Implémentation et performances

Sommaire

Ce chapitre présente l'implémentation effectuée des différentes méthodes détaillées aux chapitres 2, 4 et 5. Nous présentons également les aspects performance de ces méthodes. Nous allons d'abord introduire la plate-forme *Arcane* sur laquelle se sont basés nos développements. L'architecture informatique de notre prototype est ensuite décrite. Puis, nous montrons comment les considérations de performance ont influencé la construction de ce prototype notamment au travers des modèles de programmation parallèle employés. Des résultats de performance obtenus sur différentes machines parallèles sont discutés en fin de chapitre.

6.1 La plate-forme *Arcane*

Arcane [GL09] est une plate-forme de développement pour les codes de simulation numérique volumes/éléments finis 2D/3D parallèles. Initiée en 2000 par le CEA/DAM, IFPEN collabore depuis 2007 à son développement. A l'opposé d'une plate-forme d'intégration, *Arcane* vise à proposer un ensemble d'outils et abstractions dédié au développement d'applications scientifiques de simulation pouvant s'exécuter sur des machines massivement parallèles. La plate-forme est aujourd'hui utilisée à IFPEN pour la mise au point d'applications de nouvelle génération en géosciences (modélisation de bassin, simulateurs de réservoir, stockage de CO_2, ...) à vocation industrielle.

La plate-forme *Arcane* a pour objectifs principaux de :

- simplifier au maximum l'écriture des modules numériques et d'environnement par la prise en charge des aspects informatiques,
- garantir de bonnes performances sur les machines parallèles actuelles et limiter l'impact des évolutions matérielles futures,

– assurer un cadre de développement de qualité : outils de mise au point, temps de compilation réduit, simplicité de mise en œuvre.

6.1.1 Architecture de la plate-forme

Arcane est écrite en C++ et fait environ 200 mille lignes de code. Le code est écrit pour être le plus simple possible pour les non-spécialistes du C++ en limitant les techniques complexes du langage (par exemple les *expressions templates*). La plate-forme fonctionne sous *Unix* et *Windows*.

Arcane est construite sur une architecture logicielle type *component*. Chaque fonctionnalité est fournie par un composant appelé *service* dans la terminologie *Arcane*. Un service doit se conformer à un comportement défini par un *contrat*. Un contrat est ici défini par un ensemble d'opérations et de contraintes que l'on appelle *interface* (au sens informatique). Un service donné implémente une telle interface. Par exemple, *Arcane* fournit une interface de maillage `IMesh` avec des opérations standards comme obtenir le nombre de mailles ou ajouter/retirer des mailles. Un service peut accéder à d'autres interfaces et ainsi utiliser d'autres services. De fait, un service agit comme un *plugin*. En utilisant cette approche, *Arcane* est hautement adaptable et n'est pas liée à une implémentation spécifique.

Arcane prend en charge par défaut une multitude de services dont en particulier :
– la gestion des structures liées au maillage,
– la gestion des grandeurs portées par le maillage à travers une base de données,
– le parallélisme,
– l'aide à la construction de briques logicielles utilisateurs,
– la gestion de ces briques logicielles et de leurs interactions,
– les options de configuration des briques utilisateurs (jeu de données),
– le mécanisme de protections / reprises,
– la fourniture de fonctions utilitaires (mathématiques, listing, temps d'exécution, ...),
– les outils d'analyse et d'aide à la mise au point,
– le mécanisme de retour-arrière,
– les sorties spécifiques de dépouillement (courbes et historiques).

6.1.2 Maillage et variable

Un élément central proposé par la plate-forme est la gestion des maillages. *Arcane* prend en charge les maillages non-structurés 2D ou 3D. Les différents éléments du maillage sont appelés des *entités*. Il existe 4 genres d'entités : nœuds (0D), arêtes (1D), faces (2D), et mailles (2D ou 3D). *Arcane* définit un grand nombres de types de face ou maille (triangle, tétraèdre, pyramide, quadrangle, etc) et il est possible de spécifier d'autres types. De plus, le maillage est complètement dynamique, c'est-à-dire qu'il est possible d'ajouter ou retirer des éléments durant la simulation.

La structure de données sous-jacente au maillage a été conçue pour que l'utilisation du maillage soit la plus indépendante possible de la dimension de l'espace. En particulier, un développeur pourra, si l'application s'y prête, écrire un code similaire pour traiter les cas à deux dimensions et à trois dimensions. Ainsi, une maille peut être un élément 2D ou 3D. Une face est toujours le bord d'une maille. Par exemple, un hexaèdre est composé de 6 faces et un quadrangle de 4 faces.

Il est possible de définir des grandeurs portées par les éléments du maillage. On parle alors de *variable*. Typiquement, ce sont les objets manipulés par l'utilisateur pour conceptualiser des quantitiés physiques, par exemple, une pression définie aux centres des mailles ou une vitesse définie au centre des faces. *Arcane* propose des outils d'énumérations et d'accès à la connectivité des entités permettant d'écrire des algorithmes génériques par rapport à la dimension du maillage. Une variable est définie par :
– Un type de donnée comme entier, réel ou booléen,
– Une dimension pouvant être scalaire, tableau 1D ou 2D,
– Une entité de maillage supportant les données : maille, nœud, face ou arête.

On présente ici un exemple d'utilisation de cette structure de maillage. Ce code définit le profil du système linéaire obtenu en résolvant une équation d'advection avec un schéma de type volumes finis à deux points.

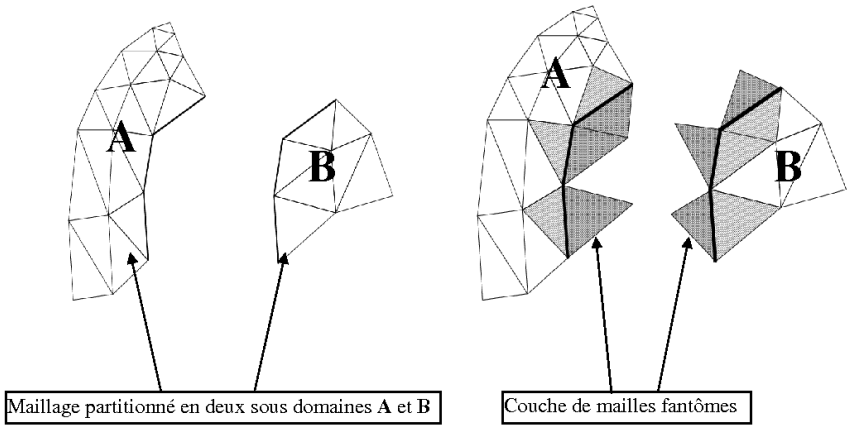
```
// On énumère les mailles présente dans le domaine
  ENUMERATE_CELL(icell, m_mesh -> ownCells()) {
    // On récupère la maille associée à l'itérateur icell
    const Arcane::Cell cell = *icell;
    // On demande l'indice associé à cette maille
    const Arcane::Integer id = m_cache_index[cell.localId()];

    // On énumère les faces associées à la maille cell
    ENUMERATE_FACE(iface, icell -> faces()) {
      // On s'intéresse uniquement aux faces internes
      if(not iface -> isBoundary()) {
        const Arcane::Face face = *iface;
        // On récupère les indices des deux mailles liées à la face face
        Arcane::Integer id_back = m_cache_index[face.backCell().localId()] ;
        Arcane::Integer id_front = m_cache_index[face.frontCell().localId()] ;

        // On donne au constructeur du système linéaire les indices des coefficients
        // de la matrices qui peuvent être non nuls
        m_linear_system_builder -> defineData(id, id_back) ;
        m_linear_system_builder -> defineData(id, id_back) ;
      }
    }
  }
```

