Chapitre 6

Utilisation du parallélisme lors de la génération du maillage E.F.

6-1 Motivation :

L'évolution technologique des années 80 a permis de réduire les coûts de production des divers composants d'un ordinateur, en même temps qu'elle en augmente les performances à la fois en temps de calcul, en volume de traitement et en fiabilité. Le niveau technologique atteint est tel qu'il est maintenant possible de construire des architectures multiprocesseurs et de les utiliser efficacement, avec quelques processeurs pour des systèmes généraux (plusieurs dizaines) ou encore davantage pour des machines spécialisées (quelques milliers).

Le concept de parallélisme rompt avec l'approche classique qui consiste à diminuer le temps de calcul en effectuant plus vite chaque opération. En calcul parallèle, le gain de temps provient de la réalisation simultanée de plusieurs

opérations. L'algorithmique parallèle est étudiée depuis très longtemps, en particulier la complexité de certains problèmes parallèles. Mais les diverses architectures parallèles ont poussé les algorithmiciens à imposer de nouvelles contraintes à leurs modèles pour les rendre plus proches de la réalité.

La parallélisation d'une méthode requiert plusieurs phases : tout d'abord, il est commode de partir de diverses écritures de la méthode sous forme algorithmique, puis d'en faire une étude théorique de complexité pour bien saisir le parallélisme intrinsèque. Après avoir choisi la (ou les) bonne(s) version(s), on peut examiner son implémentation, c'est-à-dire la façon d'exécuter les instructions sur une machine cible. Cela pose en particulier les problèmes de placement d'instructions, de circulation des données, de gestion efficace des communications, etc.

Dans tous les cas, le développement d'ordinateurs parallèles conduit à réévaluer la plupart des algorithmes usuels en fonction de nouveaux critères de viabilité et de performance. Pour paralléliser un algorithme, on commence par partitionner le problème en sous-tâches. L'élaboration du graphe de précédence permet de définir les contraintes temporelles pour l'exécution des tâches. Reste alors à affecter les tâches aux processeurs, en respectant les contraintes de précédence ainsi que les contraintes matérielles liées à l'architecture de la machine. Ces dernières permettent de sélectionner, parmi toutes les versions parallèles obtenues, celle qui est la plus efficace pour l'ordinateur utilisé. En général, ces contraintes sont de deux types: accès limité aux données et problèmes de synchronisation des processeurs.

Comme les processus de génération de maillage en deux et trois dimensions peuvent être laborieux et longs, dans ce chapitre, on explorera la possibilité d'implanter les algorithmes des différentes étapes de génération de maillage sur des machines parallèles afin de diminuer le temps de génération de maillage en éléments finis.

6-2 Architecture et programmation parallèle :

Le traitement parallèle est une forme de traitement de l'information qui permet, en cours d'exécution, l'exploitation d'événements concurrents. Ces événements se situent à plusieurs niveaux : au niveau du programme, de la procédure, de l'instruction ou à l'intérieur d'une instruction. L'introduction du parallélisme à l'intérieur d'un programme peut se faire au niveau des procédures. Il nécessite la décomposition du programme en tâches, la recherche des relations de dépendance entre ces tâches et la programmation en parallèle des tâches indépendantes. Il peut se traiter au niveau du système d'exploitation (parallélisation automatique de programmes, compilateurs intelligents) ou au niveau algorithmique.

