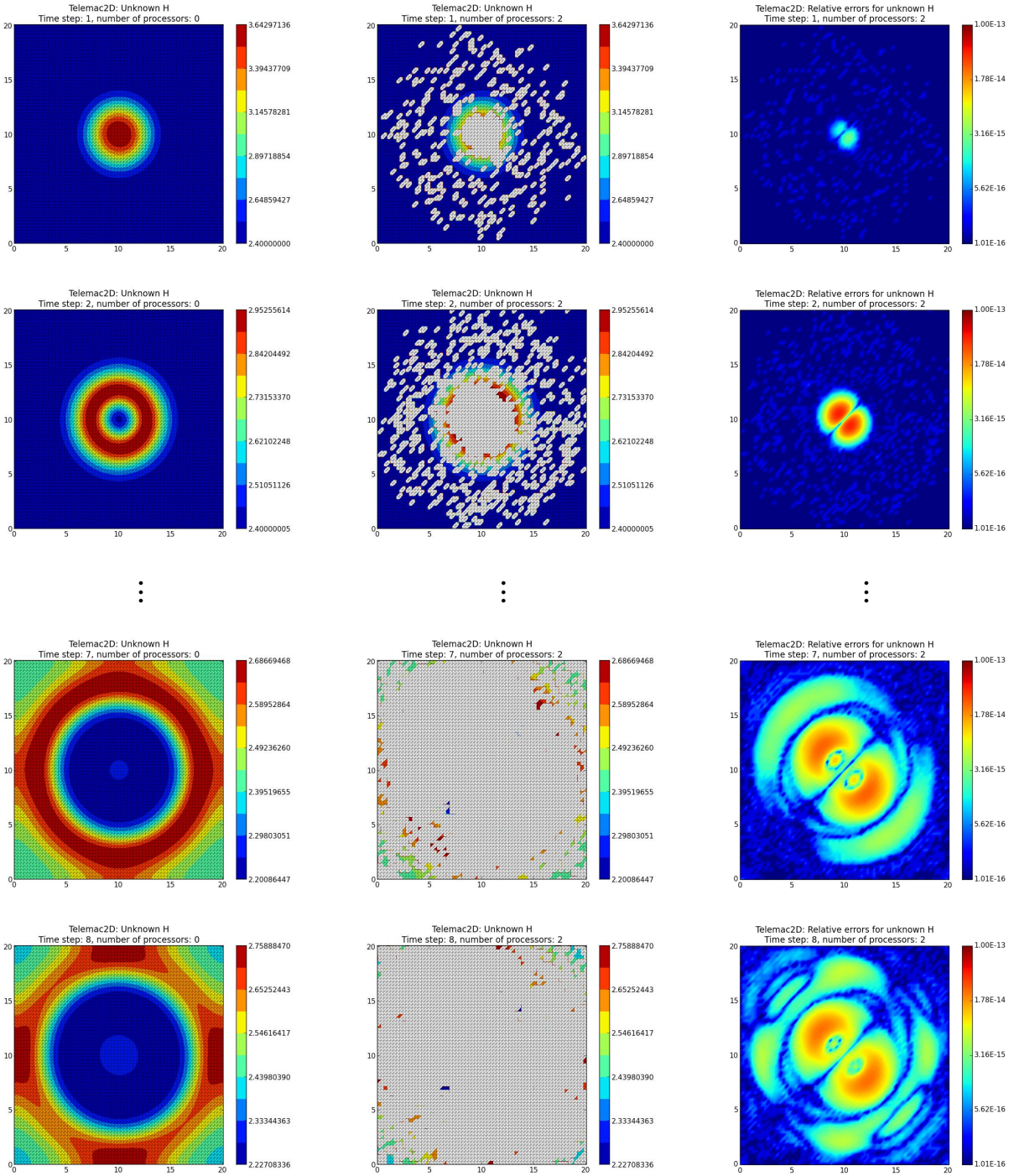


FIGURE 4.5: *gouttedo* : white spots are non-reproducible water depth values between the sequential run (left) and a 4 processors run (middle). Right: relative error map between these two simulations. Time steps: $1, 2, \ldots, 7, 8 \times 0.2$ sec.

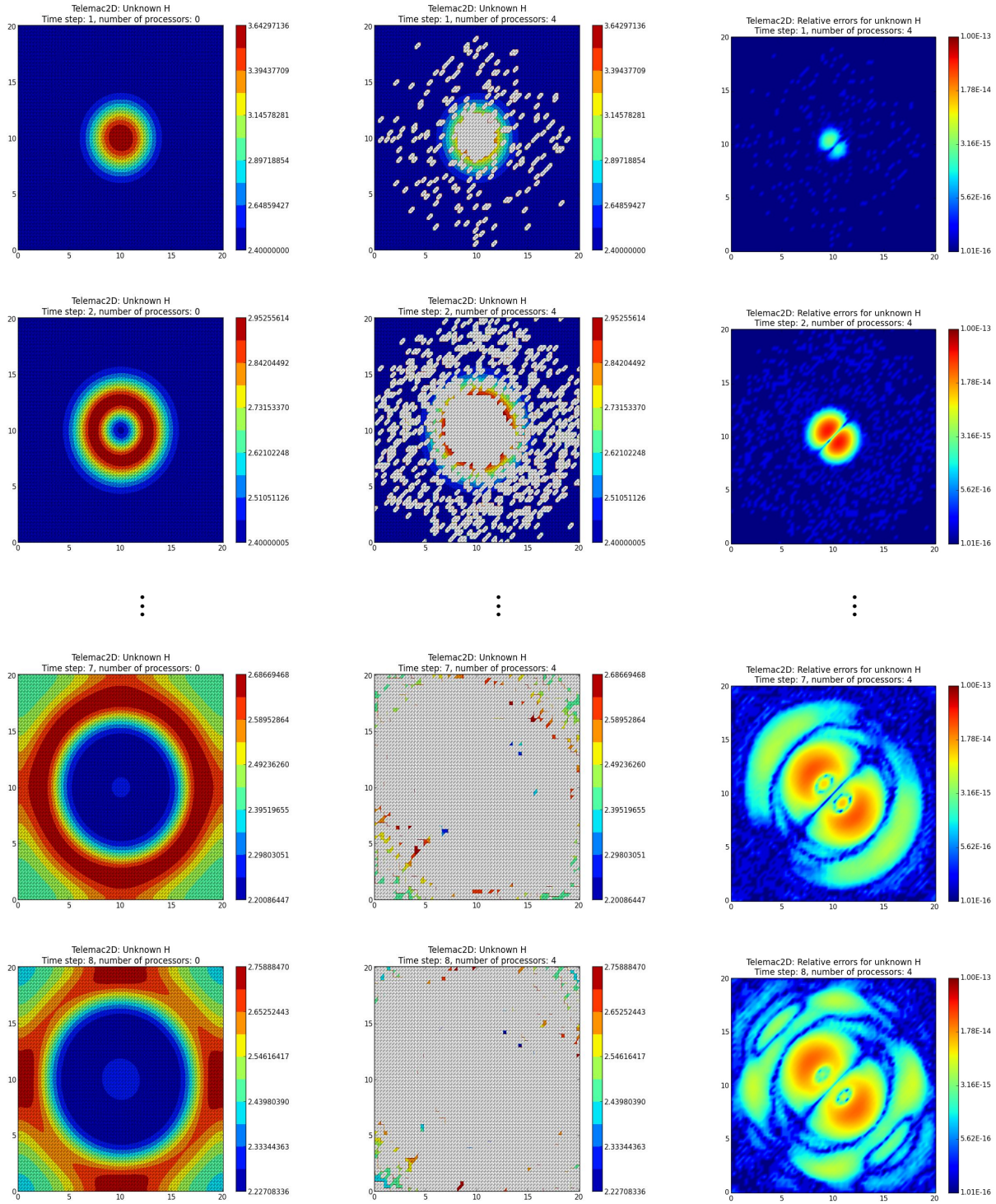
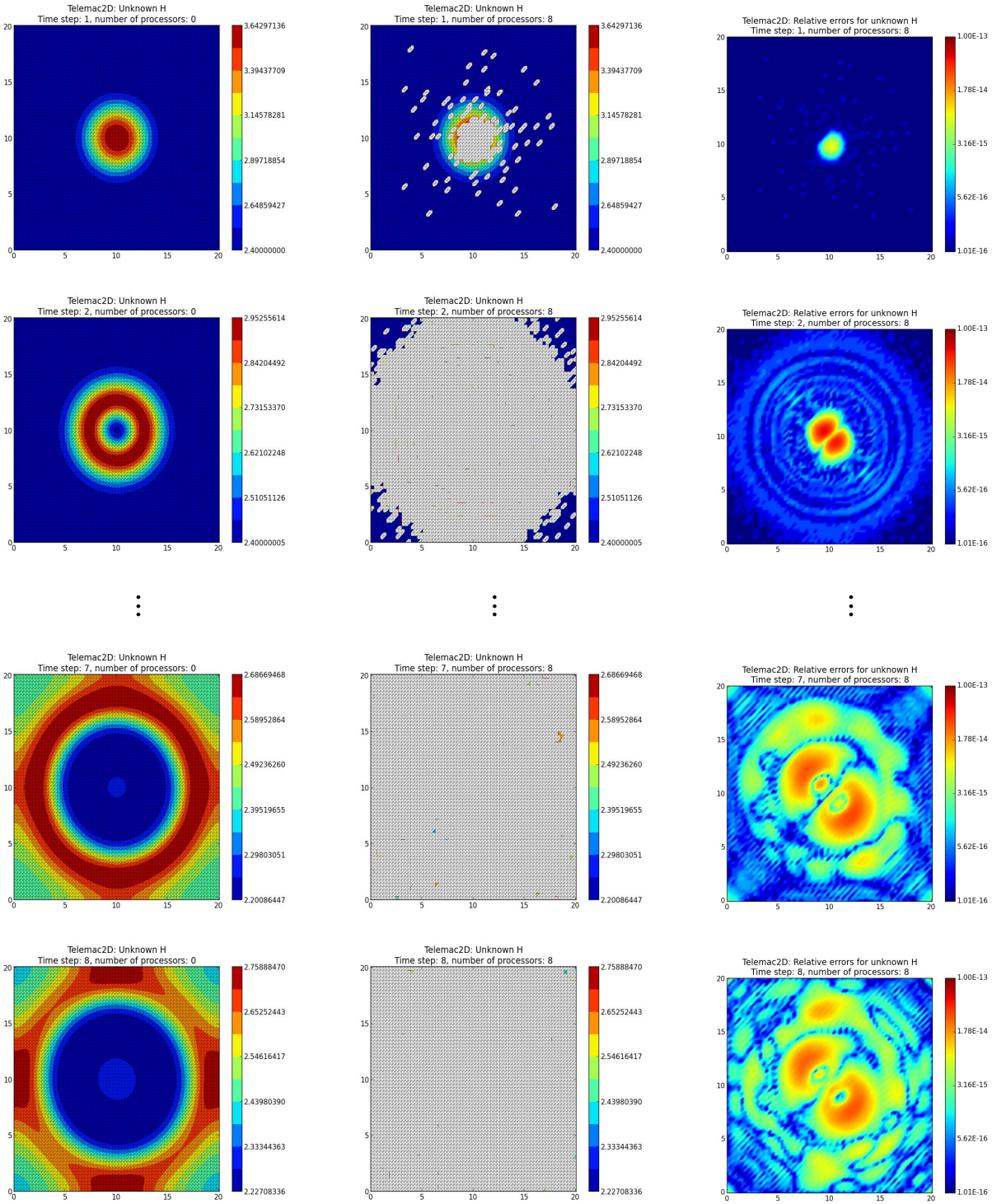


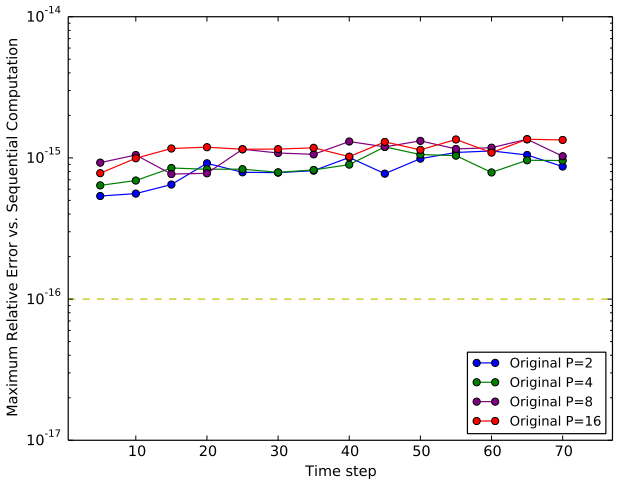
FIGURE 4.6: *gouttedo* : white spots are non-reproducible water depth values between the sequential run (left) and a 8 processors run (middle). Right: relative error map between these two simulations. Time steps: $1, 2, \ldots, 7, 8 \times 0.2$ sec.



4.5.2 Non-reproducibility of Tomawac

We studied the Tomawac's Nice test case which simulates the effect of high-speed ferry waves when approaching the Mediterranean harbor of Nice. We use the notations A^s to denote a sequential simulation and A^p its p-parallel one *i.e.* with p sub-domains. The relative variations between the two simulations to measure the reproducibility as: $\max |A^p - A^s|/|A^s|$. This measure is illustrated in Figure 4.7 between the sequential execution (Original A^s) and the p-parallel (Original A^p), at every mesh node for every time step (x-axis) and for a number of processors (*i.e.* sub-domains) $p = 2, 4, 8, 16$. The non-reproducibility of these computations is illustrated by displaying the maximum relative error as the y-axis. The colored plots correspond to the number of processors and the dashed line is the referenced computing precision (*binary64* with $\mathbf{u} \approx 1.1 \times 10^{16}$).

FIGURE 4.7: Floating-point computations (Original).
Mean frequency wave, module Tomawac, case test Nice



If there was no difference between the sequential and the parallel executions, the colored plots would all overlap on the dashed line. If the parallel results were similar when varying the number of processors, the colored plots would overlap each other. But it is not the case. The relative errors are around the value 10^{-15} that means that 1 decimal is different in the result values between the sequential and the parallel runs. We will further explain that in Tomawac simulations, the numerical reproducibility failure is limited to one

source which is the FE assembly step that builds the second member of the system.

4.6 When floating-point arithmetic meets parallel computing

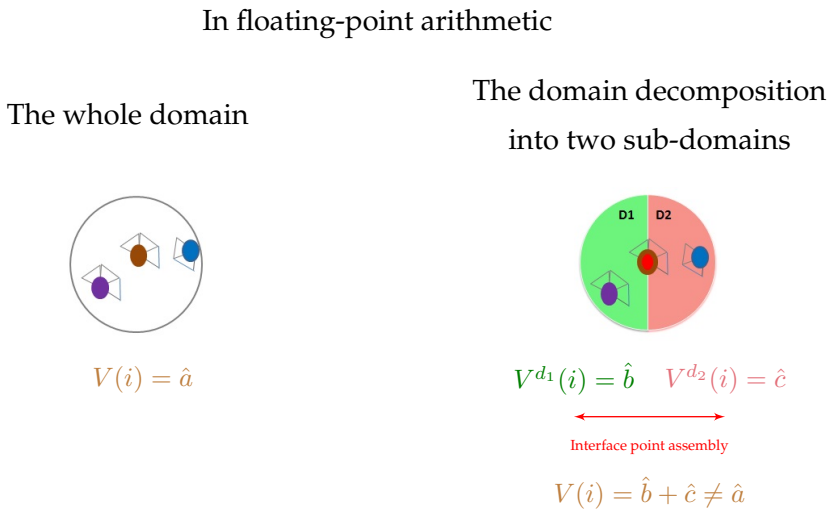
In this section, we detail the effects of the parallelism which yield to the openTelemac non-reproducibility. These impacts occur in both the building and the solving phases. The sources of non-reproducibility is the finite element assembly and the dot product. In the following we explain why these computation results become different when the number of computing unit varies.

4.6.1 Interface node assembly: the floating-point case

As previously mentioned in Section 4.4.2, the parallel resolution relies on domain decomposition that introduces inner and interface mesh nodes. The latter belong to a common boundary between several sub-domains and are shared between several computing units. The interface node assembly is one of the main significant differences between the sequential and the parallel resolutions. We explain how it greatly affects the numerical reproducibility.

Previously, we explain how in the parallel execution the interface nodes are assembled in order to compute the global value of the node. In Figure 4.3 we mentioned that in exact arithmetic the assembled value is the same than the sequential value. In floating-point arithmetic this equality is not satisfied. Figure 4.8 illustrates the same instance of node i in the domain which becomes an interface point in the decomposition into two sub-domains. Here, the sequential value of the node i is $V(i) = \hat{a}$ which suffers from rounding errors. In the parallel case, the two sub-domains d_1 and d_2 compute respectively V^{d_1} and V^{d_2} from the partial values at the interface node i. Each local computation also suffers from rounding errors and return: $V^{d_1}(i) = \hat{b}$ and $V^{d_2}(i) = \hat{c}$. The two sub-domains communicate to exchange their value and accumulate them also producing another rounding error. Finally, because of this different rounding error propagation, the required equality is not realized: $\hat{b} + \hat{c} \neq \hat{a}$.