6.1.3 Parallélisme

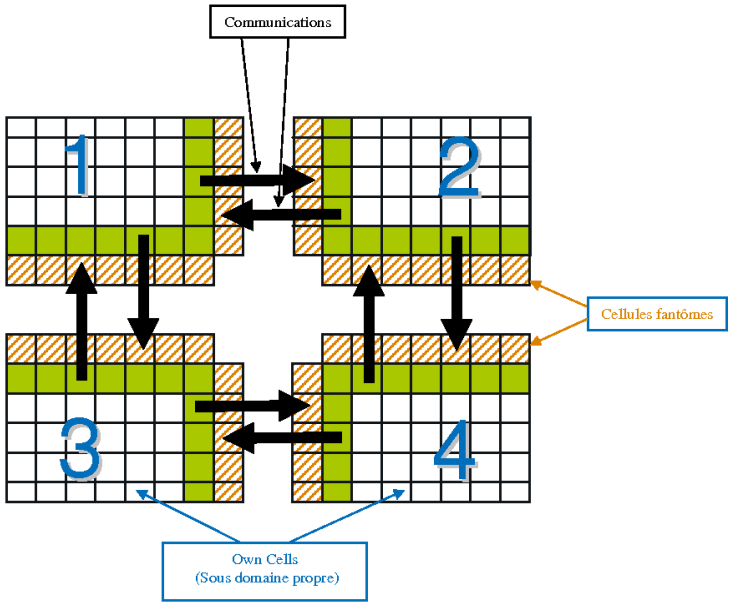
Arcane a été conçue pour s'exécuter sur des supercalculateurs composés de plusieurs milliers (et bien plus) de cœurs de calcul. Dans ce cas, le choix a été fait de partitionner le maillage en *sous-domaines* répartis sur chaque cœur de calcul. Chaque élément du maillage est *possédé* par un et un seul cœur de calcul. Chaque variable portée par le maillage est donc distribuée à travers tous les cœurs. On parle alors de la distribution des données. Les algorithmes calculant ces variables peuvent nécessiter l'utilisation d'un voisinage, les sous-domaines sont donc complétés par une ou plusieurs couches de mailles représentant une duplication d'éléments du maillage. Ces mailles dites *fantômes* sont la recopie des mailles voisines au sous-domaine. Cela forme donc une couche de recouvrement entre les sous-domaines.



La synchronisation des données entre sous-domaines est à effectuer explicitement par l'utilisateur :

```
\\ pressure est une variable
pressure.synchronize();
```

Notons que les mailles symétriques aux mailles fantômes sont appelées les mailles *partagées*.



Modèles de programmation parallèle

Les communications entre les sous-domaines sont effectuées suivant le paradigme *échange de messages* par l'utilisation d'un gestionnaire de parallélisme `IParallelMng`. Un sous-domaine peut donc envoyer et recevoir des messages des autres sous-domaines. *Arcane* met à disposition plusieurs implémentations du gestionnaire de parallélisme permettant une adaptation fine à l'architecture matérielle sous-jacente de la machine utilisée.

Mode distribué Ce mode est adapté aux architectures à mémoire distribuée. L'implémentation par défaut est basée sur *MPI* [WD96].

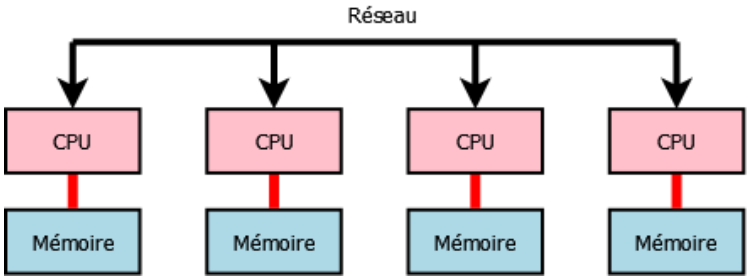


FIGURE 6.1 – Architecture distribuée

Ainsi, chaque sous-domaine est associé à un processus *MPI* ayant un espace d'adressage mémoire privé par rapport aux autres sous-domaines. Les communications se font à bas niveau par l'appel de routines *MPI* dédiées.

Mode partagé Ce mode est adapté aux architectures à mémoire partagée. L'implémentation par défaut est basée sur *TBB* [Rei07].

Ici, chaque sous-domaine est associé à un *thread*, c'est-à-dire un processus léger. Au lancement d'*Arcane*, autant de threads que de sous-domaines sont crées. Chaque thread gère ensuite indépendamment des autres

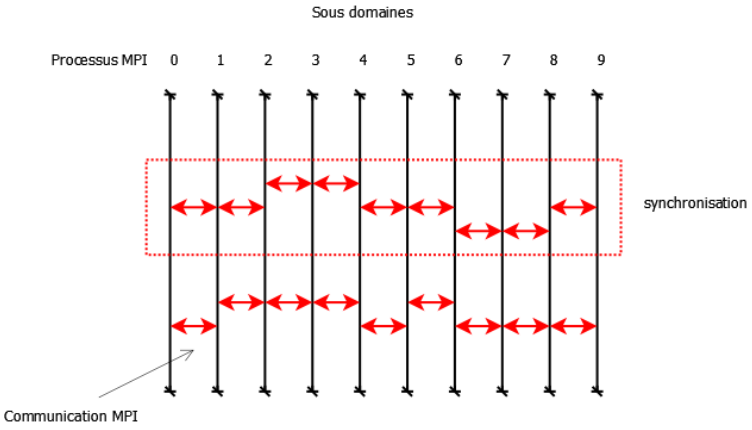


FIGURE 6.2 – Déroulement d'une application en mode distribué

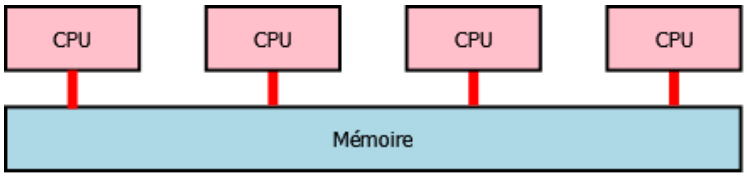


FIGURE 6.3 – Architecture partagée

le sous-domaine et les communications entre sous-domaines partagent le même formalisme qu'en mode distribué. L'avantage ici est de pouvoir substituer les appels *MPI* par des copies mémoires. De plus, chaque thread gérant la mémoire nécessaire pour les données associées au sous-domaine, les conflits mémoire sont réduits.