La notion de parallélisme recouvre de nombreux concepts, allant d'un niveau très fin (manipulation sur des bits) à un niveau plus grossier (découpage d'un programme en procédures complexes indépendantes). Plusieurs classifications ont été proposées dans la littérature [54,55]. La plus populaire, celle de Flynn [55], est basée sur le type d'organisation des flots de données et des flots d'instructions. Cependant, d'un avis général, cette classification ne permet pas de tenir compte de beaucoup de facteurs tels que le mode de fonctionnement des processeurs, l'organisation de la mémoire ou encore la granularité des processeurs. Partant de réalisations concrètes, on peut dégager trois grandes classes de machines parallèles : les machines généralistes à mémoire partagée, les réseaux de processeurs asynchrones à mémoire distribuée et les machines distribuées synchrones à parallélisme fin, massivement parallèles (machines généralistes ou spécialisées) [54]. Dans les modèles de machines parallèles des années 70, l’architecture de base est constituée d’une large mémoire commune partagée par plusieurs processeurs. Les problèmes architecturaux résident essentiellement dans les difficultés d’accès à cette mémoire. Pour améliorer le débit de la mémoire vers les processeurs, de nombreuses solutions ont été proposées : partage en plusieurs bancs mémoire, hiérarchisation de la mémoire par l’introduction de caches, généralisation de l’emploi d’opérateurs “pipeline”, conception

de réseaux d'interconnexion performants, etc. Les contraintes imposées par la mémoire et le réseau d'interconnexion sont telles que les interactions entre les processeurs sont importantes. On parle alors d'ordinateur *fortement couplé*. Les difficultés d'accès à la mémoire limitent le nombre de processeurs : au-delà de quelques dizaines de processeurs, les performances du réseau d'interconnexion se dégradent. Pour obtenir des machines à parallélisme massif, on a donc recours à des architectures où la mémoire est décentralisée et les liens limitées : chaque processeur dispose d'une mémoire locale à accès rapide et n'est relié qu'à un certain nombre de processeurs voisins.

L'augmentation du nombre de processeurs modifie énormément la structure de base de l'ordinateur. Les problèmes d'accès mémoire, deviennent cruciaux pour pouvoir acheminer des données au rythme du traitement des instructions par les processeurs. De même, les problèmes de communication entre processeurs sont importants. De nombreuses solutions ont été proposées à ces problèmes et plusieurs architectures ont vu le jour. Pour rester simple, nous présentons la classification la plus utilisée [55]. Celle-ci a pour critère de sélection le mode de contrôle des suites d'opérations élémentaires effectuées par les différents processeurs.

Le processus essentiel dans un ordinateur est l'exécution d'une suite d'instructions sur un ensemble de données. En général, les ordinateurs peuvent donc être classifiés selon la multiplicité des flots d'instructions et de données disponibles matériellement [55,56]. En conservant les initiales anglaises consacrées par l'usage, on obtient essentiellement les architectures suivantes :

- SIMD : (Single Instruction stream, Multiple Data stream) un seul flot d'instructions, plusieurs flots de données,
- SPMD : (Single Program, Multiple Data stream) chaque processeur dispose du même programme,
- MISD : (Multilpe Instruction stream, Single Data stream) plusieurs instructions successives traitent la même donnée,
- MIMD : (Multiple Instruction stream, Multiple Data stream) plusieurs flots

d'instructions et de données.

Un flot d'instructions est une suite d'instructions issues du contrôleur en direction d'un ou plusieurs processeurs. Un flot de données est une suite de données venant d'une zone mémoire en direction d'un processeur ou venant d'un processeur en direction d'une zone mémoire.

6-2-1 Architecture SIMD :

Cette structure est détaillée sur la figure (6-1). Plusieurs unités de traitement sont supervisées par la même unité de contrôle. Toutes les unités de traitement reçoivent la même instruction (ou le même programme, auquel cas on parle d'une structure SPMD) diffusée par l'unité de contrôle, mais opèrent sur des ensembles de données distincts, provenant de flots de données distincts. Chaque unité de traitement exécutant la même instruction au même instant, on obtient un fonctionnement synchrone des processeurs. La mémoire partagée peut être subdivisée en plusieurs modules. Dans ce cas, l'accès des unités de traitement aux différents modules se fait par un réseau d'interconnections. Un calculateur SIMD peut être considéré comme un monoprocesseur exécutant des instructions sur des tranches de plusieurs éléments. Il est donc particulièrement bien adapté aux traitements d'opérations vectorielles.