FIGURE 4.8: The transformation of one node i into an interface point in the domain decomposition into two sub-domains. A communication and a reduction are necessary to obtain the global value of i. Compared to Figure 4.3, this case suffers of rounding errors because of the floating-point arithmetic.



Generally, for any arbitrary vector V which is extracted from the linear system and for one interface node i that belongs to k sub-domains, $V^{d_k}(i)$ is one of the contribution of V in the sub-domain d_k at the interface node i (the computation of V^{d_k} only includes quantities related to d_k). Communications between the sub-domains $d_1, \ldots, d_k$ yield the final value $V(i)$ at the node i as the following reduction which occurs for every interface node:

$$V(i) = \sum_{\text{subdomains } d_k} V^{d_k}(i). \tag{4.11}$$

In practice here, every sub-domain uses a local table that defines its communication scheme. For instance the sub-domain d_k knows how much, which i and to which $d_{k'}$, it has to send $V^{d_k}(i)$ and to receive $V^{d_{k'}}(i)$. For a given i, the openTelemac implementation of this communication scheme introduces different, but statically defined, accumulation order with respect to the sub-domains. For instance when $p = 3$ sub-domains, the d_0's computation of $V(i)$ with Relation (4.11), denoted as $V(i)|_{d_0}$, is:

$$V(i)|_{d_0} = V^{d_0}(i) + V^{d_1}(i) + V^{d_2}(i),$$

while the d_1 and d_2 ones are:

$$V(i)|_{d_1} = V^{d_1}(i) + V^{d_0}(i) + V^{d_2}(i),$$

and

$$V(i)|_{d_2} = V^{d_2}(i) + V^{d_0}(i) + V^{d_1}(i).$$

Each reduction is statically ordered according to one increased numbering of the neighboring sub-domains. Hence the floating-point computed $V(i)|_{d_k}$ may differ over the sub-domains d_k when the number of computing units p varies. Nevertheless this static strategy ensures the numerical reproducibility between repeated interface point assemblies for a given number of sub-domains p.
We emphasize this important property already introduced in the current version of openTelemac. For instance, in Tomawac, which suffers only from the assembly as the only source of non-reproducibility, consecutive runs with a same number of computing units, *i.e.* with a given sub-domain decomposition will return the same simulation up to the last bit. We note that is not the case with the classic dynamic reduction which is another source of non-reproducibility in Telemac-2D, for instance, where even two consecutive runs with a same number of computing units are not reproducible.
To recover the solution continuity between the sub-domains, *i.e.* $V(i)|_{d_k} = V(i)|_{d'_k}$, openTelemac introduces one more communication step that shares the maximum value of every $V^{d_k}(i)$; this choice is justified by physical reasons in [25].
In theory the assembling of the interface nodes is applied at the end of the building step. In openTelemac because of many optimizations, like EBE storage and the use of wave equation, the steps of building and solving are merged, as mentioned in Section 4.1.1. So even in the solving step, the interface node assembly is applied and specially at each iteration of the conjugate gradient in the matrix-vector product (see bold product in Algorithm 4.2).

The accumulation (4.5) has the same order with respect to el for the inner nodes in the sequential and the parallel cases. Nevertheless, a given inner node i may become one interface node in another domain decomposition, *i.e.* when the number p of computing units varies. Hence this type of nodes suffers from different error propagation.

4.6.2 The dot product

The dot product in openTelemac suffers from two sources of non-reproducibility.

1. ***Dynamic reduction and partial value differences.*** In a parallel simulation, the dot product of the whole domain is computed partially by each sub-domain, then sum over all the sub-domains by a collective communication. This dynamic reduction is actually processed in MPI with a non-deterministic order. So for the same input, the results may differ as exhibited with Figure 2.6. That leads to non-reproducibility for 2 successive runs for the same number of computing units.
   In addition when the number of sub-domains varies, the error propagation varies (partial dot products vary) and yield differences in the final result too.
   The following technical point has to be taken into account since it will introduce another source of differences between the sequential and the parallel simulations.

2. ***Weighted dot product.*** The dot product is computed for the whole domain on vectors that are assembled at the interface node. So if we compute the dot product of two vectors x and y on each sub-domain, the value of the interface node will be computed n times, where n is the number of sub-domains that share i. For this case, the mostly used solution is to calculate local weighted dot product.
   For instance, let us consider 3 sub-domains d_1, d_2, d_3 which contain their inner nodes and that share the interface node i. Let (x_{d_1}, y_{d_1}), (x_{d_2}, y_{d_2}), (x_{d_3}, y_{d_3}) be 3 pairs of vectors respectively defined on these sub-domains. The locally weighted dot product (x, y) is:

$$\begin{aligned} d_1 &: \sum_{\text{inner nodes } j_{d_1}} x(j_{d_1}) \cdot y(j_{d_1}) + \frac{1}{3} x(i) \cdot y(i), \\ d_2 &: \sum_{\text{inner nodes } j_{d_2}} x(j_{d_2}) \cdot y(j_{d_2}) \ + \frac{1}{3} x(i) \cdot y(i), \\ d_3 &: \sum_{\text{inner nodes } j_{d_3}} x(j_{d_3}) \cdot y(j_{d_3}) \ + \frac{1}{3} x(i) \cdot y(i). \end{aligned} \tag{4.12}$$

   Each division by the number of sub-domains (except for power of 2

cases) introduces a rounding error which will be different when varying the number of the computing units. This introduces another source of non-reproducibility which will be corrected in Section 5.2.2.

4.7 Conclusion

In this chapter we described the main steps of the finite element method. We presented the two assembly steps, the inner node and the interface node ones when parallel simulations rely on domain decomposition. We detailed the optimized EBE matrix storage and its corresponding matrix-vector product. We illustrate the non-reproducibility of two modules in openTelemac: Telemac-2D and Tomawac. The FE solving step applies the conjugate gradient algorithm. We noticed that EBE storage merges the assembly and the solving steps.

We gave a detailed analysis of the differences between the exact and the floating-point computation of the FE process. This analyze allows us to identify the following sources of numerical non-reproducibility.

— The interface node assembly: the generated rounding errors differ when the number of sub-domains varies. This source appears in the two studied modules. In Tomawac, it is produced when building the second members of the system. In Telemac-2D, it appears in the building phase of the second members and of the diagonal of the linear system matrices . It is also produced during the EBE matrix-vector product in each iteration of the conjugate gradient.

— The MPI dot product: MPI library introduces a non-deterministic order of reductions. This source appears only in the solving phase (conjugate gradient) in Telemac-2D. In Tomawac, the solved system is diagonal so no global dot product impact this simulation.

These different sources of non-reproducibility influence the parallel implementation of the building and/or the solving phases in openTelemac. In next Chapter 5, we detail how to modify these FE simulations to recover the reproducibility within these modules.

Chapter 5

Toward reproducible simulations within openTelemac

In Chapter 4 we detail the principle of finite element method and how the parallel simulation with floating-point arithmetic introduces failures of the numerical reproducibility of the results. In this chapter, we define how to modify these computations to recover the reproducibility for the two studied test cases in Tomawac and Telemac-2D. The main methods that improve numerical reproducibility have been presented in Chapter 3.
In Section 5.1 we compare three methods applied to a Tomawac test case. These methods are compensation, Demmel and Nguyen reproducible sum and integer conversion. We conclude that compensation is the more efficient solution in this case. This motivates that, in Section 5.2 we defined and apply compensation to a Telemac-2D test case. This module is more complex than Tomawac and provides a global view of how to the finite element computation is performed. Finally, in Section 5.3 we compare the running time extra-cost of these corrections for both studied cases.

5.1 A reproducible Tomawac module

In Tomawac, the element finite assembly (4.5) and the interface node's one (4.11) are applied to only build second members. This step is the unique source of non-reproducibility which is due to different rounding error propagation at the interface nodes when varying the number of computing units, as detailed in Section 4.6.1. This was the first identified source of non-reproducibility along this thesis work. It allows us to start implementing some of the methods presented in Chapter 3. It also gives us an ideal experimental case to compare

their efficiency and also to evaluate how easily these methods can be applied in a more general scope than the one for which they were published. Comparison is ideal here since: i) the modifications are restricted to the assembly steps which is produced by only 6 subroutines in the code. ii) the running time measure suffers from little or even no bias because it tests only one modified part (the assembly steps). We propose to implement and evaluate three techniques that provide reproducibility of this assembly step. Since here this latter is not ill-conditioned, the compensated summation (Section 3.1.2) yields an accurately rounded accumulation, and so a reproducible computation. Another choice are the recent Demmel and Nguyen reproducible sums (Section 3.2.1)[15]. The latest openTelemac release also introduces integer conversions (Section 3.2.2) and so provides exact accumulations [51].

We use the following notations to measure the accuracy and the reproducibility of these solutions. These two measures were illustrated with Figure 2.8. A^s denotes a sequential algorithm and A^p its p-parallel one. Original$A$ and Reprod$A$ respectively denotes the original algorithm and its modified version. Using $\max_{rel}(A_1, A_2) = |A_1 - A_2|/|A_2|$, the measure of the ReprodA^p accuracy compared to the original sequential OriginalA^s is:

$$\text{acc} = \max_{rel}(\text{Reprod}A^p, \text{Original}A^s),$$

while reproducibility between a sequential and a parallel execution is:

$$\text{rep} = \max_{rel}(\text{Reprod}A^p, \text{Reprod}A^s).$$

Here the A algorithm is the assembly step. We recall that the sequential FE assembly step at a node i computes

$$\text{Relation (4.5):} \quad V(i) = \sum_{el=1}^{nel} W_{el}(i).$$

The parallel one starts with a similar step that yield $V^{d_k}(i)$ for the inner nodes

of the sub-domain d_k, and continues for the interface nodes i for all sub-domain that contains i with

$$\text{Relation (4.11):} \quad V(i) = \sum_{\text{subdomains} d_k} V^{d_k}(i).$$

As mentioned, in openTelemac the elementary accumulation (4.5) has the same order with respect to el for inner nodes in the sequential and the parallel cases. Nevertheless, a given inner node i may become one interface node in another domain decomposition yielding to a different rounding error propagation. Hence we must provide a reproducible assembly to every mesh node, both in the sequential and in the parallel simulations.

5.1.1 Compensated computation

Compensated assembly are easy to derive from inner and interface node assembly steps: these different sets of rounding errors are taken into account such that the remaining rounding error in a compensated $V(i)$ does not depend anymore of the number of sub-domains. For every vector V we introduce the $[V, E_V]$ pair where E_V accumulates the generated rounding error during the computation of V. Here compensation consists in accumulating and propagating every generated rounding error until Relation (4.11) is applied to assemble the interface nodes. Hence, every vector V now comes from and goes along the computation with its accumulated rounding errors E_V, until the last reduction in (4.11) after which E_V compensates V.
This compensated finite element assembly Reprod$A$ modifies the classic accumulation (4.5) as:

$$[V(i), E_V(i)] = \text{ReprodAss}_{el=1,\ldots,nel} W_{el}(i), \tag{5.1}$$

where 2Sum (Algorithm 3.2) computes the rounding error of each node i:

$$(V(i), e_i) \quad = \quad \text{2Sum}(V(i), W_{el}(i)),$$

and $E_V(i)$ accumulates the errors e_i for all elements that include the node i. The compensated assembly of an inner node is:

$$V(i) + E(i).$$

The interface node assembly (4.11) now applies to the pairs $[V, E_V]$. For all sub-domains d_k that share the interface node i, Relation 4.11 is modified as:

$$[V(i), E_V(i)] = \bigoplus_{\text{subdomains } d_k} [V^{d_k}(i), E_V^{d_k}(i)], \tag{5.2}$$

which again uses 2Sum as follows:

$$(V(i), e_k) = \text{2Sum}(V(i), V^{d_k}(i)), \tag{5.3}$$

$$E_V(i) = E_V(i) + E_V^{d_k}(i) + e_k. \tag{5.4}$$

Step (5.3) computes $V(i)$ by accumulating $V^{d_k}(i)$ and also returns the generated rounding error e_k. Step (5.4) accumulates in $E_V(i)$ this e_k and the previous errors $E_V^{d_k}(i)$. Finally, compensation occurs after the last reduction of every interface node i to yield the whole vector V as:

$$V + E_V. \tag{5.5}$$

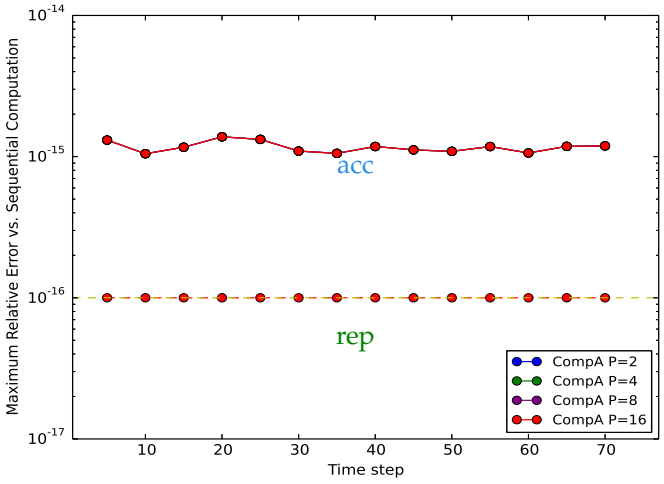
We emphasize that this compensation applies once to the vector of inner and interface nodes after the end of the interface node assembly, *i.e.* Relation (5.2). We already noted the very slight modification of the communications between the computing units that now exchange the pair of floating point values $[V^{d_k}(i), E^{d_k}(i)]$, and not only its first one. Nevertheless no more supplementary communication step is introduced here.

Similarity with double-double arithmetic, *e.g.* QD in [1], exists here since entries are also floating-point couples $[V, E_V]$ and computations update the error term E_V. Nevertheless no normalization (with 2Sum nor Fast2Sum) applies here to maintain the relative accuracy between V and E_V. No partial compensation neither applies before Relation (5.5).

Tomawac recovers its numerical reproducibility with this compensated assemblies. We consider the *Nice* test case presented in Section 4.5.2. Figure 5.1 exhibits that all parallel executions return the same simulation results up to the

computing precision rep for $p = 2, 4, 8, 16$. Since expected, compensation also improves the accuracy and the acc plot actually measures the original accuracy of the floating point assembly (OriginalA^s). The reproducible computation provides about one digit enhancement which is consistent with the error bound of the classical floating point accumulation, see Relation (3.2). Compensation does not introduced an important extra-cost, as will be exhibited in Section 5.3.

FIGURE 5.1: Compensated assembly. Tomawac (*Nice* test case, Mean frequency wave)



5.1.2 Reproducible sums

The main difficulty to integrate Demmel and Nguyen algorithm within the assembly step in Tomawac is the need for maxima of the inner point and the interface point contributions, respectively $\max_i W_{el}(i)$ in Relation (4.5), and one other maximum value for Relation (4.11). As we already mentioned it, the assembly loop in the Algorithm 4.1 consists in indirect accumulations. In this scope, ReprodSum and FastReprodSum introduces a significant volume of supplementary computations and communications.

One more previous run of the assembly double loop is necessary to identify the maximum values for every inner node i (and also generates some extra-storage on every computing unit). The interface node assembly is even more costly since every computing unit needs to use the same maximum value before computing both Relations (4.5) and (4.11). For every sub-domain d_k,

$\max_{i_k \in d_k} W_{el}(i_k)$ is first identified and reduced over the domain before computing every $V^{d_k}(i)$ with Relation (4.5). Then Relation (4.11) is finally computed after a last classical communication step of the $V^{d_k}(i)$. In a parallel execution of this method, sub-domains communicate two times. The first is to exchange the partial maximum values. The second is the interface node assembly Relation (4.11). Consequently, we will see in Section 5.3 how this penalizes the efficiency. For that the application of the algorithm "1-Reduction" remains an interesting perspective, because it does not compute the global maximum absolute value but each sub-domain uses its local one.

Nevertheless the reproducibility failure of the floating point assembly is thus fixed for all considered parallel executions. Figure 5.2 exhibits it for $K = 1$ and $K = 2$, where K is number of shrunks which provides more accuracy. Figure 5.3 presents the accuracy behavior of these solutions for $K = 1$ and 2 compared to the previous compensated assembly. As expected, the $K = 1$ case provides less accurate results, without being here significantly interesting even compared to the original floating point ones. It also illustrates that no difference appears for $K = 2$ between the ReprodSum based assembly and the accurate compensated one.