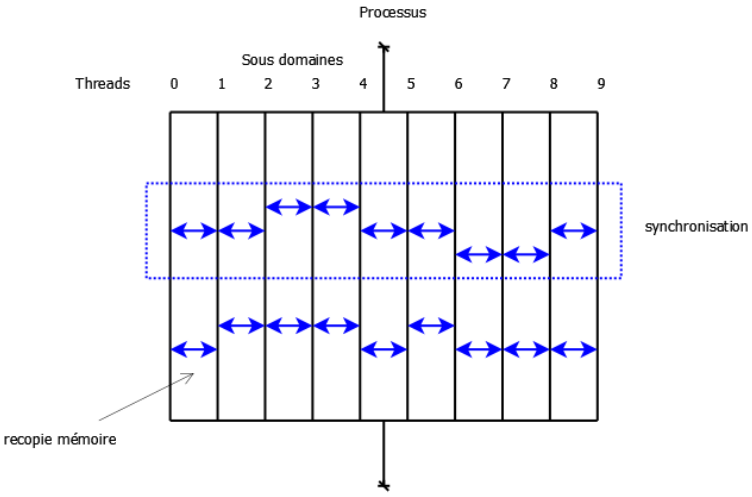


FIGURE 6.4 – Déroulement d'une application en mode partagé

Mode hybride Ce mode est adapté aux architectures massivement parallèles actuelles. En effet, les machines modernes sont composées de nœuds de processeurs à mémoire partagée connectés entre eux par

un réseau rapide. Ainsi, il y a une hiérachie mémoire où des ressources peuvent partager une partie de la mémoire.

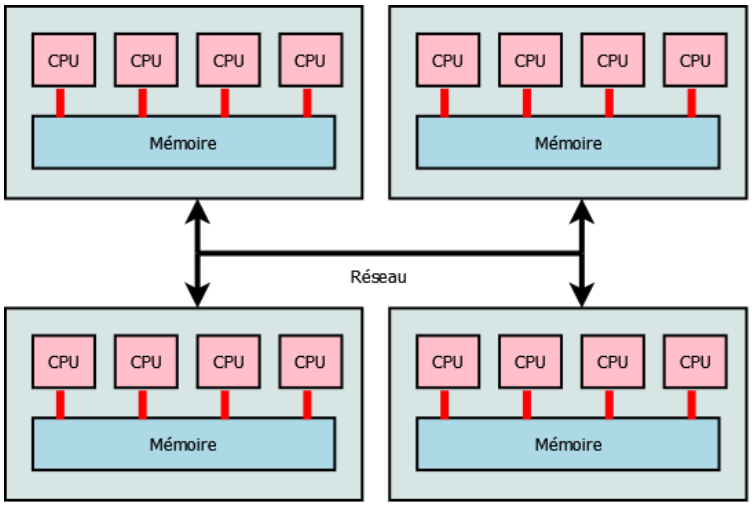


FIGURE 6.5 – Architecture traditionnelle d'un supercalculateur

Ce mode mélange les deux modes précédents. Les sous-domaines sont associés à des threads et en fonction de la localité du voisinage, les communications peuvent être de la recopie mémoire (partagée) ou de l'envoi de message *MPI* (distribué).

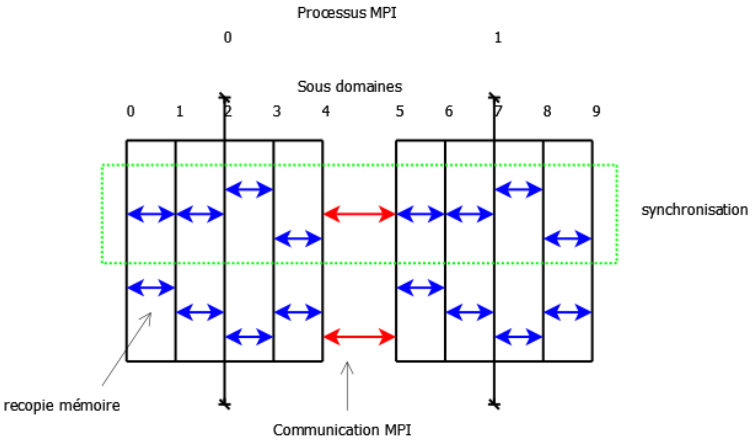


FIGURE 6.6 – Déroulement d'une application en mode hybride

Énumération parallèle

Indépendamment des modèles précédement présentés, l'énumération des entités du maillage peut également être effectuée en parallèle. Pour cela, une approche *multithread* est utilisée. Bien entendu, un prérequis pour obtenir des performances intéressantes est d'avoir des ressources disponibles pour cette parallélisation. Cela est donc utile lorque l'on exécute le prototype sur un supercalculateur (voir paragraphe 6.3).

Ainsi pour la méthode présentée au chapitre 5, le calcul des différentes fonctions de base ϕ_Σ et ψ_Σ définies sur des faces grossières Σ peut se faire en utilisant cette énumération parallèle. On construit alors une classe `ForComputer` qui contient la fonction `compute` qui calcule un certain nombres de fonctions de base.

```
class ForComputer
  {
  public:
    ForComputer(MultiThreadGlobalBasisFunctionComputerService& service)
      :  m_service(service) {}

    void compute(const Arcane::Integer begin, const Arcane::Integer size) {
      const std::vector<CoarseFace>& faces_to_compute = *m_faces_to_compute;
      for(Arcane::Integer iface = begin; iface < begin + size; ++iface) {
        const CoarseFace& face = faces_to_compute[iface];
        m_basis_functions[face]->compute();
      }
  }
```

On construit alors le foncteur `m_compute`

```
  m_compute = new RangeFunctorT<ForComputer>(m_computer, &ForComputer::compute);
```

Pour résoudre les problèmes de cellule en utilisant une énumération parallèle, on applique donc la fonction

```
  TaskFactory::executeParallelFor(0, faces_to_compute.size(), m_compute);
```

6.1.4 Principe de développement d'une application

Une application bâtie sur Arcane est constituée d'un ensemble fonctionnel de briques utilisateurs. Chacune représente un modèle physique, un algorithme mathématique ou une fonctionnalité noyau de la simulation numérique. En règle général, ces briques sont développées par des physiciens ou des mathématiciens. On parle ici de *modules* et de *services*. Chaque module est indépendant des autres et est caractérisé par :

- des *variables* représentant les paramètres physiques,
- des *points d'entrée* représentant les opérations mises à disposition par le module et pouvant être appelées par le noyau Arcane durant la simulation.