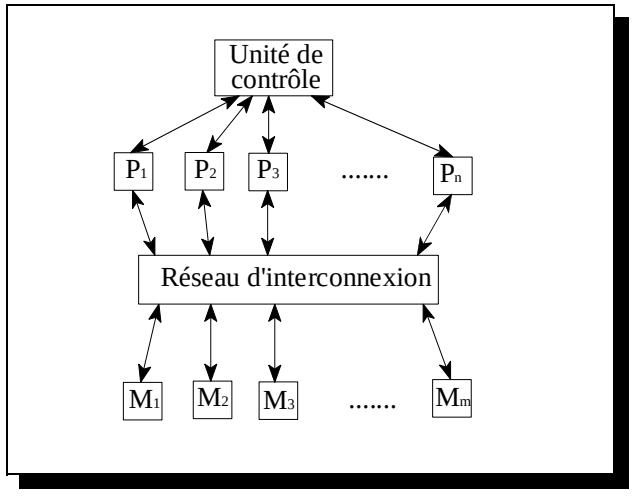

Figure 6-1 : Structure SIMD.

Dans un modèle SIMD, les processeurs peuvent communiquer entre eux soit en utilisant la mémoire partagée, ou soit par un réseau d'interconnections. Dans le modèle

SIMD avec mémoire partagée, les processeurs partagent une mémoire commune. Les processeurs peuvent accéder à cette mémoire commune simultanément. Il y a quatre catégories de modèles SIMD avec mémoire commune [57,58] :

i. accès exclusif en lecture et en écriture d'un même emplacement en mémoire;
ii. accès concurrent en lecture et exclusif en écriture ;
iii. accès concurrent en écriture et exclusif en lecture ;
iv. accès concurrent en lecture et concurrent en écriture ;

L'accès concurrent en lecture ne pose en principe aucun problème, mais l'accès concurrent en écriture pose des difficultés c'est-à-dire une façon déterministe d'anticiper le contenu de cet emplacement doit être donnée. On peut résoudre le conflit d'écriture de plusieurs façons :

- le processeur d'indice minimal peut écrire à cet emplacement ; les autres processeurs deviennent inactifs ;
- tous les processeurs ont accès en écriture en autant que les valeurs à stocker sont identiques. Autrement, l'accès est refusé à chacun d'eux ;
- la somme de toutes les valeurs à être écrites est stockée.

6-2-2 Architecture MIMD :

Les premiers ordinateurs parallèles à avoir effectivement connu un succès industriel sont les machines à mémoire partagée. Considérons p processeurs vectoriels reliés à une grande mémoire commune. Le fonctionnement des processeurs s'effectue en mode MIMD (Multiple Instruction stream, Multiple Data stream), c'est-à-dire que chaque processeur peut évoluer indépendamment des autres. Les échanges d'informations entre les processeurs se font par l'intermédiaire de la mémoire. En général, l'accès s'effectue en lecture simultanée et écriture exclusive (CREW). Les machines appartenant à cette classe possèdent un nombre relativement faible de processeurs (une dizaine tout au plus), mais chacun étant assez puissant. Plusieurs

tentatives ont été faites pour augmenter ce nombre en introduisant plusieurs niveaux hiérarchiques de mémoire.

La diminution des temps d'exécution n'est pas le seul but que les utilisateurs recherchent. Ils voudraient aussi disposer de plus en plus de mémoire pour pouvoir traiter des problèmes toujours plus grands. Ces deux aspects sont évidemment liés, car l'accès à des mémoires de grandes dimensions est lent. Comme la vitesse de base des unités de traitement augmente sans cesse, il faut organiser les mémoires de telle sorte que l'accès soit rapide. On dispose pratiquement toujours de petits dispositifs très rapides, locaux aux processeurs (registres ou mémoires caches) pour permettre un transfert anticipé (pipeline) des données et limiter ainsi les accès mémoire. De plus, la grande mémoire est divisée en modules distincts possédant chacun son propre canal d'entrées/sorties (les bancs mémoire). Ces bancs sont reliés aux processeurs par l'intermédiaire de bus ou d'un réseau d'interconnexion. Le premier est une ressource rapide, mais bloquant, le second permet en général tout accès des processeurs aux bancs, avec blocage de temps en temps.