FIGURE 5.2: Reproducible based assembly.
Tomawac (*Nice* test case, Mean frequency wave)

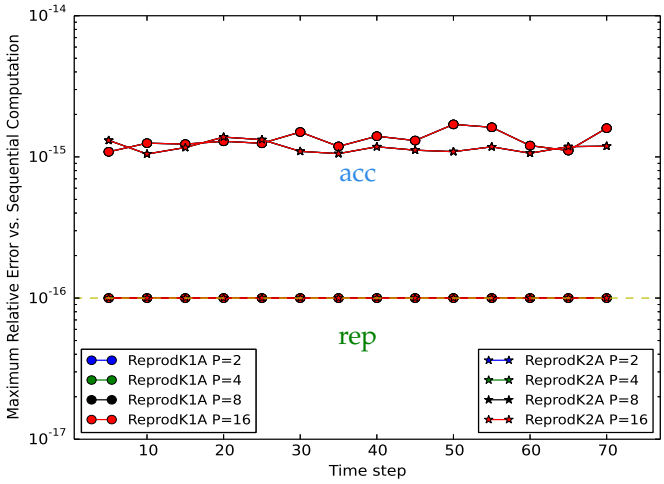
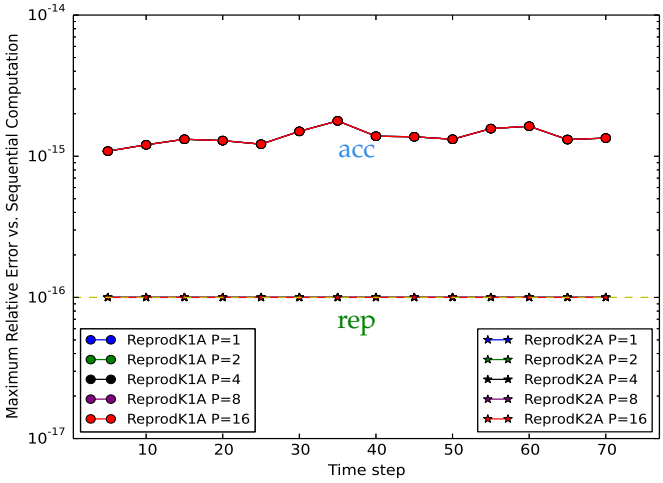


FIGURE 5.3: Accuracy of the reproducible based assembly for $K = 1, 2$ compared to the compensated one.
Tomawac (*Nice* test case, Mean frequency wave)



5.1.3 Integer conversion

The last solution towards a reproducible assembly step is to benefit from exact integer arithmetic as already introduced in Section 3.2.2. Such kind of integer conversion is actually provided with one of the latest Telemac release [51], v 7.0. We now describe and discuss this implementation.
It consists in one 8 byte integer conversion of every floating point value concerned by assembly computations. Integer accumulations are exact as long as no overflow occurs and their conversions back to floating point values yield reproducible computed results. This 8 byte integer conversion is simple. We denote INT K8 a 8 byte integer value and the function HUGE(INT K8) returns the largest number of this integer type. In Fortran 90 for instance, this type of signed integer has a range for from -2^{63} to $2^{63} - 1$. A floating-point vector V of size np, is converted into the integer vector IV by:

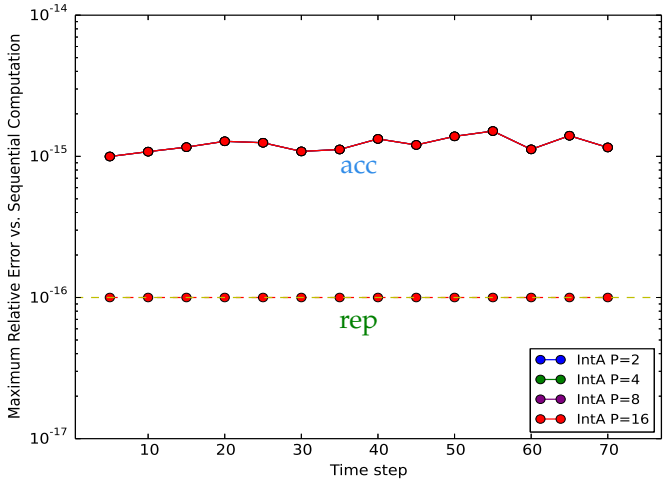
$$IV = NINT(V \times Q_8(V, np)), \tag{5.6}$$

where NINT( ) rounds a real number to the nearest whole number. The scaling factor Q_8 is chosen to avoid the overflow while summing V. We have [51]:

$$Q_8(V, np) = \frac{\text{HUGE}(INTK8)}{np \times \max(|V|)}.$$

This conversion introduces two approximation levels. The first one comes from the difference between the floating-point discretization and the integer one. We illustrate it with one example in decimal. Let $np = 10$ and $\max(|V|) = 10^6$. Let $IV \in [0, 10^8[$ (componentwisely) thus HUGE$(IV) = 10^8$. The conversion $IV = V \times Q$ applies here with $Q = 10$. So every floating point value with more than one fractional digit suffers from a 10^{-1} absolute error of conversion. This corresponds to a relative error from 10^{-3} to 10^{-7} for the whole V range that acts as data errors before the summation. Even if the sum is exactly computed, cancellation is well known to magnify the data errors. Hence no accuracy bound better than 10^{-4} can be expected after such (fancy) conversion.
The floating point evaluation of the scaling factor Q_8 introduces a second approximation level. Indeed evaluation (5.6) suffers from rounding errors in the Q_8 evaluation, in the multiplication and then in the division on the way back to floating point values. Of course, these errors are smaller than the previous discretization ones but occur in every conversion (5.6).
Nevertheless changing Q_8 to its next larger power of the base, *i.e.* to 2^α with $2^{\alpha-1} < Q_8 \leq 2^\alpha$, in binary arithmetic, could be proposed as a slight modification that avoids this approximation error. In this case, it is clear that no difference appears between the exact floating-point sum and its integer counterpart as long as $V \times 2^\alpha$ is an exact $INT\ K8$ value. There is no difficulty to apply this integer conversion to the assembly process in Tomawac. The pre and post conversions are applied on-the-fly while accumulating inner nodes in Relation (4.5). The interface assembly step only needs to communicate the integer values to evaluate Relation (4.11).

FIGURE 5.4: Integer assembly.
Tomawac (*Nice* test case, Mean frequency wave)



Integer conversion yields the expected reproducibility of the simulation as exhibited with Figure 5.4. It does not appear a significant loss of accuracy compared to the previous solutions. The Tomawac computation is a favourable case for this integer conversion strategy. Indeed few terms (about 8 terms for each node) are accumulated for every node i and their ranges remain small enough to avoid suffering a lot from the discretization error of the integer conversion.

5.1.4 A reproducible Tomawac

Assembly is one of the core step in every finite element resolution. Tomawac is simple enough to successfully only focus on this assembly step. We have been able to recover its numerical reproducibility with the three presented solutions. In this context we can summarize their actual feasibility as follows. The compensated solution appears to be the easiest one to apply and provides accurate results for a low computing extra-cost. The integer conversion provided in Tomawac is also easy to derive and introduces a low extra-cost. Even if it is definitely not redhibitory here, this solution may introduce a risky loss of accuracy in other cases. A careful error analysis should be performed before introduced in other applications. The solution that uses the recent reproducible sum algorithms is generally efficient, but in our case it applies less

easily and introduces a significant communication extra-cost.

5.2 A reproducible Telemac-2D module

In Chapter 4, we have described the main steps of the finite element computation implemented in Telemac-2D and we have exhibited the non-reproducibility sources. In what follows we distinguish two phases: the building of the linear system and its solving. We detail how to apply error-free transformations and compensated algorithms in the involved computation. We also highlight where we have to compensate and where the rounding errors have no more effect onto the numerical reproducibility.

5.2.1 Building the linear system

As detailed in Chapter 4, the building phase of System (4.2),

$$\begin{pmatrix} A_1 & 0 & 0 \\ 0 & A_2 & 0 \\ 0 & 0 & A_3 \end{pmatrix} \begin{pmatrix} H \\ U \\ V \end{pmatrix} = \begin{pmatrix} C_1 \\ C_2 \\ C_3 \end{pmatrix},$$

includes the two following non-reproducible sequences. They derived from the different propagation of the rounding errors at the interface nodes when varying the number of sub-domains. Hence interface node assembly (4.11) or (5.5) appears differently during the simulation. Table 5.1 exhibits how this interface node assembly is merged within these phases in the resolution work flow. Parentheses in Table 5.1 describe the corresponding component dependencies.

TABLE 5.1: Transformation workflow in System (4.2) and related component dependencies in parentheses.

Unknowns	H	U	V
System building:			
Finite element assembly (4.5) & Algebraic transformations (4.3)	$A_2, A_3, C_1(A_2, A_3)$ $A_1(A_2, A_3)$	$C_2(H)$	$C_3(H)$
Interface node assembly (4.11)	A_2, A_3, C_1	C_2	C_3
System solving:			
	Conjugate gradient $A_1, C_1 \rightarrow H$	$C_2, A_2 \rightarrow U$	$C_3, A_3 \rightarrow V$
Interface node assembly (4.11)	In each iteration $\boldsymbol{A_1 d}$		

In the following we describe the computations which are modified.

(a) ***The finite element assembly steps (Sections 4.2.2 and 4.6.1).*** In a sequential resolution, Relation (4.5) returns $V(i)$ for every node i. In a parallel resolution, the accumulation is distributed over the sub-domains d_k and its computation differs between inner nodes and interface nodes. Relation (4.5) returns $V(i)$ for every inner node i, while for one interface node it only computes the partial contribution $V^{d_k}(i)$. To obtain the final value of these interface nodes, an interface node assembly is processed as defined in Relation (4.11).

These two assemblies are applied to build System (4.2), *i.e.* the second members C_1, C_2, C_3, the diagonal of the matrix A_1 and the diagonal matrices A_2, A_3. Since the matrix A is EBE stored, it is also applied to the extra-diagonal part of the matrix-vector product $\boldsymbol{A_1 d}$ in the conjugate gradient iterations (see Section 5.2.2). Let us remark that all these quantities are vectors.

Reproducible assembly: We apply the compensated assembly defined in Section 5.1.1. As in Tomawac $[V, E_V]$ vector pairs are introduced. So the Relation (4.5) is written as Relation (5.1). The compensated assembly

of an inner node i is:

$$V(i) + E(i),$$

The interface node accumulation (4.11) now applies to pairs $[V, E_V]$ to become Relation (5.2). For all sub-domains $d_k \in D_i$ that share the interface node i, the Relation (5.2) is applied. Finally, compensation occurs after the last reduction of every interface node i to yield the whole vector V as

$$V + E_V.$$

(b) ***The algebraic transformations (4.3).*** These operations are applied to the assembled vectors V of size np. They include componentwise vector operations as products, additions, scalar scalings of vectors, *etc.* The rounding error propagation of these operations differs only when the concerned vector(s) is (are) not assembled at the interface nodes, since different rounding errors are produced when varying the number of subdomains. Otherwise they will be the same and so do not affect the reproducibility.

Reproducible algebraic transformations: As already mentioned, the vector V can be modified by algebraic operations between the FE assembly and the interface point assembly steps. Consequently its error term E_V has to be propagated too. For that purpose, all algebraic operations in BIEF must be modified. This modification propagate E_V (when it exists) and takes into account the rounding errors of the operation. For example, the Hadamard product $V = X \circ Y$ between two vectors X and Y ($V(i) = X(i) \times Y(i)$) is now computed as the following pair:

$$[V, E_V] = [X, E_X] \circ [Y, E_Y], \tag{5.7}$$

with:

$$\begin{aligned} (V(i), e(i)) &= \text{2Prod}(X(i), Y(i)), \\ E_V &= X \circ E_Y + Y \circ E_X + e, \end{aligned}$$

using the classic 2Prod error free transformation [14] or the 2MultFMA

[48] which are presented in Section 3.1.2. Others operations in (4.3) derive similarly. More implementation details are provided in Section 6.3.

Tables 5.2, 5.4 and 5.5 will help the reader to gradually follow the reproducibility enhancement of System (4.2) during the correction of the building and the solving phases. For the moment, its first column exhibits that all the components suffer from non-reproducibility in the original Telemac-2D computation.

Now, the second column of Table 5.2 concludes the building correction phase, *i.e.* the previously described modifications of the finite element assembly, the interface node assembly and the algebraic transformations. These modifications provide the reproducibility of the diagonal matrices and of the vectors.

Let us remark that A_1 remains non-reproducible because it is not yet assembled at the interface nodes. This correction will be integrated during the solving phase in next Section 5.2.2.

TABLE 5.2: Reproducibility enhancement steps of System (4.2) components in *gouttedo*. After the building phase corrections (to be continued $\cdots$).

	original	after the building phase
A_1	✗	✗
A_2	✗	✓
A_3	✗	✓
C_1	✗	✓
C_2	✗	✓
C_3	✗	✓
H	✗	✗
U	✗	✗
V	✗	✗

5.2.2 The linear system resolution

Matrix-vector product and dot product are the main floating-point computations in this solving phase. As already described in Section 4.2.3, EBE storage leads to efficient matrix-vector products in the finite element context: it avoids to assemble the whole matrix to compute a matrix-vector product and it also reduces the matrix memory print. Hence EBE storage introduces a specific processing of the matrix-vector product, merging product and assembly.

As already mentioned, the simulation solves three subsystems (4.2). The H

system is solved with the conjugate gradient algorithm, see Algorithm 4.2, with a diagonal preconditioning detailed in Section 4.3.1 and recalled here. Dot products are denoted in parentheses and matrix-vector products denoted in bold as $\boldsymbol{Ad}$:

Initialization:

$$r^0 = \boldsymbol{Ax^0} - b,\ d^0 = r^0,\ \rho^0 = \frac{(r^0, r^0)}{(d^0, \boldsymbol{Ad^0})}$$
$$x^1 = x^0 - \rho^0 d^0$$

Iterate: until stopping criterion (4.10):

$$r^m = r^{m-1} - \rho^{m-1}\boldsymbol{Ad^{m-1}}$$
$$d^m = r^m + \frac{(r^m, r^m)}{(r^{m-1}, r^{m-1})} d^{m-1}$$
$$\rho^m = \frac{(r^m, d^m)}{(d^m, \boldsymbol{Ad^m})}$$
$$x^{m+1} = x^m - \rho^m d^m$$

Thanks to the EBE storage, a matrix M is decomposed as one assembled diagonal matrix D and non-assembled extra-diagonal matrices X_{el}, as detailed in Section 4.2.3.

In the parallel resolution with sub-domain decomposition, two different sources affect the reproducibility of the conjugate gradient.

1. One causes from the EBE storage that governs the EBE matrix-vector product $\boldsymbol{Ad}$. Hence, this computation includes an interface node assembly (4.11) at each iteration.
2. The other source is the dot product. Each sub-domain computes a partial dot product but the conjugate gradient needs the result of the whole dot product. Hence a collective communication occurs with a non-deterministic computation order.

As shown in Table 5.1, when the H system is solved, the second members C_2 and C_3 of the U and V systems are computed and assembled at the interface node with Relation (4.11). Finally these two diagonal systems yield U and V.

A reproducible linear system resolution

Now we detail how to recover the reproducibility of the three subsystems included in System (4.2). The main computation step solves $A_1 H = C_1$ with the conjugate gradient Algorithm 4.2.

As already mentioned in Table 5.2, the second member C_1 is reproducible since the building phase. This is not the case for matrix A_1 that has not been assembled at the interface nodes and so that may differ from one sub-domain to another. However, A_1 is built together with its accumulated errors E_{A_1} explained in Section 5.2.1.

We recall that the EBE matrix-vector product is denoted in bold in Algorithm 4.2. As detailed in Section 4.2.4, it verifies:

$$M \cdot V = D \circ V + \sum_{el=1}^{nel} X_{el} \cdot V_{el}.$$

Now we present how to modify the product $\boldsymbol{Ad}$ to recover its reproducibility. At this point, we know $[A_1, E_{A_1}]$. The first term in $M \cdot V$ is the Hadamard product $D \circ V$. Now, the diagonal part D is associated with its errors E_D, so we compute $[DV, E_D] \circ V$ as in Relation (5.7).
The second term is a finite element assembly applied to the extra-diagonal terms of A_1. This assembly is now computed with the compensated one (5.1). Then, these two pairs are componentwisely summed to obtain the $M \cdot V$ product and its associated rounding errors. In the parallel case, the interface nodes are assembled with Relation (5.2). The last step is the compensation $MV + E_{MV}$. This modified matrix-vector product is applied to $\boldsymbol{Ad}$ in the conjugate gradient iterations and become reproducible.

The second source of non-reproducibility in the conjugate gradient is the dot product (in parentheses). This operation suffers from two issues which are detailed in Section 4.6.2:

1. the locally weighted dot product,
2. its computation relies on a dynamic parallel reduction.