Les modules décrivent leurs variables, points d'entrée et options de configuration dans un fichier XML. À partir de ce fichier, Arcane génère, durant la compilation du module, les classes parentes qui définissent des attributs représentant les variables. Ces classes déclarent également les points d'entrée en méthodes abstraites et offrent les services techniques communs aux modules (accession au maillage, opérateur de trace, etc.). Les variables sont identifiées par leurs noms. En conséquence, les modules peuvent partager les données : deux modules utilisant une variable de même nom partagent les mêmes données. Un point d'entrée d'un module est une méthode rendant le module visible pour Arcane. La méthode peut ainsi être référencée dans un fichier XML décrivant la boucle en temps de l'application. La figure 6.7 résume l'organisation d'un programme sous Arcane.

Nous donnons ici un exemple de fichier XML décrivant un module Arcane :

```
<module name='Hydrodynamique'>
 <variables>
  <variable field-name='velocity' name='Velocity'  data-type='real3'
            item-kind='node' dim='0'/>
  <variable field-name='density' name='Density' data-type='real'
            item-kind='cell' dim='0'/>
 </variables>
 <entry-points>
  <entry-point method-name='computeForces' name='CalculDesForces'
       where='compute-loop' />
  <entry-point method-name='moveNodes'  name='DeplacementDesNoeuds'
       where='compute-loop' />
```

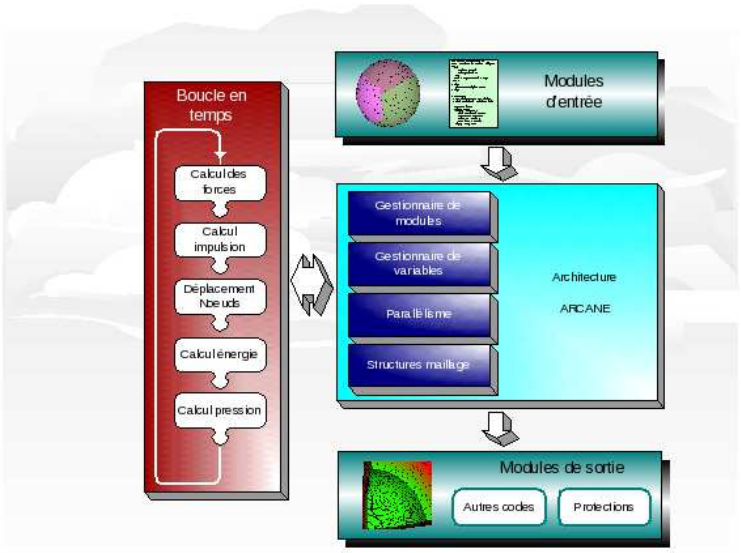


FIGURE 6.7 – Organisation d'un programme sous Arcane

```
 </entry-points>
 <options>
  <simple name='cfl' type='real' default='0.5'>
   <description>Définition de la CFL</description>
  </simple>
 </options>
</module>
```

Enfin, les modules peuvent partager des fragments de code au travers de composants dédiés appelés services. Les services peuvent être techniques comme un outil d'écriture parallèle sur fichier ou spécifiques au domaine d'application comme un solveur algébrique ou un schéma. Un service est vu comme un contrat représenté par un ensemble d'opérations rassemblées dans une interface. Les modules choisissent les implémentations des services dans leur fichier de configuration.

6.2 Architecture du prototype

On présente ici les différentes briques logicielles développées durant cette thèse. Notre but était de mettre au point un prototype suffisamment modulable pour permettre des évolutions et ajouts rapides.

6.2.1 Maillages

Arcane propose un lot d'outils avancés concernant le maillage. Toutefois, la problématique multi-échelle est centrée sur le lien entre les éléments de deux maillages distincts : un maillage grossier agrégeant les éléments d'un maillage fin. Ce lien a été formalisé dans un objet `MultiscaleMesh` contenant deux maillages *Arcane* `IMesh` ainsi que des éléments grossiers `CoarseCell` et `CoarseFace`. Chaque élément grossier est composé des différents éléments fins associés. De plus, une face grossière `CoarseFace` contient les mailles grossières dont elle est le bord.

Un lecteur de maillage cartésien `MultiscaleMeshReader` a également été conçu pour construire efficacement et en parallèle les maillages `Arcane` fin et grossier ainsi que le maillage multi-échelle `MultiscaleMesh`. Notons que la numérotation choisie des éléments est la numérotation classique $\{i, j, k\}$.

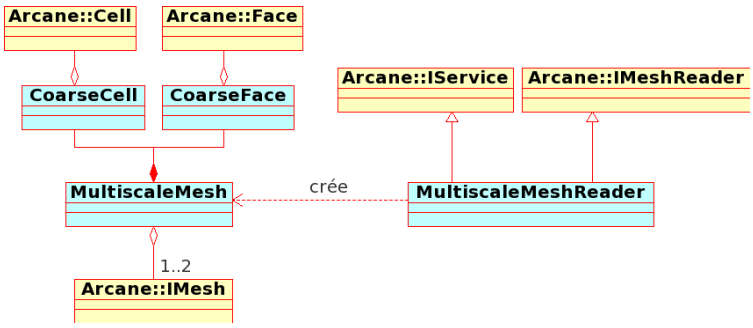


FIGURE 6.8 – Maillage multi-échelle

L'appel de ce lecteur est effectué dans l'initialisation de la plate-forme `Arcane`. Ce n'est pas à l'utilisateur de piloter explicitement la construction du maillage. En effet, ce lecteur est un service *Arcane* de type `IMeshReader`. L'approche environnement de programmation par composant prend son sens sur cet exemple d'ajout de fonctionnalité à la plate-forme.