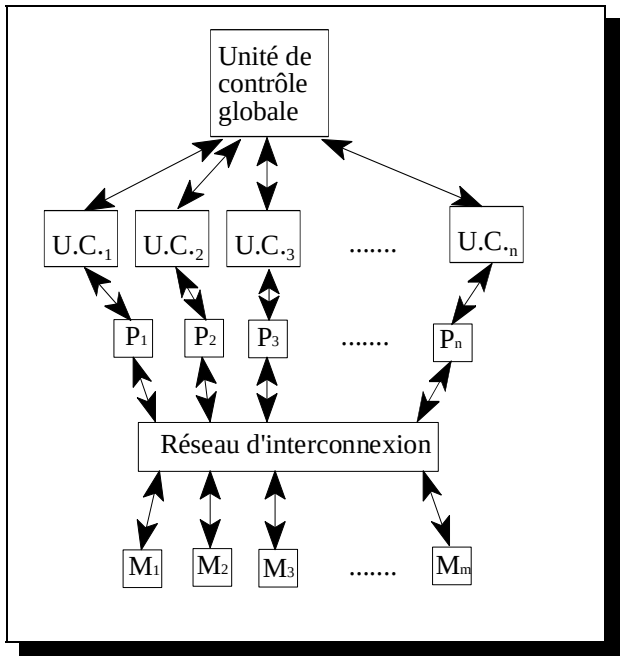

Figure 6-2 : Structure MIMD.

La figure (6-2) présente la structure MIMD. La différence profonde entre cette structure et la précédente est le fait que, dans ce cas, chaque processeur possède sa propre unité de contrôle. Les processeurs ont donc un fonctionnement indépendant (en particulier asynchrone) et exécutent des programmes différents.

Une autre classification peut être obtenue en distinguant les moyens de communication entre processeurs. Les contraintes imposées par la mémoire et le réseau d'interconnexion sont telles que les interactions entre les processeurs sont

importantes.

6-2-3 Architectures pipeline, vectorielle et MISD :

Le principe des architectures pipelines est le suivant [60] : on divise l'opération à effectuer en étapes de durées égales qui s'exécutent successivement. Les entrées d'une étape sont constituées par les sorties de l'étape précédente. Le pipeline est une unité matérielle qui reproduit cette division. Il est donc composé d'étages séparés par des registres nécessaires au stockage des données intermédiaires. Au sens de la classification de Flynn [54, 55], ces architectures relèvent du mode MISD. En effet, une même donnée est traitée par un flot multiple d'instructions élémentaires successives.

Un vecteur est un ensemble ordonné de n éléments. Chaque élément est un scalaire, nombre réel flottant, entier, booléen ou caractère. Un processeur vectoriel est une unité permettant de traiter des vecteurs. Il est constitué de registres de stockage et d'une ou plusieurs unités pipelines. Il traite des instructions vectorielles où toutes les combinaisons vecteurs-scalaires utiles sont disponibles.

Dans le modèle MISD, plusieurs opérations sont effectuées sur une même donnée (en parallèles). Les processeurs partagent une mémoire commune. Par exemple, si on veut déterminer si un entier positif z est premier, chaque processeur doit essayer un sous-ensemble des diviseurs possibles de z.