The first issue is easily corrected changing the weighting coefficients to eliminate the produced rounding errors. For instance, let us consider 3 sub-domains d_1, d_2, d_3 which share an interface node i and for instance $d_1 > d_2, d_3$. Let

(x_{d_1}, y_{d_1}), (x_{d_2}, y_{d_2}), (x_{d_3}, y_{d_3}) be 3 pairs of vectors respectively defined on these sub-domains. The locally weighted dot product (x, y) (4.12) becomes:

$$\begin{aligned} d_1 &: \sum_{\text{inner nodes } j_{d_1}} x(j_{d_1}) \cdot y(j_{d_1}) + 1 \cdot x(i) \cdot y(i), \\ d_2 &: \sum_{\text{inner nodes } j_{d_2}} x(j_{d_2}) \cdot y(j_{d_2}) \ + 0 \cdot x(i) \cdot y(i), \\ d_3 &: \sum_{\text{inner nodes } j_{d_3}} x(j_{d_3}) \cdot y(j_{d_3}) \ + 0 \cdot x(i) \cdot y(i). \end{aligned} \tag{5.8}$$

This weighted dot product is introduced to ensure that each interface node is only counted once. Our proposed modification is simple: now it is counted once for the sub-domain with largest rank number (d_1 here).

The second issue is produced by the dynamic reduction which is presented in Section 2.2.1. It is corrected by using a compensated dot product that produces result accurate as if computed in twice the working precision. By using dot2, Algorithm 3.8, we easily derive a parallel compensated dot product dot2_rep that yields reproducibility *in our context*: here the dot product condition numbers are reasonably smaller than $1/\mathbf{u}$ and hard rounding cases are rare. Indeed, for too ill-conditioning product another algorithms should be used (see Section 3.1.2).

Sequential dot2 accumulates both the dot product and the generated rounding errors (by the additions and the multiplications) and finally compensates them together. In our parallel implementation, each computing unit computes its local dot products and its generated rounding errors. The computation of this pair [value, error] is denoted dot2_rep and detailed in Table 5.3. These local pairs are exchanged and accumulated by every computing unit using Sum2, Algorithm 3.7.

TABLE 5.3: Dot products as implemented in the original and the reproducible openTelemac

Dot product	Sequential run	Parallel run
Original version	$r = \mathsf{dot}(x, y)$	$r_p = \mathsf{dot}(x, y)$ $r = \mathsf{all\_reduce}(r_p)$
Reproducible version	$r = \mathsf{dot2}(x, y)$	$[r_p, e_p] = \mathsf{dot2\_rep}(x, y)$ $\mathsf{all\_gather}(r_p, e_p)$ $r = \mathsf{Sum2}(r_p, e_p)$

Figures 5.5a and 5.5b plot the relative differences compared to a sequential MPFR execution (1000 bits) of the original and compensated parallel dot products respectively. Our implementation uses the latter which yields expected reproducibility.

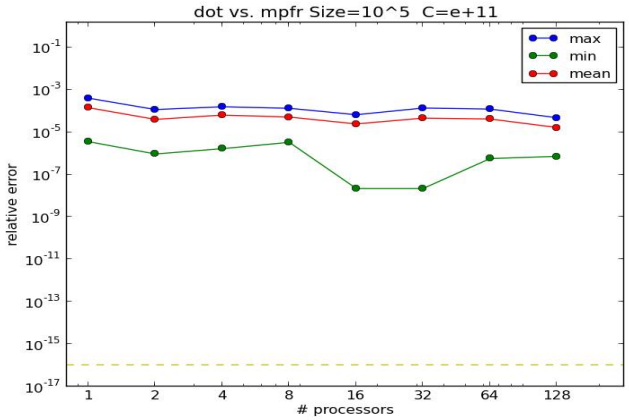
We now have reproducible versions of the matrix-vector product (bold) and of the dot product (parentheses) introduced in the conjugate gradient. This is sufficient to compute a reproducible output H.

Table 5.4 is the progression of Table 5.2 after these previous corrections. The new column (the third one) illustrates that after these corrections the H component becomes reproducible. However, the A_1 component never become reproducible because it is not assembled at the interface nodes. But we need to compute its rounding errors to compensate the matrix-vector product, as detailed previously.

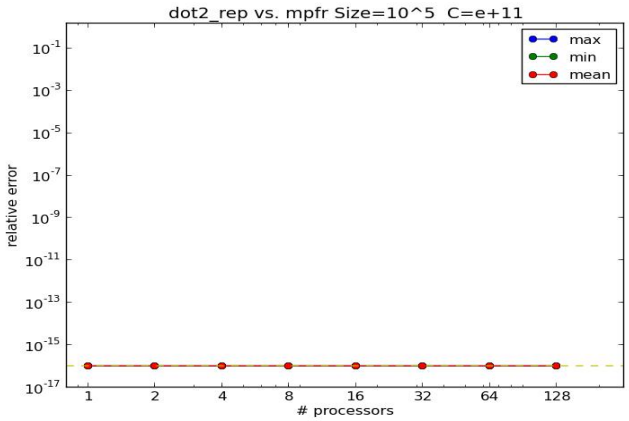
TABLE 5.4: Reproducibility enhancement steps of System (4.2) components in *gouttedo*. After the H solving phase corrections, (to be continued $\cdots$).

	original	after the building phase	after the H solving
A_1	✗	✗	✗
A_2	✗	✓	✓
A_3	✗	✓	✓
C_1	✗	✓	✓
C_2	✗	✓	✓
C_3	✗	✓	✓
H	✗	✗	✓
U	✗	✗	✗
V	✗	✗	✗

FIGURE 5.5: Relative differences in the original and the compensated parallel dot product algorithms (min, max, mean). x-axis: p processors (1...128). Horizontal dotted line : **u** and $cond \approx 10^{11}$ and $n = 10^5$.



(a) The original parallel dot product.



(b) The compensated parallel dot product.

It is important to note here that the conjugate gradient still suffers from rounding errors generated by the divisions and the other operations. Nevertheless, these errors are similar in the sequential and in the parallel simulations because the operations are applied to actually reproducible vectors *i.e.* to assembled vectors at the interface nodes.

Recovering reproducibility of the last two steps is now straightforward. The U and V diagonal subsystems depend on H which is reproducible now. The second members C_2 and C_3 are built (from H) and are assembled at the interface nodes before the resolution, see Table 5.1. The reproducible interface

node assembly, described in Section 5.2.1, applies here during this building phase. Hence the matrices and the second members A_2, C_2, A_3 and C_3 are now reproducible. This leads to the reproducible diagonal resolution of the U and V subsystems from similar algebraic transformations to those presented in Section 5.2.1.
Table 5.5 now displays in the fourth column a whole reproducible simulation.

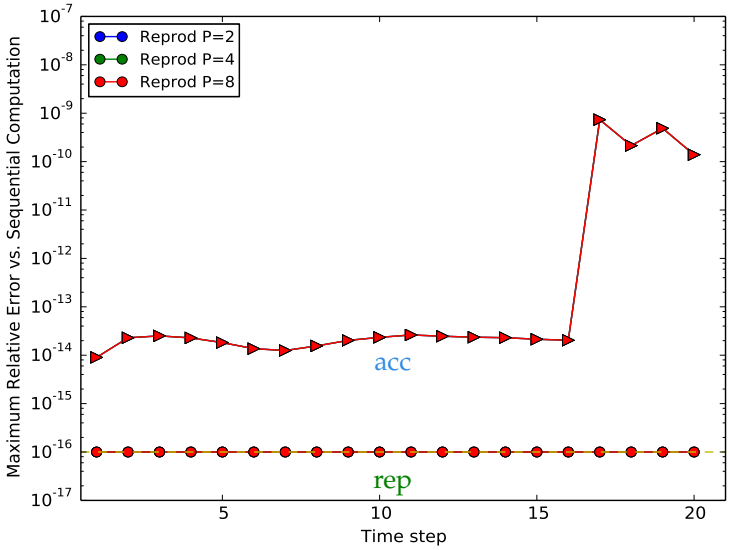
TABLE 5.5: Reproducibility enhancement steps of System (4.2) components in *gouttedo*. After the whole solving phase corrections.

	original	after the building phase	after the H solving	after the U, V solving
A_1	✗	✗	✗	✗
A_2	✗	✓	✓	✓
A_3	✗	✓	✓	✓
C_1	✗	✓	✓	✓
C_2	✗	✓	✓	✓
C_3	✗	✓	✓	✓
H	✗	✗	✓	✓
U	✗	✗	✗	✓
V	✗	✗	✗	✓

5.2.3 A reproducible Telemac-2D

We recover the numerical reproducibility of a Telemac-2D test case using the compensation techniques. Figure 5.6 displays the two measures of convincing reproducible results. The rep plot is the maximum relative error over the whole *gouttedo* domain between the compensated parallel simulation and the compensated sequential one. The number of processors varies: $p = 2$, 4, 8. All the plots are superposed and constant at the precision level. This illustrates the reproducibility of the compensated simulations. The second plot acc displays the maximum relative difference between the original sequential Telemac-2D simulation and the compensated ones. Relative differences varies from 10^{-14} to 10^{-10}. This validates the compensated simulations that are very similar to the original sequential Telemac-2D simulation. As already mentioned, this latter is considered by openTelemac developers as their reference simulation. Nevertheless since compensation provides more accuracy than a computation at the working precision, this curve certainly displays the loss of accuracy of the openTelemac developer's reference result.

FIGURE 5.6: Reproducibility rep of the compensated simulation and accuracy acc compared to the original Telemac-2D for the water depth in *gouttedo*.
x-axis: time steps ($1 \ldots 20 \times 0.2$ sec). y-axis: maximum relative difference. Number of processors: p = 2, 4, 8.



Figures 5.7, 5.8 and 5.9 now display the reproducible simulation of the *gouttedo* test case. Compared to the original ones Figures 4.4, 4.5 and 4.6 no more white spot appears between the sequential run and when varying the number of processors ($p = 2, 4, 8$), respectively.

FIGURE 5.7: *gouttedo* : Numerical reproducibility, no more white spot for the water depth values between the sequential (left) and a 2 processors run (right).
Time steps: 1,2, . . . ,7,8.

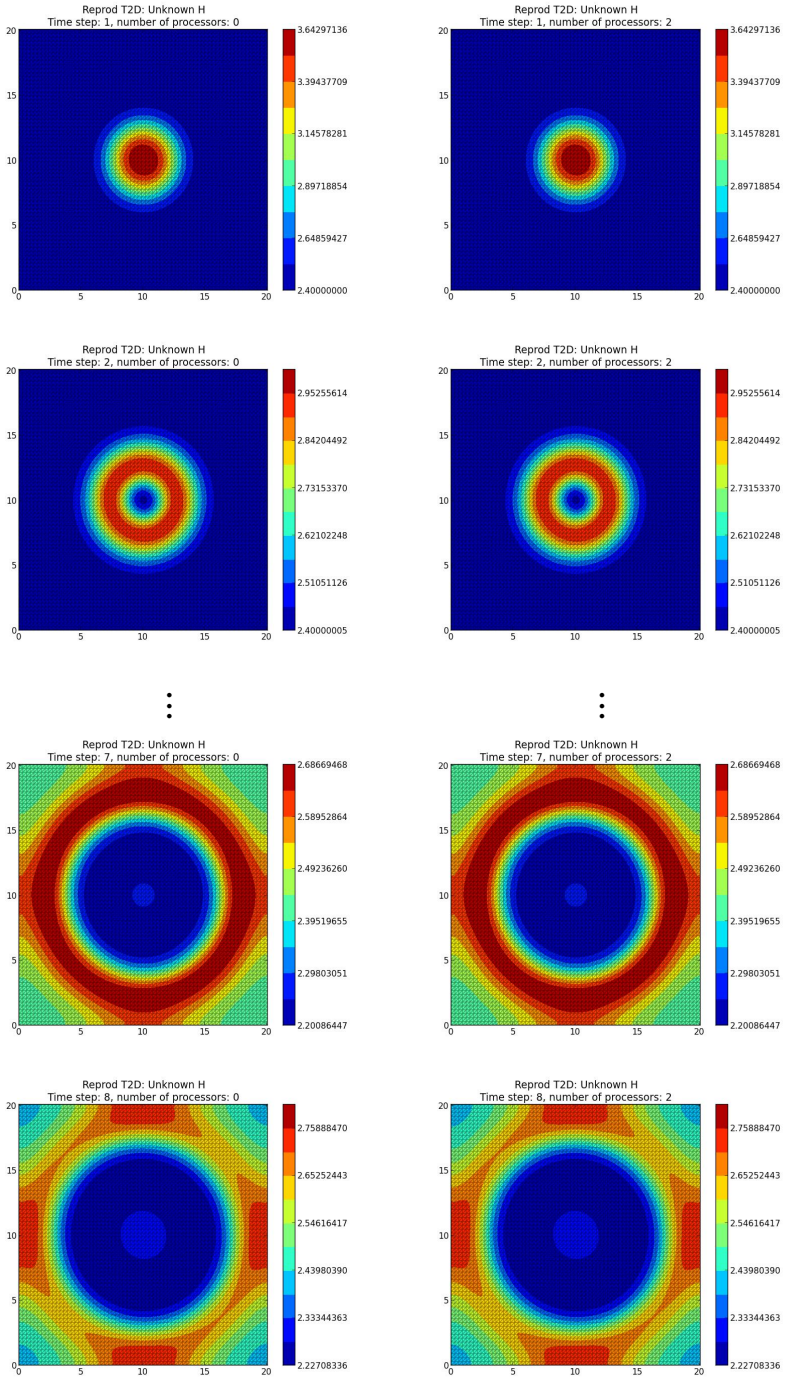


FIGURE 5.8: *gouttedo* : Numerical reproducibility, no more white spot for the water depth values between the sequential (left) and a 4 processors run (right).
Time steps: 1,2, . . . ,7,8.