6.2.2 Organisation du module principal

Le module principal est appelé `Multiscale`. Sa structure est présentée sur la figure 6.9. Ce module construit d'abord les données d'entrée en utilisant des modèles physiques représentés sur la droite du diagramme. Une fois ces paramètres connus, le module fait appel à différents services numériques pour résoudre les différentes équations permettant d'obtenir l'évolution de l'écoulement diphasique. En particulier, le module `Multiscale` requiert une résolution en pression et une résolution en saturation. En fonction des services utilisés pour résoudre ces équations, on pourra donc obtenir une résolution IMPES (voir paragraphe 2.3) ou IMPIMS (paragraphe 2.4). On peut également résoudre l'équation en pression en utilisant une des méthodes multi-échelles présentées aux chapitres 4 et 5.

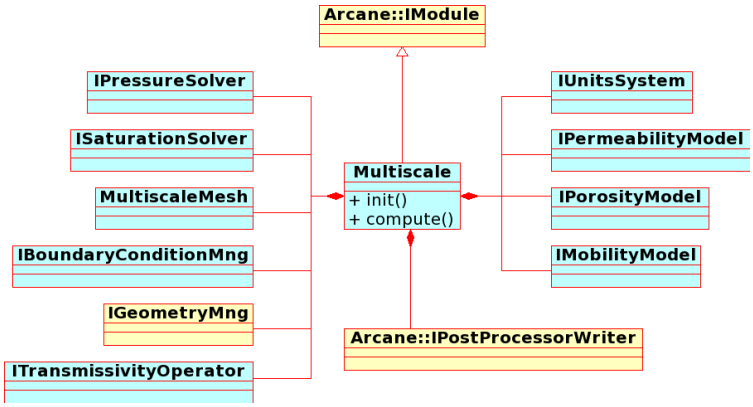


FIGURE 6.9 – Diagramme UML représentant le module `Multiscale`

6.2.3 Organisation des services physiques

Nous avons vu que la première étape dans la résolution est le calcul des données physiques. Ces calculs sont effectués en utilisant des services qui diffèrent selon le modèle considéré. Nous détaillons ici les différents services implémentés dans le prototype qui permettent d'obtenir la perméabilité (voir figure 6.10). Ainsi, on peut choisir des perméabilités constantes (`CstPermeabilityModel`) ou dépendantes des

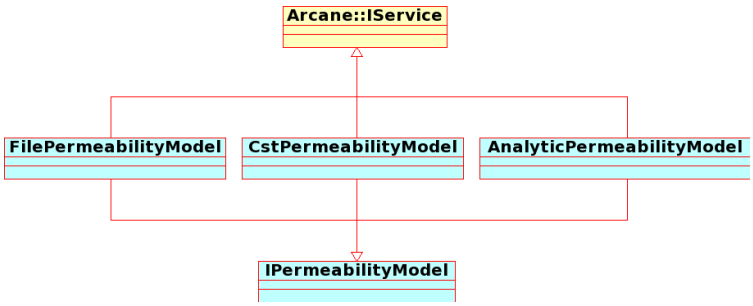


FIGURE 6.10 – Diagramme UML représentant les services définissant les modèles de perméabilité

coordonnées des mailles (`AnalyticPermeabilityModel`). Une troisième implémentation de cette interface, `FilePermeabilityModel`, permet de définir la perméabilité à partir d'un fichier.

D'autres services assez semblables sont disponibles pour modéliser la mobilité et la porosité.

6.2.4 Organisation des services numériques

Solveur en saturation

Nous avons vu au chapitre 2 que la saturation pouvait être résolue en utilisant un schéma implicite (paragraphe 2.3.2) ou explicite (paragraphe 2.4). Ces deux méthodes de résolution sont donc implémentées dans notre prototype (voir figure 6.11).

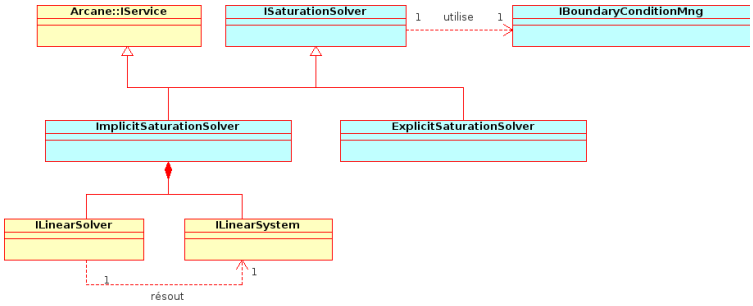


FIGURE 6.11 – Diagramme UML représentant les services implémentant les différentes méthodes de résolution du système en saturation

Solveur en pression

Les chapitres 2, 4 et 5 ont permis de définir trois méthodes pour résoudre le problème en pression (voir figure 6.12).

- La méthode utilisant des volumes finis définie au chapitre 2 est appliquée dans le service `ImplicitSolver`.
- La méthode *Allaire-Brizzi* présentée au chapitre 4 est utilisée dans le service `AllaireBrizziSolver`.
- Enfin, la méthode aux éléments finis mixtes multi-échelles [CH02] détaillée au chapitre 5 est implémentée dans le `MultiscaleSolver`.

Nous avons également mis en place une résolution de la pression par une méthode aux éléments finis de Lagrange dans le `FiniteElementSolver`. Le paragraphe 5.3.1 présente aussi un critère de mise à jour des fonctions de base dépendant de la variation des mobilités dans une maille grossière. Si on choisit d'appliquer cette mise à jour partielle, il suffit de préciser qu'on utilise le service `MobilityVariationUpdate`. Dans le cas contraire, on choisira le service `AlwaysUpdate`.

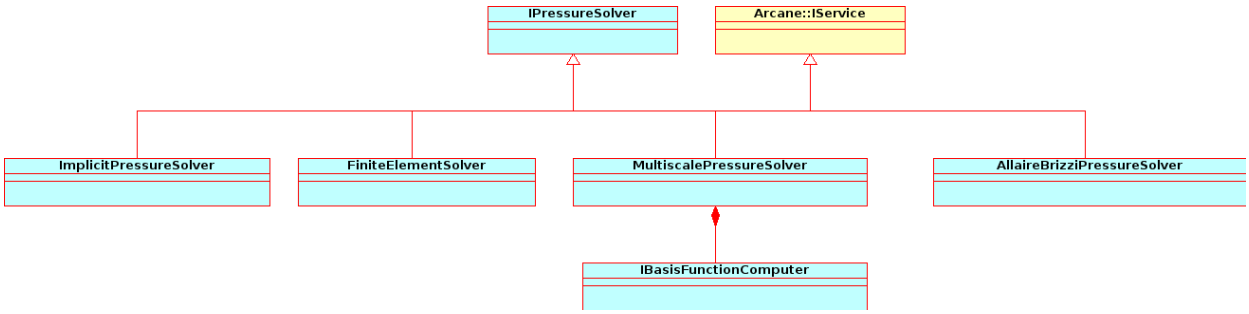


FIGURE 6.12 – Diagramme UML représentant les services représentant les différents solveurs en pression

Nous nous intéressons plus spécifiquement au service utilisant la méthode aux éléments finis mixtes multi-échelles. Ce service est détaillé dans la figure 6.13. Nous avons vu dans le chapitre 5 qu'il y avait deux façons différentes de définir les problèmes de cellule permettant de caluler les fonctions de base.

- La méthode présentée au paragraphe 5.2.1 est appelée `LocalComputer` car le calcul des fonctions de base dépend uniquement d'informations locales.
- La méthode présentée au paragraphe 5.3.2 est appelée `GlobalComputer` car elle nécessite l'utilisation d'une information globale, et plus précisément, du champ de vitesse d'un écoulement monophasique.

Pour ces deux méthodes, on peut choisir aussi d'effectuer le calcul des fonctions de base en utilisant une énumération parallèle (voir paragraphe 6.1.3). Nous avons donc également défini les services `MultiThreadGlobalComputer` et `MultiThreadLocalComputer`.

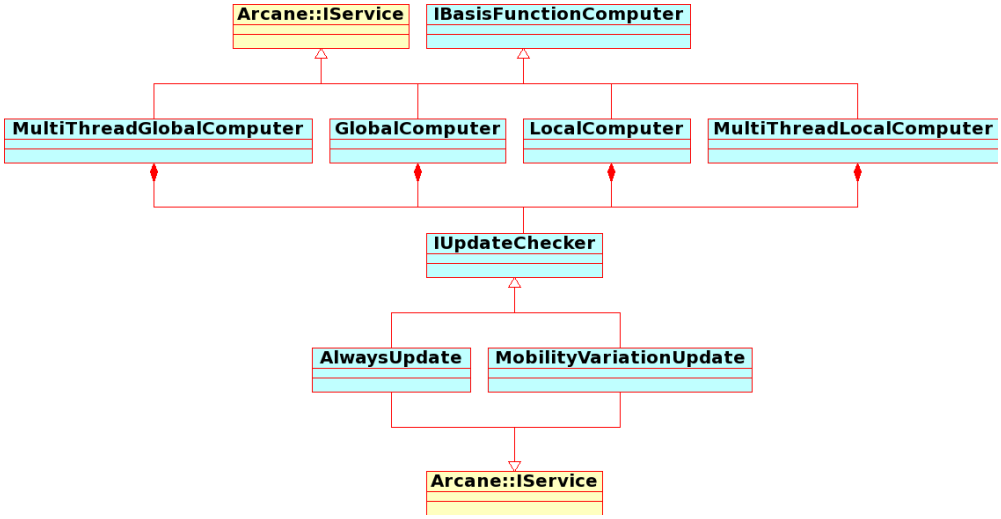


FIGURE 6.13 – Diagramme UML représentant les services définissant les différentes méthodes pour calculer les fonctions de base multi-échelles

6.3 Résultats de performance

Nous nous intéressons maintenant aux résultats de performance obtenus avec notre implémentation de la méthode multi-échelle présentée au chapitre 5.

6.3.1 Présentation des supercalculateurs utilisés

Nous présentons ici les différentes machines sur lesquelles nous avons pu tester le prototype présenté dans ce chapitre. Nous avons tout d'abord utilisé le supercalculateur d'IFPEN, ENER110. Ce supercalculateur,

mis en service le 29 janvier 2013, dispose d'une puissance de crête de 126 TFlops ce qui le place au 391^{e} rang mondial du top 500 des supercalculateurs de juin 2013 [Top13].

Ses caractéristiques sont les suivantes :

- un processeur est composé de 8 cœurs Intel Sandy Bridge 2,6 GHz,
- chaque nœud est composé de 2 processeurs c'est-à-dire de 16 cœurs et de 64 Go de RAM (donc 4Go par cœur),
- le supercalculateur comporte 378 nœuds de ce type,
- le réseau est de type infiniBand FDR.

Grâce au GENCI (Grand Équipement National de Calul Intensif) nous avons également pu disposer de temps de calcul sur le supercalculateur *CURIE* (15^{e} au top 500 de juin 2013) hébergé dans les locaux du CEA. Ce supercalculateur dispose de plusieurs types de nœuds. Les nœuds que nous avons utilisés pour tester notre prototype sont les nœuds "larges" qui ont les caractéristiques suivantes :

- un processeur est composé de 8 cœurs Intel Nehalem-EX 2,27 GHz,
- chaque nœud est composé de 16 processeurs c'est-à-dire 128 cœurs et 512 Go de RAM (donc 4Go par cœur),
- le supercalculateur comporte 360 nœuds de ce type,
- le réseau est de type infiniBand QDR.

6.3.2 Cas SPE10 3D

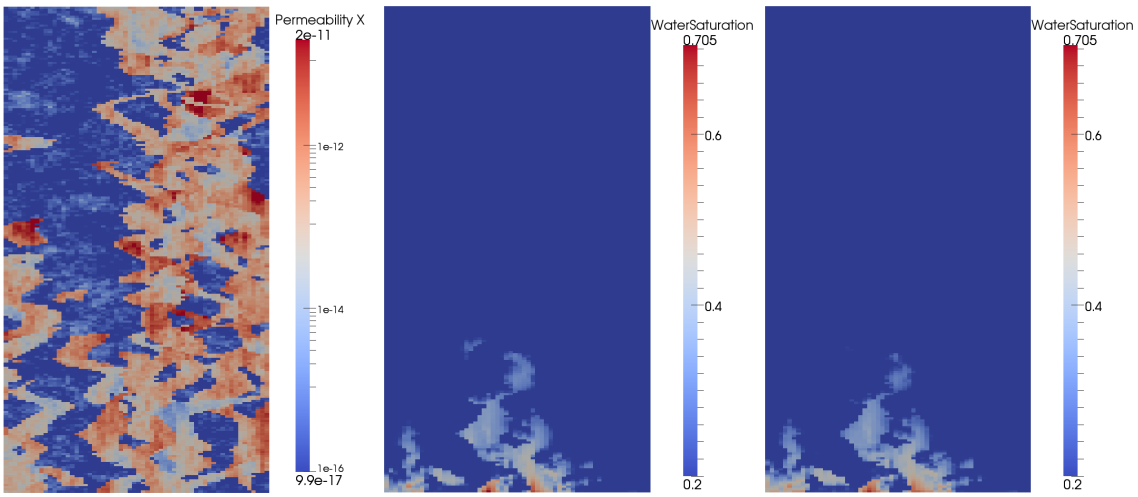
Dans ce paragraphe, nous considérons le cas SPE 10 à trois dimensions présenté dans le paragraphe 2.5.1. Pour ce cas, nous imposons une pression de 1000 psi et une saturation en eau égale à 1 sur le bord $y = 0$. Sur le bord où y est maximal, une pression de 500 psi et une saturation en eau égale à 0 sont fixées. Sur les autres bords, une condition de flux nul est imposée. Une première simulation est effectuée sur le maillage fin avec le schéma *IMPIMS*. Nous mesurons le temps nécessaire pour simuler cet écoulement jusqu'à ce que le volume poreux d'eau injecté soit égal à 1% du volume poreux total. Le pas de temps choisi correspond à dix fois le pas de temps maximal pour un schéma *IMPES* (voir équation (2.12)). Ce test est effectué en séquentiel puis en parallèle avec un nombre croissant de processeurs. Les temps de calcul présentés dans le tableau 6.1 sont en fait les temps totaux passés par le programme pour construire et résoudre les problèmes en pression. Les résultats présentés dans ce tableau sont obtenus en utilisant uniquement le parallélisme MPI pour partitionner le maillage en sous domaines.