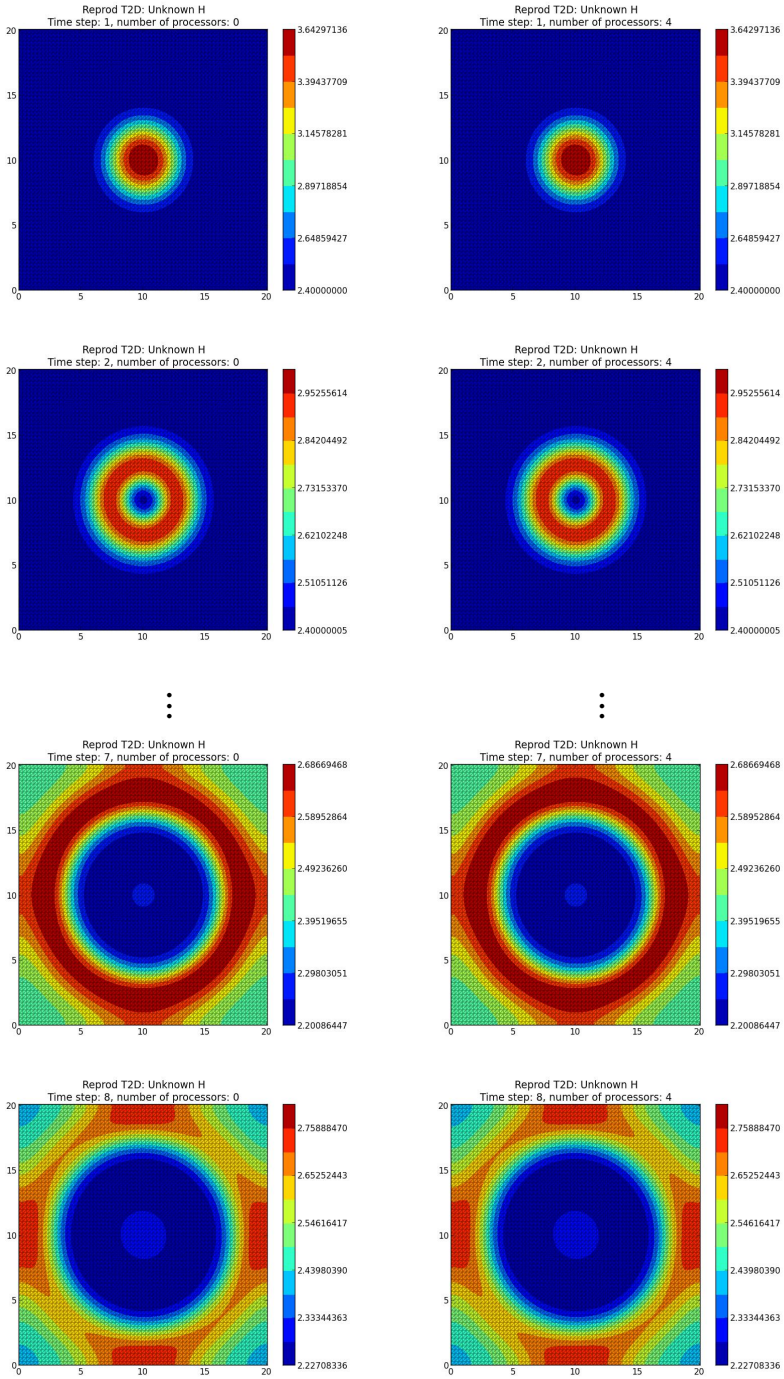
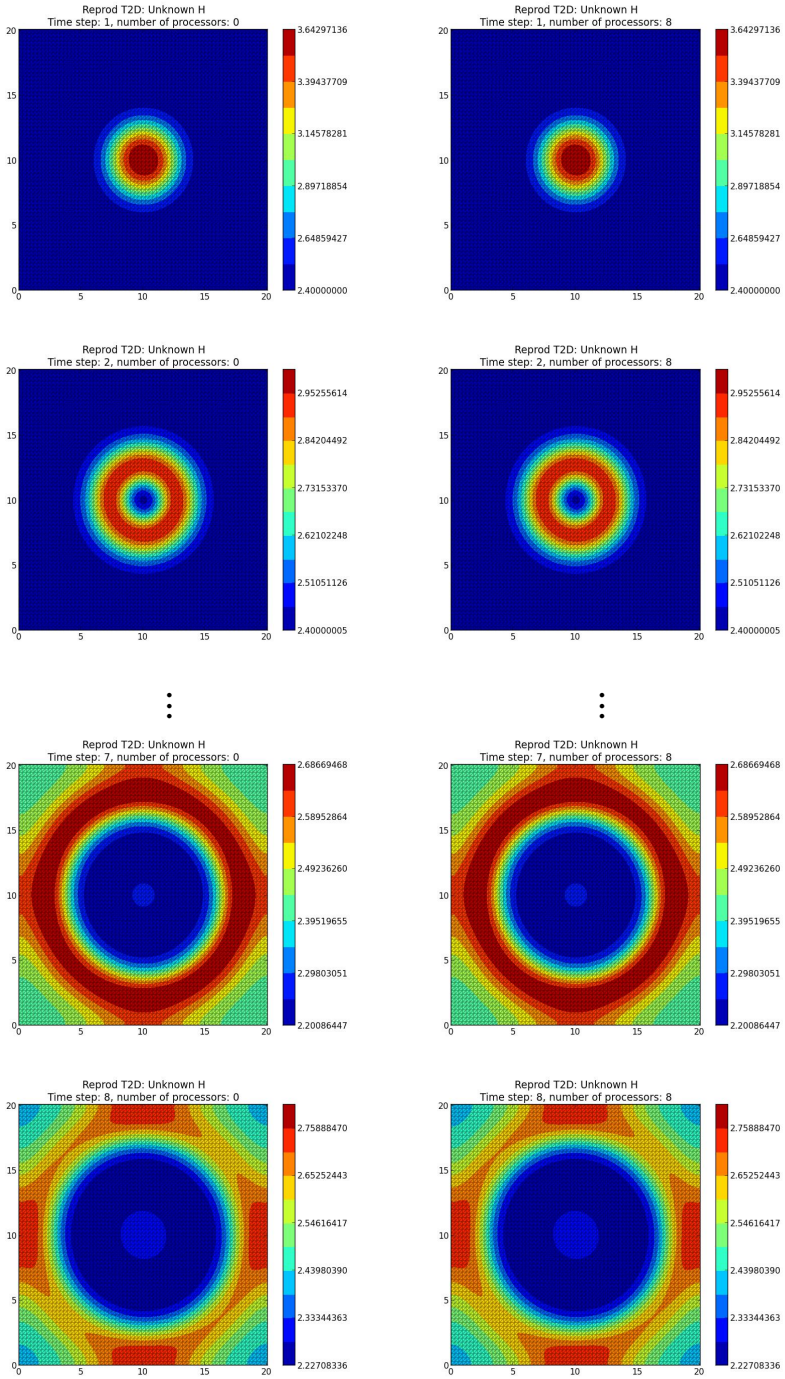


FIGURE 5.9: *gouttedo* : Numerical reproducibility, no more white spot for the water depth values between the sequential (left) and a 8 processors run (right).
Time steps: 1,2, . . . ,7,8.



5.3 The extra-cost of the proposed reproducibility

The running time performance of simulations is of course one important parameter to measure, specially for critical ones which may have to provide real-time results. In this thesis, we are not interested in the performance of the openTelemac simulator itself. Several works improve its performance and its parallelization are presented in [6], [19], [29]. What is interesting for us is to measure the extra-cost of the modifications we introduced to provide reproducible results, compared to the original code. This measure helps to choose between several methods that yield reproducibility.

Measuring reliable running time performance is a hard task. It can for instance suffers non-reproducibility itself [42], [22]. Indeed the running time of a binary program varies, even using the same dataset input and the same environment, as explained in Chapter 2. The operating system also introduces sporadic intrusions into the test environment that are difficult to take into account. The measure in a parallel environment is even less reliable because the non-deterministic events. So our further presented measures correspond to repeat 15 times each execution and to report the minimal measure.

An extra-cost analysis compared to the original versions is performed for the studied reproducible test cases in Tomawac and Telemac-2D. We measure the running time in cycles on the v7.2 version of openTelemac. We use the hardware counter RDTSC assembly instruction (ReaD Time Stamp Counter), that counts the number of cycles since a reset [33]. Tests run with the following hardware and software environment:

- socket: Intel Xeon E5-2660 2.20GHz (L3 cache = 20 M)
- 2 sockets of 8 cores each
- GNU Fortran 4.6.3, -O3
- OpenMPI 1.5.4
- Linux 3.5.0-54-generic

In openTelemac and generally in any simulation, a lot of read/write data occur at both the pre and post simulation steps. For the test cases considered here, it appears to be not significant to measure the whole simulation. Indeed the

extra-cost of our modifications is negligible compared to the complete simulation cost. So we limit our measure to the modified part compared to the original one.
Listing 5.1 illustrates the parallel measure process for the modified part of the code. In lines 1 and 5 we synchronized the processors to be sure that they are at the same level of the computation (since the beginning of the modifications until the end). Each processor returns its local time (line 8) and finally we chose the minimal one.

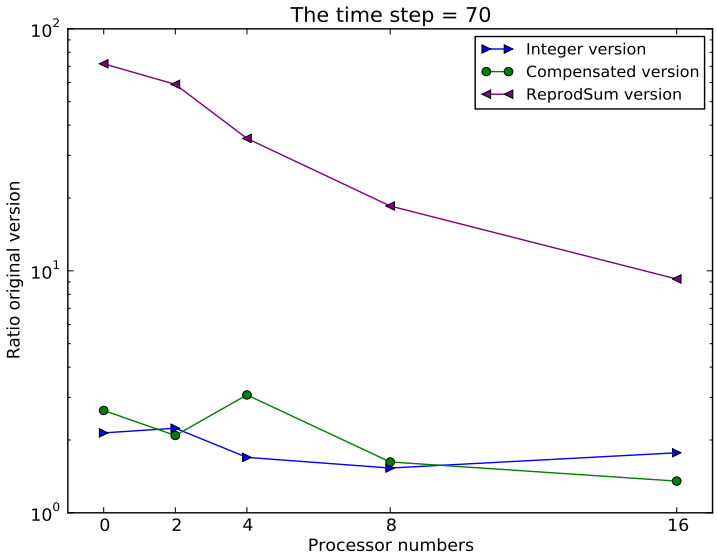
Listing 5.1 Synchronization of the processors in the performance test

```
1 MPI_BARRIER()
2 !Measure begins here
3 CALL RDTSC(Begin_Timer)
4 !Modified computations...
5 MPI_BARRIER()
6 !Measure ends here
7 CALL RDTSC(End_Timer)
8 Total_Time = End_Timer - Begin_Timer
```

5.3.1 Extra-cost of the reproducible Tomawac

We begin to compare the three reproducible versions of the Tomawac module presented in Section 5.1. This will allow us to confirm our choice for the compensation approach that was already analyzed to be easiest to implement in our study context.
As described in Section 5.1, the FE assembly step and the interface one are only modified in the reproducible Tomawac version. Hence extra-cost measures focus these assembly steps.
In Figure 5.10, the x-axis represents the number of processors ($p = 0$ for the sequential execution and $p = 2, 4, 8, 16$ for the parallel ones). The y-axis measures the extra-cost ratio compared to the original version. We remark one important extra-cost of the reproducible sum solution (purple plot) that ranges from 10 to 100. The extra-cost of the integer conversion and the compensated version are roughly similar and represent between $\times 2$ and $\times 3$ the original calculation time. We note that the speed up of the parallel implementation does not appear in these plots because the measure parts include all the communications between the sub-domains.

FIGURE 5.10: Reproducible Tomawac: Extra-cost of the three solutions (compensation, reproducible sum, integer conversion) compared to the original floating-point computation. (*Nice* test case)



We recall that integer conversion introduces a risky loss of accuracy. The solution with Demmel and Nguyen reproducible sum yields reproducibility but introduces here a higher extra-cost. Moreover in the accuracy term, in this measure the reproducible sum (K=1) produces result at the working precision, while the compensated one computes as accurately as in a twice working precision computation.
This justifies that we chose to develop one accurate and efficient compensated solution for the more complex Telemac-2D module.

5.3.2 Extra-cost of the reproducible Telemac-2D

We recover the numerical reproducibility of the *gouttedo* Telemac-2D test case using the compensation techniques presented in Section 5.2. We recall that both the building and solving phases are modified in this module. So we measure from the begin to the end of the time loop that includes all the modifications, see Listing 5.1. This avoids the large amount of read/write data in the rest of the simulation.
Since the extra-cost depends on the magnitude of the simulation, we compare 3 different meshes with 4624 ($\times 1$), 18225 ($\times 4$) and 72361 ($\times 8$) nodes. Table 5.6

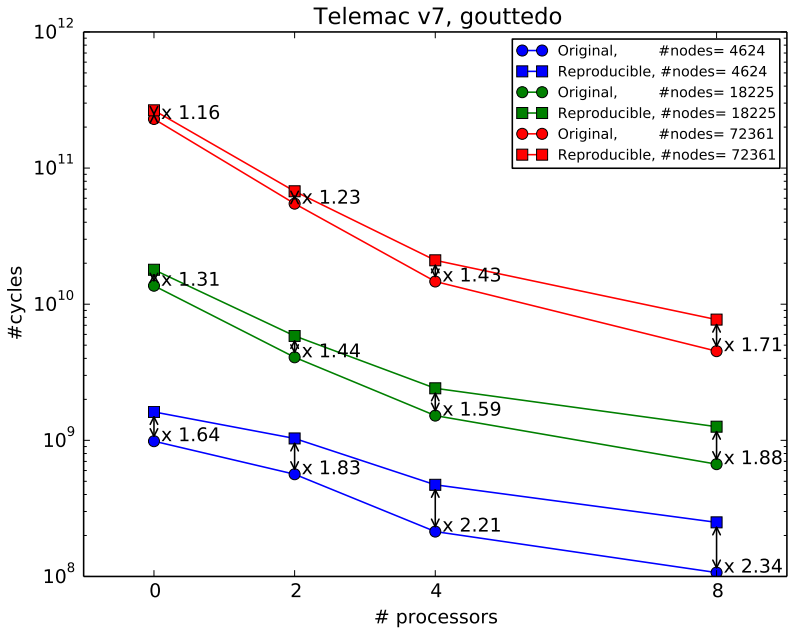
presents the significant increasing number of interface nodes as the mesh is refined. Of course, this also corresponds to a more important communication cost.

TABLE 5.6: The number of the interfaces point in 3 several meshes, when the number of computing units varies

#IP		#nodes		
		4624 nodes	18225 nodes	72361 nodes
#procs	2 procs	72	143	280
	4 procs	304	674	1368
	8 procs	501	1152	2020

Figure 5.11 presents the running time measured in cycles (y-axis) of the compensated reproducible version (■) and the original version (●) when the number of processors varies (x-axis). It also shows the ratio between these two versions and allows us to remark that the compensated algorithms double, more or less, the time of the core computations.

FIGURE 5.11: Extra-cost of the reproducible (compensated) computation compared to the floating-point one, (test case *gouttedo*, Telemac-2D).



The simulation time increases as the number of mesh nodes because of the extra-computations in the building and the resolution of a larger system. In addition, the number of iterations of the conjugate gradient significantly increases with respect to the dimension of the system. It is interesting to note that the extra-cost for reproducibility benefits from this simulation increasing time since our modifications only impact the performance of the core computation. At the contrary for a given mesh size, the ratio is larger when increasing the number of processors. This is due to the increase of the number of interface nodes and to their extra-cost treatment in the compensated version (Relation 5.2).

5.4 Conclusion

In this chapter we have been able to recover reproducible simulations with two modules included in openTelemac. We mainly use compensation techniques for the more general treatment provided by the Telemac-2D module. The first source of non-reproducibility is the non-deterministic error propagation at the interface nodes. We recall that this step is implicitly present in several parts of the computation (building and solving phases). It is sufficient to store and propagate these errors, and finally compensate them into the computed value after every step of the interface node assembly. These corrections are applied for both the parallel and the sequential simulations to yield the expected reproducibility. The second source is the dynamic reduction of the parallel implementation of the dot product in the conjugate gradient iterations. It is corrected by implementing a compensated dot product that computes in about twice the working precision (in sequentiel and parallel executions).

The considered test case *gouttedo* appears to be well conditioned enough with relative errors between the sequential results and the parallel ones varying between 10^{-15} and 10^{-13}. This allows us to estimate that a twice working precision can compute accurately rounded results, *i.e.* which leads to reproducible simulations using one level of the compensation techniques.

This approach is reasonable in term of running time extra-cost. We measured no significant extra-cost of the whole reproducible simulation compared to the original one when the read/write data process are considered. The

core computation takes about twice more time in the reproducible version. Of course, extra-cost could be of the same order for larger cases.

The feasibility and the efficiency of the compensation have been previously compared to other solutions like integer conversion and reproducible sums [15]. These three techniques were applied to the *Nice* test case of the Tomawac module where the non-reproducibility source is only the finite element assembly step. The compensated solution appeared to be the more efficient one. In this context we can summarize their current feasibility as follows. The compensated solution appears to be the easiest one to apply and provides accurate results for a low computing extra-cost. The integer conversion provided in Tomawac is also easy to derive and introduces a low extra-cost. Even if it is definitely not redhibitory here, this solution may introduce a risky loss of accuracy in other cases. A careful error analysis should be performed before other applications. The solution that uses recent reproducible sum algorithms is generally efficient, but in our case it applies less easily and introduces a significant communication extra-cost.

Chapter 6

How to implement reproducibility in openTelemac

In Chapter 5, we analyze and obtain the numerical reproducibility of *gouttedo* and *Nice*, two test cases of the modules Telemac-2D and Tomawac, by using the compensation techniques. To obtain a full reproducibility, *i.e.* in all openTelemac modules, these techniques have to be integrated in other computations that differ between the sequential execution and the parallel one. To facilitate such a task, this chapter aims to be a useful technical complement of Chapter 5.

We start describing in Section 6.1 the methodology to track the computation of the concerned problem. This process aims to identify the sources which produce the non-reproducibility. Then, we detail the modifications introduced in the code. As already mentioned, openTelemac relies on its finite element library BIEF. This one includes many Fortran 90 subroutines which provide the data structure, the building and the solving phases of the simulation. Almost all our modifications have been restricted to these library subroutines. We describe four types of modification: data structure (Section 6.2), algebraic operations (Section 6.3), building phase and solving phase (Sections 6.4 and 6.5). We exhibit and explain the modified parts highlighting them and commenting them in the listings proposed along the chapter.

The users choose between the original computation or a reproducible one, in the test case file (where all the parameters of the simulation are defined) via the keyword "FINITE ELEMENT ASSEMBLY" (or "ASSEMBLAGE EN ELEMENTS FINIS" in French). It corresponds to the Fortran variable `MODASS` that takes the values 1,2 and 3 respectively for the original, the integer and the

compensated mode. In this chapter we only consider the implementation of the compensated computation.

6.1 Methodology

We describe how to identify the source of non-reproducibility in a computation sequence. The strategy is to observe the components of the linear system (4.2) as the computation is progressing. For that, we introduce the subroutine `glob_vec` to observe the reproducibility of a concerned vector after each computation, see Listing 6.1. This process allows us to detect if a computation is reproducible or not.

In a parallel simulation, the vectors are distributed over the sub-domains where each node has a local and a global number, as described in Section 4.2. In order to compare the component values when the sub-domain number differs, we need to rebuild the global vector, *i.e.* for the whole domain. For that we use the structure `KNOLG` which maps the local number of a component to the global one. This treatment is realized in lines from 16 to 26.

As detailed in Chapter 5, reproducibility can be observed only after the interface point assembly (and the compensation), because the interface points are different for one decomposition to another.

For that, in our observation we identify if the interface point assembly has been already performed in lines 7 and 9 of Listing 6.1. If it is not the case, we call the assembly processing `parcom` or `parcom_comp`, corresponding to the computation mode. (we note that the test are realized on a copy of the component, line 3). In the compensated mode (`MODASS`=3), the sequential and the parallel cases both benefit from the compensation, lines 41 and 11 respectively

.