1 proc.	2 proc.	4 proc.	8 proc.	16 proc.
2 939,69 s	1 571,63 s	817,27 s	454,81 s	293,56 s

Table 6.1 – Temps de calculs de la pression obtenus avec le solveur IMPIMS sur maillage fin en utilisant le parallélisme MPI

Les mailles sont ensuite agglomérées pour obtenir un maillage grossier comprenant 12 mailles dans la direction x, 20 dans la direction y et 5 dans la direction z. Comme dans le cas fin, nous mesurons le temps nécessaire pour atteindre un volume poreux d'eau injecté égal à 1% du volume poreux total. La méthode multi-échelle que l'on considère ici est la méthode globale avec une mise à jour partielle des fonctions de base avec $\varepsilon_{tol} = 0{,}7$ (voir paragraphe 5.3.1). Avant de comparer les temps de calcul, nous vérifions que cette solution multi-échelle approche assez bien la solution fine. La figure 6.14 présente les saturations de la couche $z = 40$ obtenues à l'état final avec la méthode multi-échelle ainsi que celles obtenues avec la solution fine. L'erreur relative en norme L^2 sur le domaine 3D complet entre la solution fine et la solution multi-échelle est de 4,84%. On en déduit donc que cette solution est assez proche de la solution fine.

Les temps de calcul des pressions sont présentés dans le tableau 6.2. Là aussi, le maillage est parallélisé avec MPI et il y a autant de sous-domaines que de processeurs.



(a) Perméabilité (b) Solution Fine (c) Solution Global70

FIGURE 6.14 – Comparaison des saturations obtenues avec la méthode multi-échelle et l'approche classique sur la couche $z = 40$ du cas SPE 10 3D

1 proc.	2 proc.	4 proc.	8 proc.	16 proc.
105,00 s	54,65 s	35,54 s	27,61 s	22,74 s

TABLE 6.2 – Temps de calculs de la pression obtenus avec le solveur IMPIMS multi-échelle en utilisant le parallélisme MPI

Nous remarquons donc que la méthode multi-échelle résout beaucoup plus rapidement le même problème : sur ce cas, en séquentiel, le temps de calcul est 28 fois plus faible en multi-échelle qu'avec une méthode classique. Cependant, on remarque que l'accélération apportée par la parallélisation est beaucoup plus faible avec la méthode multi-échelle qu'avec un solveur implicite (voir figure 6.15 détaillée plus tard). Cela est dû à la trop faible taille du système grossier ($12 \times 20 \times 85$). Le système linéaire n'est donc constitué que de quelques dizaines de milliers d'inconnues. L'utilisation de 16 processus MPI pour résoudre ce système n'est donc pas nécessaire. Pour mieux utiliser les ressources disponibles nous allons diminuer le nombre de processus MPI et utiliser une énumération parallèle de type multithread (voir paragraphe 6.1.3).

Tous les temps de calcul obtenus sont présentés dans le tableau 6.3. Dans ce tableau, *le nombre de processus MPI est égal au nombre de cœurs disponibles divisé par le nombre de threads.*

Nb cœurs	Solveur fin	1 thread	2 threads	4 threads	8 threads
1	2 939,69 s (1 proc. MPI)	105,00 s (1 proc. MPI)			
2	1 571,63 s (2 proc. MPI)	54,65 s (2 proc. MPI)	55,35 s (1 proc. MPI)		
4	817,27 s (4 proc. MPI)	35,54 s (4 proc. MPI)	33,24 s (2 proc. MPI)	43,91 s (1 proc. MPI)	
8	454,81 s (8 proc. MPI)	27,61 s (8 proc. MPI)	22,26 s (4 proc. MPI)	26,67 s (2 proc. MPI)	38,63 s (1 proc. MPI)
16	293,56 s (16 proc. MPI)	22,74 s (16 proc. MPI)	18,74 s (8 proc. MPI)	17,36 s (4 proc. MPI)	23,69 s (2 proc. MPI)

TABLE 6.3 – Temps de calculs de la pression obtenus avec les solveurs en pression fin et multi-échelle en fonction du nombre de cœurs utilisés et du nombre de threads ($Nb_{MPI} = Nb_{coeur}/Nb_{thread}$)

On remarque donc que l'utilisation de l'énumération parallèle pour le calcul des fonctions de base permet une meilleure utilisation de la puissance de calcul. Ainsi, avec 16 cœurs disponibles, l'utilisation de 4 threads au lieu d'un seul permet un gain de près de 24%.