Listing 6.1 The BIEF_OBJ structure in `glob_vec`

```
1 !Input: X is the observed vector, if FLAG_ASS is true an interface point
  assembly is performed
2 !Copy the concerned vector to a temporary one
3 CALL OS('X=Y ',X=MESH%T,Y=X)
4 !In parallel case
5 IF (NCSIZE .NE. 0) THEN
6   IF (MODASS .EQ. 1) THEN
7     IF(FLAG_ASS) CALL PARCOM(MESH%T,2,MESH)
8   ELSEIF (MODASS .EQ. 3) THEN
```

```
9     IF(FLAG_ASS) THEN
10      CALL PARCOM_COMP(MESH%T,MESH%T%E,2,MESH)
11      MESH%T%R=MESH%T%R+MESH%T%E
12    ENDIF
13  ENDIF
14 !Procedure to obtain the global vector which is distributed over the sub-
   domains
15 !Each sub-domain stores its maximum local point
16  NPOIN_GLOBAL=MAXVAL(MESH%KNOLG%I)
17 !Sub-domains exchange their maximal local points to store the maximal, which
    represents the number of nodes in the whole domain
18  NPOIN_GLOBAL=P_IMAX(NPOIN_GLOBAL)
19  ALLOCATE(VALUE_GLOBAL(NPOIN_GLOBAL))
20  VALUE_GLOBAL(:)=HUGE(1.D0)
21  DO I=1,NPOIN
22     VALUE_GLOBAL(MESH%KNOLG%I(I))=MESH%T%R(I)
23  END DO
24  DO I=1,NPOIN_GLOBAL
25     VALUE_GLOBAL(I)=P_DMIN(VALUE_GLOBAL(I))
26  END DO
27 !Write the result by the master sub-domains in a file
28  IF (IPID .EQ. 0) THEN
29    OPEN(UNIT=99,FILE='./'//X%NAME//'VEC_GLOBAL.TXT')
30    WRITE(99,*) X%NAME,", NB POINTS:",NPOIN_GLOBAL
31    DO I=1,NPOIN_GLOBAL
32      WRITE(99,*) VALUE_GLOBAL(I)
33      CALL FLUSH(99)
34    END DO
35    CLOSE(99)
36  END IF
37  CALL P_SYNC
38 !In sequential case, write the results in a file
39 ELSE
40  IF ((MODASS .EQ.3) .AND. FLAG_ASS)THEN
41    MESH%T%R=MESH%T%R+MESH%T%E
42  ENDIF
43  OPEN(UNIT=99,FILE='./'//X%NAME//'VEC_GLOBAL.TXT')
44  WRITE(99,*) X%NAME,", NB POINTS:",NPOIN
45  DO I=1,NPOIN
46     WRITE(99,*) MESH%T%R(I)
47     CALL FLUSH(99)
48  END DO
49  CLOSE(99)
50 END IF
```

6.2 Modifications in the data structure

The main data type in the BIEF library is `BIEF_OBJ` which may be a vector, a matrix or a block. The part of the subroutine `bief_def` in Listing 6.2 illustrates some of the vector and matrix structure types.
We write `V%R` the R component of the vector V which corresponds to the data. In the compensated version, these R values will be associated, when necessary, with the accumulation of the corresponding generated rounding errors. These errors will be stored in a component named E, and we write `V%E` to access to it. The same notations exist for a diagonal D of the matrix M: we write `M%D%R` for the data and `M%D%E` for the errors. The component E accumulates the generated rounding errors in each computation with R, this latter will be corrected by a compensation $R + E$.

Listing 6.2 The BIEF_OBJ structure in `bief_def`

```
1  ! Structures in the object BIEF_OBJ:
2  TYPE BIEF_OBJ
3    INTEGER TYPE     ! 2: vector,  3: matrix,  4: block
4    CHARACTER(LEN=6) NAME ! Name of the object
5  ! For vectors
6    INTEGER NAT        ! 1:DOUBLE PRECISION 2:INTEGER
7    INTEGER ELM        ! Type of element
8    INTEGER DIM1       ! First dimension
9    INTEGER DIM2       ! Second dimension
10 ! Double precision vector
11 ! Data are stored here
12   DOUBLE PRECISION,POINTER,DIMENSION(:)::R
13  ! Errors are stored here
14     DOUBLE PRECISION, POINTER,DIMENSION(:)::E
15 ! For matrices
16   INTEGER STO        ! 1: EBE storage  3: EDGE-BASED storage
17   TYPE(BIEF_OBJ),POINTER :: D ! Pointer to a BIEF_OBJ for the diagonal
18   TYPE(BIEF_OBJ),POINTER :: X ! Pointer to a BIEF_OBJ for extra-
     diagonal terms
19 END TYPE BIEF_OBJ
```

Note 1. The new component `V%E` is allocated in the subroutine `bief_allvec`.

Note 2. When the routine parameters only include `BIEF_OBJ` type, our modifications are automatically available in the body of the subroutine: all the structure components are accessible as `V%R` or `V%E`. Nevertheless, some subroutines work directly with double precision vectors that used

to pass an object's component, as `V%R`. In this case, we have to modify the subroutine parameter by manually adding a supplementary one for `V%E`.

6.3 Modifications in the algebraic operations

In Chapter 5, we explain how the rounding errors `V%E` must be updated for each algebraic operation on `V%R`. Every operation on a block or a vector is called by the subroutine `os`, which only verifies the structure before calling the subroutine `ov`. This latter computes the required operation *op* on the passed vectors `X%R`, `Y%R`, `Z%R`, for instance it computes `X%R = Y%R + Z%R`.
In the compensated mode, the new subroutine `ov_comp` is called, and the passed vectors are associated with their own error vectors `X%E`, `Y%E`, `Z%E`, to also update them. Listing 6.3 illustrates the modified vector add and the Hadamard product, for instance.

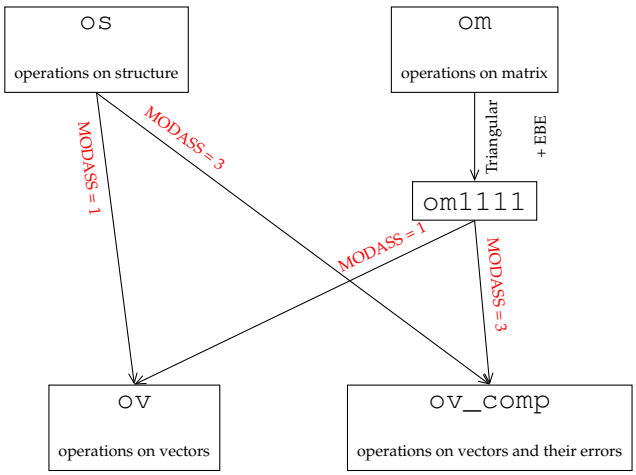
Listing 6.3 The algebraic operations in `ov_comp`

```
1  !X,Y and Z represent the values
2  !!X_ERR,Y_ERR and Z_ERR represent the errors
3  !For  initialization
4  CASE('0  ')
5  DO I=1,NPOIN
6    X(I) = 0.D0
7     X_ERR(I)=0.D0
8  ENDDO
9  !Copy Y to X
10 CASE('Y  ')
11 DO I=1,NPOIN
12   X(I) = Y(I)
13   X_ERR(I) = Y_ERR(I)
14 ENDDO
15 !Add two vectors
16 !In the original code is X(I) = Y(I) + Z(I)
17 DO I=1,NPOIN
18   CALL TWOSUM(Y(I),Z(I),X(I),ERROR)
19   X_ERR(I)=(Y_ERR(I)+Z_ERR(I))+ERROR
20 ENDDO
21 !Value by value product
22 !In the original code is X(I) = Y(I) * Z(I)
23 DO I=1,NPOIN
24  CALL TWOPROD(Y(I),Z(I),X(I),ERROR)
25  X_ERR(I)=(Y(I) * Z_ERR(I))+(Y_ERR(I) * Z(I))
```

```
26 &          +(Y_ERR(I)  *  Z_ERR(I))
27   X_ERR(I)=X_ERR(I)+  ERROR
28 ENDDO
```

All these operations are also applied to the diagonal and the extra-diagonal terms of the EBE matrix structure, respectively stored as vectors in `M%D` and `M%X`. The difference is in the sequence of the calls, which begin by the subroutine `om` for matrix instead of `os` for other structures. In `om`, the storage and the type of the element are verified to call the subroutine `om1111` for triangular elements and EBE storage. In this subroutine, several tests are verified to then pass the corresponding component vector of the matrix to the subroutines `ov` or `ov_comp`. In the compensated version, the only modification in `om` and `om1111` is at the subroutine parameter level to pass the error vectors.

FIGURE 6.1: A general scheme of the algebraic operation calls



6.4 Modifications in the building phase

As detailed in Chapter 5, the steps of the building phase which condition the reproducibility are the finite element assembly and its complement in parallel, the interface node assembly. In the compensated mode, the generated rounding errors of an elementary addition are calculated by the subroutine `2sum` (Algorithm 3.2) which is added in BIEF. In practice, the computation of any vector is realized in the subroutine `vectos`. Listing 6.4 is a part of this subroutine and we detail it in three steps.

Listing 6.4 The call of the FE assembly under the two modes of the computation in `vectos`

```
1  ! Note: VEC is a reference to SVEC%R
2  IF(MODASS.EQ.1) THEN
3    CALL ASSVEC(VEC, IKLE, NPT ,NELEM,NELMAX,IELM1,
4  &   T,INIT,LV,MSK,MASKEL,NDP)
5  ELSEIF(MODASS.EQ.3 ) THEN
6   CALL ASSVEC(VEC, IKLE, NPT ,NELEM,NELMAX,IELM1,
7  &  T,INIT,LV,MSK,MASKEL,NDP,SVEC%E)
8  ENDIF
9  ! Implicit modification in PARCOM
10 IF(ASSPAR) CALL PARCOM(SVEC,2,MESH)
11 IF(ASSPAR.AND.MODASS.EQ.3) THEN
12 ! The compensation of all the values
13  DO I = 1 , MESH%NPOIN
14     VEC(I)= VEC(I)+SVEC%E(I)
15  ENDDO
16 ENDIF
```

1. `vectos` calls the subroutine `assvec` that computes the finite element assembly process corresponding to the computation mode, from line 2 to 8. In the original mode, only the vector `VEC` is passed into the `assvec` call, while in the compensated mode, we also pass the vector of errors `SVEC%E`.

Listing 6.5 The FE assembly in `assvec`

```
1  !X refers to VEC and ERRX refers to SVEC%E
2  DO IDP = 1 , NDP
3   DO IELEM = 1 , NELEM
4     IF (MODASS.EQ.1)
5   &   X(IKLE(IELEM,IDP)=X(IKLE(IELEM,IDP)+W(IELEM,IDP)
6     ELSEIF (MODASS.EQ.3) THEN
7       CALL 2SUM(X(IKLE(IELEM,IDP)),
8     &   W(IELEM,IDP),X(IKLE(IELEM,IDP)),ERROR)
9       ERRX(IKLE(IELEM,IDP))=ERRX(IKLE(IELEM,IDP))+ERROR
10    ENDIF
11  ENDDO
12 ENDDO
```

As showed in Listing 6.5, only the vector `VEC` is assembled in the original computation (line 5). In the compensated mode, `VEC` is assembled by the subroutine `2sum` (lines 7 and 8) that also computes the rounding error `ERROR` for each node `IKLE(IELEM,IDP)`. The vector `SVEC%E` accumulates the generated errors `ERROR` of each node.

2. The second implicit modification in Listing 6.4 (line 10) is the interface node assembly that is launched by the subroutine `parcom`. This later calls `parcom2`, which then calls `paraco` twice in the original mode. We recall that the first call is to assemble the sub-domain contributions and the second is to recover the solution continuity between the sub-domains by sharing their maximum value (this choice is justified by physical reasons in [25]).
   In the compensated mode, `parcom2_comp` and `paraco_comp` replace `parcom2` and `paraco`, respectively.
   The modification in `parcom_comp` are the addition of the error component parameter and the suppression of the second call of `paraco` which is no more need because our corrections recover the solution continuity between the sub-domains.
   The main modifications occur in `paraco_comp` and are presented in Listing 6.6. The Fortran subroutines `p_iread`, `p_iwrit` and `p_wait_paraco` call respectively the MPI operations: `mpi_irecv`, `mpi_isend` and `mpi_waitall`. The communication corresponds to a non-blocking receive with a blocking send [45]. The `BIEF_MESH` structures, `BUF_RECV` and `BUF_SEND`, are declared in `bief_def` to receive and send the exchanged data between the sub-domains. The assembly of the sub-domain contributions in is a simple accumulation of these received data, see Listing 6.6 (line 38).
   In the compensated computation, two new structures `BUF_RECV_ERR` and `BUF_SEND_ERR` are added to also exchange the computed errors. Here the assembly is realized with the `2sum` subroutine that computes the rounding error `ERROR1` of the data accumulation (line 40) and `ERROR2` for the error accumulation (line 42). These two values are added with the error contributions in each iteration (line 44).

Listing 6.6 The IP assembly in `paraco_comp`

```
1 ! Receive step
2 DO IL=1,NB_NEIGHB
3   IKA = NB_NEIGHB_PT(IL)
4   IPA = LIST_SEND(IL)
5   CALL P_IREAD(BUF_RECV(1,IL),IAN*IKA*NPLAN*8,
6 &            IPA,PARACO_MSG_TAG,RECV_REQ(IL))
7   CALL P_IREAD(BUF_RECV_ERR(1,IL),IAN*IKA*NPLAN*8,
```

```
8  &            IPA,PARACO_MSG_TAG,RECV_REQ(IL))
9  ENDDO
10 ! Send step
11 DO IL=1,NB_NEIGHB
12   IKA = NB_NEIGHB_PT(IL)
13   IPA = LIST_SEND(IL)
14   ! Initializes the communication arrays
15   K = 1
16   DO J=1,NPLAN
17     DO I=1,IKA
18       II=NH_COM(I,IL)
19       BUF_SEND(K,IL) =V1(II,J)
20       BUF_SEND_ERR(K,IL) =ERRX(II)
21       K=K+1
22     ENDDO
23   ENDDO
24   CALL P_IWRIT(BUF_SEND(1,IL),IAN*IKA*NPLAN*8,
25 &             IPA,PARACO_MSG_TAG,SEND_REQ(IL))
26     CALL P_IWRIT(BUF_SEND_ERR(1,IL),IAN*IKA*NPLAN*8,
27 &               IPA,PARACO_MSG_TAG,SEND_REQ(IL))
28 ENDDO
29 ! Wait received messages
30 DO IL=1,NB_NEIGHB
31   IKA = NB_NEIGHB_PT(IL)
32   IPA = LIST_SEND(IL)
33   CALL P_WAIT_PARACO(RECV_REQ(IL),1)
34   K=1
35   DO J=1,NPLAN
36     DO I=1,IKA
37       II=NH_COM(I,IL)
38 ! Original version: V1(II,J)=V1(II,J)+ BUF_RECV(K,IL)
39       CALL 2SUM(V1(II,J),BUF_RECV(K,IL)
40   &       ,V1(II,J),ERROR1)
41       CALL 2SUM(ERRV(II),BUF_RECV_ERR(K,IL)
42   &       ,ERRV(II),ERROR2)
43       ERROR=ERROR1+ERROR2
44       ERRV(II)=ERRV(II)+ERROR
45       K=K+1
46     ENDDO
47   ENDDO
48 ENDDO
```

3. The latest modification in Listing 6.4, from line 11 to 16, is the compensation after the interface point assembly. As detailed in Section 5.2.1, we have to compensate the accumulated errors to the data. After this step this vector becomes reproducible.

Note 3. This procedure is applied to every vector and EBE matrix. The diagonal `M%D%R` is a vector and its accompanying vector error term `M%D%E` is calculated in a similar way.

Note 4. For the studied *gouttedo* test case, the other calls of `parcom` are in the subroutines `propag`, `masbas2d`, `matrbl`. In the compensation mode, these subroutines apply the previously modifications (items 2 and 3).

6.5 Modifications in the solving phase

The resolution phase applies the conjugate gradient method (Algorithm 4.2) provided by the subroutine `gracjg`. The modifications impact the computations of the dot product in function `p_dots`, and the EBE matrix-vector product in subroutine `matrbl`, which are called by `gracjg`.

i) The dot product $X \cdot Y$

Subroutine `p_dots` calls the corresponding dot product according to the computation mode, as showed in Listing 6.7.