Sur la figure 6.15, nous présentons l'efficacité des différentes méthodes. Tout d'abord, nous définissons l'efficacité par le temps de calcul total en parallèle divisé par le temps de calcul en séquentiel. Nous entendons par temps de calcul total le temps de calcul multiplié par le nombre de processeurs. Cette *efficacité* est donc un nombre inférieur à 1. Nous comparons alors l'efficacité du solveur fin avec celle de la méthode multi-échelle. On remarque que le solveur fin est beaucoup plus efficace que la méthode multi-échelle. Nous avions déjà remarqué que le maillage grossier comportait trop peu de mailles pour que l'utilisation d'un grand nombre de processeurs soit utile. Nous introduisons également l'efficacité de la méthode multi-échelle si on utilise le multithreading pour calculer les problèmes de cellule. Bien sûr, dans ce cas, on considère, pour chaque nombre de processeurs la valeur la plus faible observée dans le tableau 6.3. La figure 6.15 nous

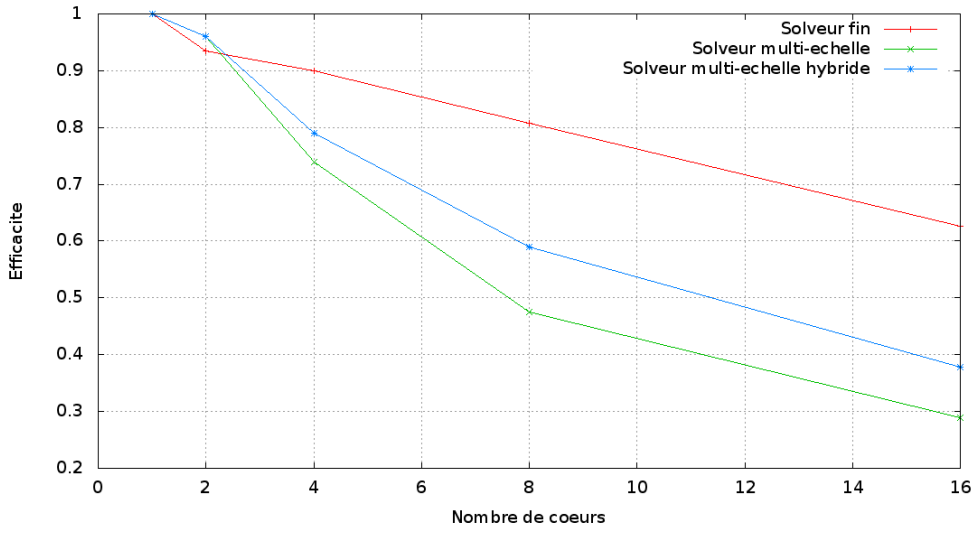


FIGURE 6.15 – Graphique présentant l'efficacité des différents solveurs

permet donc de déduire que l'utilisation de l'énumération parallèle augmente l'efficacité de la méthode.

Nous avons également testé notre prototype avec 16 cœurs sur un nœud "large" du supercalculateur CURIE. Comme indiqué précédemment, ce nœud possède une architecture différente et le but de ce second test est vérifier que le multithreading améliore également l'efficacité du code sur cette configuration de noeud. Le tableau 6.4 présente les résultats obtenus.

Solveur fin 16 proc. MPI	1 thread 16 proc. MPI	2 threads 8 proc. MPI	4 threads 4 proc. MPI	8 threads 2 proc. MPI
770,03 s	54,65 s	42,49 s	34,67 s	45,24 s

TABLE 6.4 – Temps de calculs de la pression obtenus avec les solveurs en pression fin et multi-échelle pour 16 cœurs en fonction du nombre de threads

On remarque que, sur cette seconde machine, le gain en performance entre le solveur fin et le solveur multi-échelle est du même ordre. Cependant, l'utilisation de plusieurs threads est ici beaucoup plus intéressante : avec 4 threads, le gain en temps de calcul par rapport à un parallélisme uniquement distribué est de plus de 35%.

6.3.3 Cube

On considère ici un cas théorique mis en place pour tester la capacité de la méthode multi-échelle sur des cas comportant un très grand nombre de mailles. Dans ce cas, le réservoir est un cube comportant 256 mailles dans chaque direction. Le nombre total de mailles est donc de près de 17 millions. Chaque maille est un cube de 5 mètres de côté. Le champ de perméabilité a été généré en utilisant des méthodes statistiques employées par le logiciel *Condor* développé par IFPEN (voir figure 6.16). On simule un écoulement diphasique dans

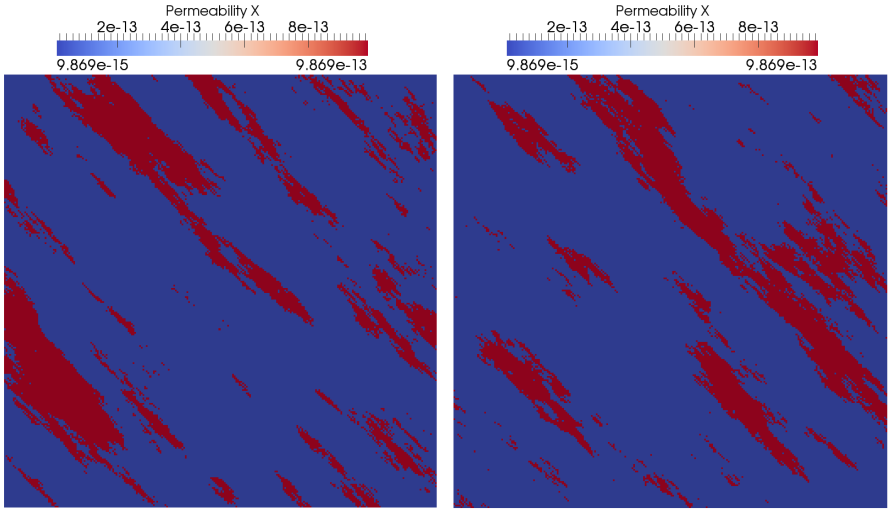


FIGURE 6.16 – Perméabilités du cube composé de 256 mailles dans chaque direction. À gauche pour $z = 200$ et à droite pour $z = 640$

ce réservoir en imposant une pression de 1000 psi et une saturation en eau égale à 1 sur le bord $x = 0$. Sur le bord où x est maximal, on impose et une pression de 500 psi et une saturation en eau égale à 0. Nous présentons dans le tableau 6.5 les résultats de performance obtenus lorsqu'on simule un écoulement dans ce cube jusqu'à atteindre un volume poreux d'eau injectée correspondant à 1% du volume poreux disponible. Le maillage grossier utilisé pour effectuer les simulations multi-échelles dans ce cas est composé de 32 mailles dans chaque direction.

Solveur fin 128 proc. MPI	1 thread 128 proc. MPI	4 threads 32 proc. MPI	8 threads 16 proc. MPI	16 threads 8 proc. MPI
22 153,31 s	694,38 s	466,80 s	576,02 s	773,23 s

TABLE 6.5 – Temps de calculs obtenus avec les solveurs en pression fin et multi-échelle pour 128 cœurs en fonction du nombre de threads

On remarque que, dans ce cas nécessitant de grandes capacités de calcul, la méthode multi-échelle est toujours beaucoup plus performante que la résolution classique (32 fois plus rapide). Comme dans le cas précédent, l'utilisation du multithreading pour le calcul des fonctions de base permet de réduire encore le temps de calcul. On atteint alors un temps de calcul plus de 47 fois plus faible.