Listing 6.7 The calls of the corresponding dot product in `p_dots`

```
!Declaration of a pair of double precision
 DOUBLE PRECISION PAIR(2)
!The computation of the corresponding dot product
!In the original version
!DOT for the sequential and P_DOT for parallel executions
IF (MODASS .EQ. 1) THEN
  IF(NCSIZE.LE.1.OR.NPTIR.EQ.0) THEN
    P_DOTS=DOT(NPX,X%R,Y%R)
  ELSE
    P_DOTS=P_DOT(NPX,X%R,Y%R,MESH%IFAC%I)
  ENDIF
!In the compensated version
!DOT_COMP for the sequential and P_DOTPAIR for parallel executions
ELSEIF (MODASS .EQ. 3) THEN
  IF(NCSIZE.LE.1.OR.NPTIR.EQ.0) THEN
    P_DOTS=DOT_COMP(NPX,X%R,Y%R)
  ELSE
    CALL P_DOTPAIR(NPX,X%R,Y%R,MESH%IFAC%I,PAIR)
  ENDIF
ENDIF
! Final sum on all the sub-domains (MPI subroutines)
IF (MODASS .EQ. 1) THEN
```

```
23   IF(NCSIZE.GT.1) P_DOTS = P_DSUM(P_DOTS)
24 ELSEIF (MODASS .EQ. 3) THEN
25   IF(NCSIZE.GT.1) P_DOTS = P_DSUMERR(PAIR)
26 ENDIF
```

In the sequential original mode (NCSIZE< 1 and MODASS= 1), the dot product is computed with the function dot as:

```
DO I = 1 , NPOIN
  DOT= DOT + X%R(I)*Y%R(I)
END DO
```

In the parallel original mode, the dot product of the whole domain is computed partially by each sub-domain, in function p_dot, as:

```
DO I = 1 , NPOIN
  P_DOT = P_DOT+X%R(I)*Y%R(I)*IFAC(I)
END DO
```

where IFAC is the weight used to avoid to compute several times the interface nodes (Section 4.6.2 and 5.2.2). These partial contributions are summed over all the sub-domains to compute the global dot product by the MPI dynamic reduction in p_dsum.

In the compensated mode, a twice more accurate scalar product is computed. In sequential, function dot_comp (Algorithm 3.8) computes a such accurate sequential dot product. It accumulates both the dot product and the generated rounding errors (addition and multiplication) and finally compensates them together.

Note: in Fortran the name of the function is the output of this function.

Listing 6.8 The sequential Dot2 in the subroutine DOT_COMP

```
1 CALL 2PROD(X(1),Y(1),P,EP)
2 DO I = 2 , NPOIN
3   CALL 2PROD(X(I),Y(I),PP,EPP)
4   CALL 2SUM(P,PP,P,E)
5   EP=EP+(E+EPP)
6 END DO
7 DOT_COMP = P+EP
```

In the parallel implementation, each sub-domain computes its local scalar product and the corresponding generated rounding errors, to return a pair [data, error] in subroutine p_dotpair.

Listing 6.9 The parallel Dot2 in the subroutine p_dotpair

```
!Input: X(NPOIN),Y(NPOIN). Output: PAIR(2)
CALL 2PROD(X(1),Y(1)*IFAC(1),P,EP)
DO I = 2 , NPOIN
  CALL 2PROD(X(I),Y(I)*IFAC(I),PP,EPP)
  CALL 2SUM(P,PP,P,E)
  EP=EP+(E+EPP)
END DO
CALL 2SUM(P,EP,PAIR(1),PAIR(2))
```

These local pairs are exchanged between processors via MPI_ALLGATHER and are accurately accumulated by `sum2` in every processor, see Listing 6.10 in lines 8 and 11.

Listing 6.10 The final sum on all the sub-domains

```
!In original version
!CALL MPI_ALLREDUCE(MYPART,P_DSUM,1,MPI_DOUBLE_PRECISION,
!                   MPI_SUM,MPI_COMM_WORLD,IER)
!In compensated version
CALL MPI_COMM_SIZE (MPI_COMM_WORLD, NUM_PROCS, IER)
ALLOCATE(ALL_PARTIAL_SUM(1:2*NUM_PROCS))
ALL_PARTIAL_SUM=0.D0
CALL MPI_ALLGATHER (MYPART, 2, MPI_DOUBLE_PRECISION,
&              ALL_PARTIAL_SUM,2, MPI_DOUBLE_PRECISION,
&              MPI_COMM_WORLD, IER)
CALL SUM2(2*NUM_PROCS, ALL_PARTIAL_SUM,P_DSUMERR)
DEALLOCATE(ALL_PARTIAL_SUM)
```

ii) The matrix-vector product $M \times V$

The EBE storage and the EBE matrix-vector product have been detailed in Sections 4.2.3 and 4.2.4. Matrix M is stored as `M%D` of size `NPOIN` for its diagonal terms and `M%X` for its extra-diagonal ones of size `NPOIN*(NPOIN-1)` in each element `IELEM`.

The $M \times V$ product is launched by subroutine `matrbl` called in the conjugate gradient. Three subroutines are then called: `matvec`, `matvct` and `mv0303`.

In the compensated version, the subroutine parameters of the two later ones are modified to pass the associated errors `M%D%E`, `V%E`. In `mv0303`, the Hadmard product $DA \times Y$ computed by `ov_comp` is modified to update the associated errors.

Listing 6.11 EBE matrix-vector product: the multiplication of the extra-diagonal elementary terms and the diagonal terms of the matrix with the corresponding elements of the vector in `mv0303`.

```
1  ! Here Y refers to V%R, DA refers to M%D%R and XA refers to M%X
2  !Contribution of extra-diagonal terms XA * Y
3  DO IELEM = 1 , NELEM
4    W1(IELEM) = XA12(IELEM) * Y(IKLE2(IELEM))
5  &           + XA13(IELEM) * Y(IKLE3(IELEM))
6    W2(IELEM) = XA23(IELEM) * Y(IKLE3(IELEM))
7  &           + XA21(IELEM) * Y(IKLE1(IELEM))
8    W3(IELEM) = XA31(IELEM) * Y(IKLE1(IELEM))
9  &           + XA32(IELEM) * Y(IKLE2(IELEM))
10 END DO
11 !Contribution of the diagonal DA * Y
12 IF ( MODASS .EQ.1) THEN
13   CALL OV ('X=YZ ', X , Y , DA , C , NPOIN )
14 ELSEIF (MODASS .EQ. 3) THEN
15   CALL OV_COMP ('X=YZ ', X , Y , DA , C , NPOIN
16  &           ,X_ERR, Y_ERR , DA_ERR )
17 ENDIF
```

Listing 6.11 (from lines 3 to 10) illustrates the process of the elementary contribution computations. These latter proceed to a finite element assembly. In the compensation mode, this assembly is performed as we detailed in subroutine `assvec` (Listing 6.5). The final step assembles the matrix-vector product at the interface point in `matrbl` with a `parcom` call and finishes with a compensation operations.

6.6 Conclusion

In this chapter we detail, with a technical point of view, the modifications we introduced in openTelemac to recover the reproducibility of the studied test cases. The first difficulty in this work was to define and to apply the methodology detailed in Section 6.1 to such a huge code. The second difficulty was to identify the sources of non-reproducibility, *i.e.* where the rounding errors differ between the sequential and the parallel simulations, and to distinguish their implementations in (again) this huge code. It was inevitable to manipulate three openTelemac components: the BIEF library, the parallel library and Telemac-2D module which include respectively 493, 46 and 192 subroutines. The modifications to obtain reproducibility were restricted to about 30 subroutines, mostly in BIEF. We list these modified subroutines at the end of this section.

The first source is the non-deterministic error propagation at the interfaces nodes. We recall again that this step is implicitly present in several parts of the computation (building and solving phases). It is sufficient to store and propagate these errors and finally to compensate them into the computed value after every step of interface node assembly. These corrections are applied for both the parallel and the sequential simulations to yield the expected reproducibility between the two execution modes. The second source is the dynamic reduction of the parallel implementation for the dot product in the conjugate gradient iterations. It is corrected by implementing a dot product that computes in about twice the working precision. Here it yields reproducible results whereas this is not true for very ill-conditioned ones. In this latter case, more compensated steps can be applied for instance.

We think that these details are important to the continuity of this work. Of course, that this chapter necessitates a little knowing of openTelemac code. The integration of our modifications is still in progress and it is expected that this will be available in the next distributed version of openTelemac. One integration difficulty is that the code was changed in the meantime of this work, which requires a careful merge between all these modifications.

List of modified subroutines

BIEF library.

— ***Modified:*** `almesh`, `assvec`, `bief`, `bief_allvec`, `bief_def`, `matrbl`, `matrix`, `matvec`, `om`, `mv_0303`, `om_1111`, `os`, `p_dots`, `parini`, `precd1`, `solve`, `vectos`.

— ***Added:*** `ov_comp`, `dot_comp`, `p_dot_comp`, `parcom_comp`, `parcom2_comp`, `paraco_comp`,`twosum`, `twoprod`.

Parallel library.

— ***Modified:*** `interface_parallel`.

— ***Added:*** `p_dsum_err`.

Telemac-2D module.

— ***Modified:*** `lecdon_telemac`, `masbas2d`, `propag`, `telemac2d.dico`.

Chapter 7

Conclusion and perspectives

This manuscript is devoted to the numerical reproducibility issue in computer science. We study this important reliability limitation of simulations focusing a large scale open software largely used for industrial and scientific hydrodynamics simulations. In Chapter 2 we explain how floating-point arithmetic and parallel computations collapse the numerical reproducibility of *a priori* deterministic arithmetic sequences. We exhibit several simulation cases with openTelemac that suffer from numerical reproducibility failures. We also briefly consider the more general motivation for reproducibility in research and present some main useful tools to reach this target in computer science research. Of course, numerical reproducibility is one of the steps to achieve toward this important issue.

In Chapter 3 we present and analyze existing methods that may fix this failure. In order to recover numerical reproducibility, we describe methods that improve the accuracy and others that eliminate the non-associativity of addition, which is the main culprit of this issue.

In this work we target to recover the numerical reproducibility in openTelemac. The first questions to consider were the following. What are the non-reproducibility sources? How to correct it applying existing techniques in this complex framework? How much does this improvement cost?
In Chapter 4 we describe the main steps of a finite element simulation as implemented in openTelemac. We detail that the Tomawac module has only one source of non-reproducibility which is the finite element assembly. This step is applied only on three vectors which are the second members of a diagonal system. In Chapter 5 we present how to recover its numerical reproducibility and also compare the applicability of three existing solutions: compensation, Demmel and Nguyen reproducible sums [15] and integer conversions [51].

The compensated solution appears to be the easiest one to apply to the assembly step and also provides accurate results for a low computing extra-cost.

These first results justify that we choose to apply compensation in the more complex module Telemac-2D. In Chapter 4 we explain the complexity of this simulation by detailing the building of the linear system for the three vector unknowns of the test case *gouttedo*, and how its components depend from each other. The difficulties were to identify the sources of the non-reproducibility, *i.e.* where the rounding errors differ between the sequential and the parallel simulations. In practice, a time consuming task was to distinguish their implementations in such a huge code. The first source is the non-deterministic error propagation at the interfaces nodes. We recall that this step is implicitly merged in several parts of the computation (building and solving phases). In Chapter 5 we detailed that it is sufficient to store and propagate these errors and to finally compensate them into the computed value after every interface node assembly process. This correction process is applied to each vector and to the EBE matrix-vector product in the solving step. This is implemented for both the parallel and the sequential simulations to yield the expected reproducibility. The second source is the dynamic reduction of the parallel implementation for the dot product in the conjugate gradient iterations. It is corrected by implementing a compensated dot product that produces accurate results as it corresponds to a computation in twice the working precision.
This approach appears to be reasonable in term of running time extra-cost. We measure no significant extra-cost of the whole reproducible simulation compared to the original one when the read/write data process are taken into account. Of course, as the computation core takes about twice more time in the reproducible version, the extra-cost could be of the same order for larger cases. In practice, these modifications impact three openTelemac components: the Bief library, the parallel library and Telemac-2D module which include respectively 493, 46 and 192 subroutines. We successfully managed to restrict the modifications to about 30 subroutines, mostly in BIEF.

Perspectives

This work is a first step toward a complete reproducible openTelemac. We identified and detailed the failure sources. According to their experience,

openTelemac developers are optimistic that no other source remains in the code [26]. However in this work, we only tracked and modified the computation sequence of the *gouttedo* test case in Telemac-2D to recover its reproducibility (and the more simple Tomawac resolution). Modifications were necessary to the operations envolved with vectors and EBE matrices. We define now some future work steps that can lead a completely reproducible openTelemac. We also define some possible optimizations and analysis that are not achieved in this work and that may be useful. We distinguish three directions of continuing work.

— ***Analysis issue.*** In this work we compare the efficiency of some methods that lead to the numerical reproducibility. However, more comparisons can be applied to other methods in order to identify the "best" ones. Since non-reproducibility sources are now localized, it becomes easier to perform these tests. For instance we present expansion technique in Chapter 3. The integration of the double-double library [27] may be more or less easy to apply. The advantage is that all the computations will be computed more accurately and no more hand-made modifications will be needed. The drawback of this method is that most of existing published experiments proves that expansions are more expensive in term of running time than compensation. In practice this solution should allow us to overload operations in the whole code, contrary to the compensation method that is introduced only where it is needful.

In Section 5.1 we detail the drawback of Demmel and Nguyen ReprodSumK algorithm [15]. It introduces a significant extra-cost because it needs the global maximum value over all the sub-domains to compute the pre-rounding boundary. In contrary, the newest Demmel and Nguyen 1-Reduction algorithm [16] is interesting to be tested because the pre-rounding boundary is now computed locally in each sub-domain. However the implementation of this algorithm is difficult: the coding of the manipulated values must be changed. So, in our context, it is not easy to estimate the feasibility cost of this solution neither its extra-cost without implementing and testing it.

In Chapter 4 we explain that the relative errors between the sequential and the parallel executions depend on the conditioning of the given

problem.

In the test cases studied in this thesis, relative errors vary between 10^{-15} and 10^{-13}. This leads us to estimate that results computed as in a twice working precision should yield the reproducibility. One important step is to consider more ill-conditioned test cases, in order to analyze the required level of compensation and where to apply it.

— ***Solving issue.*** Up to our knowledge, modifications detailed in Chapter 5 seem easy to be integrated in future versions of a reproducible openTelemac. The reproducibility of the whole openTelemac computation should integrate the same kind of modifications in other computation that are not yet reproducible as for other solving options, *e.g.* additional physical terms or other linear system solvers. Other matrix structures and operations should also be corrected, *e.g.* operations on block structure, edge-based storage of matrices, *etc.* In this thesis, we focus on the EBE storage where only the diagonal term is assembled. We detailed in Chapter 5 the modifications for this storage. The edge-based storage is different since extra-diagonal terms are assembled in segment. Hence, this assembly process should also be corrected. Indeed, a careful analysis of each process is necessary to locate the needed modifications.

— ***Engineering aspect.*** Implementing the kind of modifications we have described is easier when the code is suitably designed and written. For instance, in Chapter 6 (Note 2) we report the issue of the numerous mandatory hand-made modifications to pass the error component as input/output in subroutines. An engineering optimization is to always pass the whole vector structure to avoid these modifications of the input/output levels of the subroutines.

As detailed in Chapter 5, we introduced one pair [value,error] in our modifications. The error has to be propagated over the operations on the value. The operation modifications that consider the pair are applied manually. Fortran (starting from Fortran 90) supports operator overloading through defined operator which allows you to extend an intrinsic operator (*i.e.* the mathematical operations $(+, -, \times, /)$ and the assignment $=$) or to define a new operator. This tools can facilitate the modification of the whole operations and should be used for the future steps toward

a more reliable, since reproducible, openTelemac software.

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

HPC	**H**igh **P**erformance **C**omputing
MPI	**M**essage **P**assing **I**nterface
FP	**F**loating **P**oint
RN	**R**ound to **N**earest
RD	**R**ound **D**own
RD	**R**ound **U**p
RZ	**R**ound toward **Z**ero
ulp	**u**nit in the **l**ast **p**lace
V&V	**V**alidation and **V**erification
EFT	**E**rror-**F**ree **T**ransformation
MPFR	**M**ultiple **P**recision **F**loating-point **R**eliably library
flop	**fl**oating-point **op**eration
FMA	**F**used **M**ultiply-**A**dd
lsp	**l**east **s**ignificant **b**it
msb	**m**ost **s**ignificant **b**it
FE	**F**inite **E**lement
R&D	**R**esearch and **D**evelopment
EDF	**E**lectricité **D**e **F**rance
BIEF	**BI**bliothèque d'**E**lements **F**inis
PDE	**P**artial **D**ifferential **E**quations
ODE	**O**rdinary **D**ifferential **E**quations
FEM	**F**inite **E**lement **M**ethod
CG	**C**onjugate **G**radient
SPD	**S**ymmetric **P**ositive **D**efinite
EBE storage	Element-**B**y-Element storage
RDTSC	**R**ea**D** **T**ime **S**tamp **C**ounter

Bibliography

[1] URL = http://crd.lbl.gov/~dhbailey/mpdist.

[2] *Ariane 5 Flight 501 Failure: Report by the Inquiry Board*. Tech. rep. July 1996.

[3] F. C. F. Arturo Casadevall. *Reproducible Science*. 2010. URL: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2981311/.

[4] M. Baker. *Biotech giant publishes failures to confirm high-profile science*. 2016.

[5] B. W. Boehm. "Guidelines for Verifying and Validating Software Requirements and Design Specifications". In: *Euro IFIP 79*. Ed. by P. A. Samet. North Holland, 1979, pp. 711–719.

[6] C. Briere et al. "Assessment of TELEMAC system performances, a hydrodynamic case study of Anglet, France". In: *Coastal Engineering* 54.4 (2007), pp. 345–356. URL: http://doc.utwente.nl/73852/.

[7] W.-F. Chiang et al. "Determinism and Reproducibility in Large-Scale HPC Systems". In: *5th Workshop on Determinism and Correctness in Parallel Programming*. WoDet2013. Washington, USA. ACM, 2013.

[8] M. A. Christie et al. "Error Analysis and Simulations of Complex Phenomena". In: 2008.

[9] D. Christophe. "Numerical verification of an Industrial code to simulate accurately large scale Hydrodynamics events". In: *La Houille Blanche* 2 (2013), pp. 46–51. DOI: 10.1051/lhb/2013015. URL: http://dx.doi.org/10.1051/lhb/2013015.

[10] M. A. Cleveland et al. "Obtaining identical results with double precision global accuracy on different numbers of processors in parallel particle Monte Carlo simulations". In: *J. Comput. Phys.* 251.0 (2013), pp. 223 –236. ISSN: 0021-9991. DOI: http://dx.doi.org/10.1016/j.jcp.2013.05.041. URL: http://www.sciencedirect.com/science/article/pii/S0021999113004075.

[11] C. Collberg et al. “Measuring Reproducibility in Computer Systems Research”. In: (2014), pp. 1–37.

[12] L. Dagum and R. Menon. “OpenMP: An Industry-Standard API for Shared-Memory Programming”. In: *IEEE Comput. Sci. Eng.* 5.1 (Jan. 1998), pp. 46–55. ISSN: 1070-9924. DOI: 10.1109/99.660313. URL: http://dx.doi.org/10.1109/99.660313.

[13] M. Daumas. “Multiplications of Floating Point Expansions”. In: *14th IEEE Symposium on Computer Arithmetic (Arith-14 '99), 14-16 April 1999, Adelaide, Australia*. 1999, pp. 250–257. DOI: 10.1109/ARITH.1999.762851. URL: http://dx.doi.org/10.1109/ARITH.1999.762851.

[14] T. J. Dekker. “A Floating-Point Technique for Extending the Available Precision”. In: *Numer. Math.* 18 (1971), pp. 224–242.

[15] J. W. Demmel and H. D. Nguyen. “Fast Reproducible Floating-Point Summation”. In: *Proc. 21th IEEE Symposium on Computer Arithmetic*. Austin, Texas, USA. 2013.

[16] J. W. Demmel and H. D. Nguyen. “Parallel Reproducible Summation”. In: *IEEE Transactions on Computers* 64.7 (2015), pp. 2060–2070. ISSN: 0018-9340.

[17] K. Diethelm. “The Limits of Reproducibility in Numerical Simulation.” In: *Computing in Science and Engineering* 14.1 (2012), pp. 64–72. URL: http://dblp.uni-trier.de/db/journals/cse/cse14.html#Diethelm12.

[18] C. Drummond. “Replicability is not reproducibility: Nor is it good science”. In: *Proceedings of the Evaluation Methods for Machine Learning Workshop at the 26th ICML*. 2009.

[19] C. Ferrarin et al. “Tide-surge-wave modelling and forecasting in the Mediterranean Sea with focus on the Italian coast”. In: *Ocean Modelling* 61 (2013), pp. 38 –48. ISSN: 1463-5003. DOI: http://dx.doi.org/10.1016/j.ocemod.2012.10.003. URL: http://www.sciencedirect.com/science/article/pii/S1463500312001485.

[20] *Computational Reproducibility: State-of-the-art, Challenges, And Database Research Opportunities*. Vol. 0. ACM. 2012, pp. 593–596. DOI: 10.1145/2213836.2213908. URL: http://scholar.google.com/scholar?hl=en&q=http://dl.acm.org/citation.cfm%3Fid%3D2213908&sa=X&scisig=AAGBfm0VTk48vQJj7XgONzjWyHMUbbzaxQ&btnG=Search&as_sdt=0%2C15.

[21] N. Gentile, M. Kalos, and T. A. Brunner. "Obtaining Identical Results on Varying Numbers of Processors in Domain Decomposed Particle Monte Carlo Simulations". In: *Computational Methods in Transport: Granlibakken 2004*. Ed. by F. Graziani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 423–433. ISBN: 978-3-540-28125-2. DOI: 10.1007/3-540-28125-8_19. URL: http://dx.doi.org/10.1007/3-540-28125-8_19.

[22] W. Gropp and E. Lusk. "Reproducible Measurements of MPI Performance Characteristics". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 6th European PVM/MPI Users' Group Meeting Barcelona, Spain, September 26–29, 1999 Proceedings*. Ed. by J. Dongarra, E. Luque, and T. Margalef. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 11–18. ISBN: 978-3-540-48158-4. DOI: 10.1007/3-540-48158-3_2. URL: http://dx.doi.org/10.1007/3-540-48158-3_2.

[23] Y. He and C. Ding. "Using Accurate Arithmetics to Improve Numerical Reproducibility and Stability in Parallel Applications". English. In: *J. Supercomput.* 18 (3 2001), pp. 259–277. ISSN: 0920-8542. DOI: 10.1023/A:1008153532043. URL: http://dx.doi.org/10.1023/A%3A1008153532043.

[24] J.-M. Hervouet and J.-M. Janin. "Matrix storage and matrix-vector product in Finite Elements". In: *WIT Transactions on Ecology and the Environment* (1998).

[25] J.-M. Hervouet. *Hydrodynamics of free surface flows: Modelling with the finite element method.* English. John Wiley & Sons, 2007, pp. xvii+341. ISBN: 9780470035580. DOI: 10.1002/9780470319628.

[26] J.-M. Hervouet and openTelemac's group. "Private communication". Dec. 2015.

[27] Y. Hida, X. S. Li, and D. H. Bailey. "Algorithms for Quad-Double Precision Floating Point Arithmetic". In: *Computer Arithmetic, IEEE Symposium on* 0 (2001), p. 0155. DOI: http://doi.ieeecomputersociety.org/10.1109/ARITH.2001.930115.

[28] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. 2nd. SIAM, 2002, pp. xxx+680. ISBN: 0-89871-521-0.

[29] M. S. Horritt et al. "Comparing the performance of a 2-D finite element and a 2-D finite volume model of floodplain inundation using airborne SAR imagery". In: *Hydrological Processes* 21.20 (2007), pp. 2745–2759. ISSN: 1099-1085. DOI: 10.1002/hyp.6486. URL: http://dx.doi.org/10.1002/hyp.6486.

[30] S. Hunold. "A Survey on Reproducibility in Parallel Computing". In: *CoRR* abs/1511.04217 (2015). URL: http://arxiv.org/abs/1511.04217.

[31] *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. Reprinted in SIGPLAN Notices, 22(2):9–25, 1987. IEEE Computer Society. New York: Institute of Electrical and Electronics Engineers, 1985.

[32] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. New York: Institute of Electrical and Electronics Engineers, Aug. 2008, p. 58. ISBN: 0-7381-5753-8 (paper), 0-7381-5752-X (electronic). DOI: http://doi.ieeecomputersociety.org/10.1109/IEEESTD.2008.4610935.

[33] Intel. *Using the RDTSC Instruction for Performance Monitoring*. Tech. rep. Intel Corporation, 1997.

[34] F. Jézéquel, P. Langlois, and N. Revol. "First steps towards more numerical reproducibility". In: *ESAIM: Proceedings* 45 (Sept. 2014). Ed. by J. S. Dhersin, pp. 229–238. ISSN: 1270-900X. DOI: 10.1051/proc/201445023.

[35] W. Kahan. *Doubled-precision ieee standard 754 floating-point arithmetic*. Unpublished manuscript. 1987.

[36] G. Karypis and V. Kumar. *METIS – Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*. Tech. rep. 1995.

[37] D. E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Third. Reading, MA, USA: Addison-Wesley, 1998, pp. xiii+688. ISBN: 0-201-89684-2.

[38] P. Kornerup et al. "On the Computation of Correctly Rounded Sums". In: *IEEE Trans. Computers* 61.3 (2012), pp. 289–298.

[39] P. Langlois and N. Louvet. "More Instruction Level Parallelism Explains the Actual Efficiency of Compensated Algorithms". 11 pages. July 2007. URL: https://hal.archives-ouvertes.fr/hal-00165020.

[40] P. Langlois, R. Nheili, and C. Denis. "Numerical Reproducibility: Feasibility Issues". In: *NTMS'2015: 7th IFIP International Conference on New Technologies, Mobility and Security*. IEEE, IEEE COMSOC & IFIP TC6.5 WG. Paris, France, July 2015, pp. 1–5. DOI: 10.1109/NTMS.2015.7266509.

[41] P. Langlois, R. Nheili, and C. Denis. "Recovering numerical reproducibility in hydrodynamic simulations". In: *23rd IEEE International Symposium on Computer Arithmetic*. Ed. by J. H. P. Montuschi M. Schulte, S. Oberman, and N. Revol. ISBN 978-1-5090-1615-0. (Silicon Valley, USA. July 10-13 2016). IEEE Computer Society, July 2016, pp. 63–70. DOI: 10.1109/ARITH.2016.27.

[42] P. Langlois et al. *Less Hazardous and More Scientific Research for Summation Algorithm Computing Times*. Research Report RR-12021. Manuscrit soumis à Science of Computer Programming. Lirmm, Sept. 2012. URL: http://hal-lirmm.ccsd.cnrs.fr/lirmm-00737617.

[43] C. Q. Lauter. *Basic building blocks for a triple-double intermediate format*. Tech. rep. 2005-38. http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2005/RR2005-38.pdf: LIP, École Normale Supérieure de Lyon, Sept. 2005.

[44] N. Louvet. "Algorithmes compensés en arithmétique flottante : précision, validation, performances". PhD thesis. Université de Perpignan Via Domitia, Nov. 2007.

[45] *MESSAGE PASSING INTERFACE FORUM*. Mpi : A message-passing interface standard. Rapport technique. http://www.mpi-forum.org/docs/. 2012.

[46] O. Møller. "Note on Quasi Double-Precision". In: *BIT* 5 (1965), pp. 251–255.

[47] J.-M. Muller. *On the definition of ulp(x)*. Tech. rep. RR-5504. INRIA, Feb. 2005, p. 16. URL: https://hal.inria.fr/inria-00070503.

[48] J.-M. Muller et al. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010, p. 572. ISBN: 978-0-8176-4704-9.

[49] R. Nheili, P. Langlois, and C. Denis. "First improvements toward a reproducible Telemac-2D". In: *XXIIIrd TELEMAC-MASCARET User Conference*. Paris, France, Oct. 2016. URL: https://hal-lirmm.ccsd.cnrs.fr/lirmm-01371152.

[50] T. Ogita, S. M. Rump, and S. Oishi. "Accurate sum and dot product". In: *SIAM J. Sci. Comput.* 26.6 (2005), pp. 1955–1988.

[51] *Open TELEMAC-MASCARET. v.7.0, Release notes*. www.opentelemac.org. 2014.

[52] H. Oudin. "Méthode des éléments finis". Lecture. Nantes, France, Sept. 2008. URL: https://cel.archives-ouvertes.fr/cel-00341772.

[53] *POSIX Threads Programming*. http://www.llnl.gov/computing/tutorials/-workshops/workshop/pthreads/MAIN.html#WhyPthreads. Accessed 2003.

[54] J. Raamachandran. *Boundary and Finite Elements: Theory and Problems*. CRC Press, 2000. ISBN: 9780849309366. URL: https://books.google.fr/books?id=47cRZAZbhCgC.

[55] R. W. Robey, J. M. Robey, and R. Aulwes. "In search of numerical consistency in parallel programming". In: *Parallel Comput.* 37.4-5 (2011), pp. 217–229.

[56] S. M. Rump. "Ultimately fast accurate summation". In: *SIAM J. Sci. Comput.* 31.5 (2009), pp. 3466–3502.

[57] S. M. Rump, T. Ogita, and S. Oishi. "Fast High Precision Summation". In: *Non Linear Theory and Its Application* (July 2010). Ed. by IEICE, pp. 1–23.

[58] M. Sato et al. "Proceedings of the International Conference on Computational Science, ICCS 2011 A data and code model for reproducible research and executable papers". In: *Procedia Computer Science* 4 (2011), pp. 579 –588. ISSN: 1877-0509. DOI: http://dx.doi.org/10.1016/j.procs.2011.04.061. URL: http://www.sciencedirect.com/science/article/pii/S1877050911001190.

[59] J. R. Shewchuk. "Adaptive precision floating-point arithmetic and fast robust geometric predicates". In: *Discrete Comput. Geom.* 18.3 (1997). ACM Symposium on Computational Geometry (Philadelphia, PA, 1996), pp. 305–363. ISSN: 0179-5376.

[60] W. W. Smari et al. "New advances in High Performance Computing and simulation: parallel and distributed systems, algorithms, and applications". In: *Concurrency and Computation: Practice and Experience* 28.7 (2016). CPE-15-0606, pp. 2024–2030. ISSN: 1532-0634. DOI: 10.1002/cpe.3774. URL: http://dx.doi.org/10.1002/cpe.3774.

[61] I. M. Smith and L. Margetts. "The convergence variability of parallel iterative solvers." English. In: *Eng. Comput. (Bradf.)* 23.2 (2006), pp. 154–165. DOI: 10.1108/02644400610644522.

[62] L. Stanisic, A. Legrand, and V. Danjean. "An Effective Git And Org-Mode Based Workflow For Reproducible Research". In: *Operating Systems Review* 49 (2015), pp. 61 –70. DOI: 10.1145/2723872.2723881. URL: https://hal.inria.fr/hal-01112795.

[63] M. Taufer et al. "Improving numerical reproducibility and stability in large-scale numerical simulations on GPUs". In: *IPDPS*. IEEE, 2010, pp. 1–9.

[64] *The MPFR library*. URL = http://www.mpfr.org/.

[65] L. Thévenoux. "Code Synthesis to Optimize Accuracy and Speed in Floating-Point Arithmetic". Theses. Université de Perpignan Via Domitia, July 2014. URL: https://tel.archives-ouvertes.fr/tel-01143824.

[66] US General. *Accounting Office. Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia*. 1992.

[67] P. Vandewalle, J. Kovacevic, and M. Vetterli. "Reproducible research in signal processing". In: *IEEE Signal Processing Magazine* 26.3 (2009), pp. 37–47. ISSN: 1053-5888. DOI: 10.1109/MSP.2009.932122.

[68] O. Villa et al. "Effects of Floating-Point non-Associativity on Numerical Computations on Massively Multithreaded Systems". In: *CUG 2009 Proceedings*. 2009, pp. 1–11.

[69] Wikipedia. *Computer simulation — Wikipedia, The Free Encyclopedia*. [Online; accessed 2-August-2016]. 2016. URL: https://en.wikipedia.org/w/index.php?title=Computer_simulation&oldid=729252393.

HPC simulations in various scientific domains suffer from failures of numerical reproducibility because of floating-point arithmetic peculiarities. Different distributions of a parallel computation may yield different numerical results. Numerical reproducibility is a requested feature to facilitate the debug, the validation and the test of industrial or large software. In this thesis, we focus on the openTelemac software that implements finite element simulation for industrial and scientific hydrodynamics. We identify and analyze the sources of this reproducibility failure. We define and implement how to recover numerical reproducibility in two openTelemac modules. We also measure that the running time extra-cost of the reproducible version is reasonable enough in practice.

Keywords: floating-point arithmetic, reproducibility, finite element, domain decomposition, hydrodynamics simulation, compensation, openTelemac.

La non-reproductibilité numérique apparait dans divers domaines d'application de la simulation HPC. En effet, les différentes distributions d'un calcul parallèle peuvent mener à des résultats numériques différents, à cause des particularités de l'arithmétique flottante. Le besoin de reproductibilité numérique est motivé pour le débogage, le test et la validation des codes de calcul scientifique. Nous nous intéressons aux simulations par éléments finis en hydrodynamique implémentées dans le logiciel openTelemac qui est largement utilisé pour des applications industrielles et scientifiques. Nous identifions et analysons les sources de cette non-reproductibilité. Nous définissons et implementons comment récupérer la reproductibilité numérique de deux modules d'openTelemac. Nous mesurons que le sur-coût en terme de temps de calcul de la version reproductible est tout à fait raisonnable en pratique.

Mots-clés: arithmétique flottante, reproductibilité, éléments finis, décomposition de domaine, simulation hydrodynamique, compensation, openTelemac.