6-2-4 Représentation des algorithmes parallèles :

Un algorithme parallèle comprend normalement deux types d'instructions : séquentiel et parallèle. Pour les opérations parallèles :

i. Lorsque plusieurs traitements doivent être faits simultanément, nous avons :

Do traitement i **to** traitement j **in parallel**
traitement i : ...
traitement i+1 : ...
.
.
.
traitement j : ...

ii. Lorsque plusieurs processeurs doivent effectuer le même traitement simultanément, nous avons :

For i = j **to** k **do in parallel**
{traitement à effectuer par p_j, p_{j+1}, ..., p_k, simultanément}
End for

6-3 Résolution du système d'équations en parallèle :

Au moment de la construction du modèle géométrique en utilisant la représentation NURBS et aussi dans quelques autres étapes de génération de maillage, nous avons un problème de résolution d'un système d'équations. Normalement la résolution d'un système d'équations est parmi les tâches les plus lourdes d'un programme. Donc, il peut être avantageux d'exécuter cette partie en parallèle. Dans cette section, nous présentons quelques méthodes de résolution d'un système d'équations et les algorithmes correspondants en utilisant les machines parallèles.

6-3-1 La méthode d'élimination de Gauss-Jordan :

Considérons l'ensemble des équations linéaires :

$$a_{11}.x_1 + a_{12}.x_2 + a_{13}.x_3 + ... + a_{1n}.x_n = b_1 \quad (6.1)$$
$$a_{21}.x_1 + a_{22}.x_2 + a_{23}.x_3 + ... + a_{2n}.x_n = b_2$$
$$a_{31}.x_1 + a_{32}.x_2 + a_{33}.x_3 + ... + a_{3n}.x_n = b_3$$
$$.$$
$$.$$
$$.$$
$$a_{n1}.x_1 + a_{n2}.x_2 + a_{n3}.x_3 + ... + a_{nn}.x_n = b_n$$

Sous forme matricielle, on peut écrire cet ensemble d'équations comme :

A.X = B

Pour la résolution du système d'équations **A.X = B**, la méthode d'élimination de Gauss-Jordan consiste à multiplier une équation pivot par des constantes appropriées et à ajouter cette équation aux autres équations afin d'obtenir zéro à certains endroits et éventuellement des équations qui peuvent être résolues directement [61, 62]. La forme particulière d'élimination qu'on utilisera est la méthode de Gauss-Jordan. Dans cette méthode, un multiple approprié de la première équation est ajouté à chacune des autres équations, de manière à ce que les (n-1) équations résultantes n'aient pas de coefficients pour le terme x_1 . (Si la première équation n'a pas un terme impliquant x_1, on doit d'abord échanger deux équations pour en obtenir une avec un terme x_1 comme étant la première équation.) Puis, un multiple approprié de la prochaine équation est ajouté à toutes les équations pour éliminer le terme x_2 de toutes les équations à l'exception d'une.

Le procédé se poursuit jusqu'à ce que chaque équation contienne une seule inconnue, et que toutes les équations soient résolues. À chaque étape, le coefficient

qui est utilisé pour éliminer les autres coefficients est appelé le coefficient pivot.

Pour démontrer comment un algorithme peut être organisé pour exécuter le processus, on devrait construire quelques diagrammes. L'équation (6.1) aura comme représentation interne dans un ordinateur, les seules valeurs emmagasinées des coefficients $\mathbf{a_{11}}$ à $\mathbf{a_{nn}}$ et $\mathbf{b_1}$ à $\mathbf{b_n}$, correspondant aux variables indicées **A(I,J)** et **B(J)**. Puisque les signes d'addition "**+**", multiplication "**x**" et d'égalité "**=**" ne seront pas emmagasinés dans l'ordinateur de toute façon, omettons-les et écrivons seulement les coefficients et les constantes arrangés comme dans les équations, donc :

$$
\begin{array}{ccccccc}
a_{11} & a_{12} & a_{13} & \ldots & a_{1n} & b_1 & \\
a_{21} & a_{22} & a_{23} & \ldots & a_{2n} & b_2 & \\
a_{31} & a_{32} & a_{33} & \ldots & a_{3n} & b_3 & \\
. & & & & & & \\
. & & & & & & \\
. & & & & & & \\
a_{n1} & a_{n2} & a_{n3} & \ldots & a_{nn} & b_n &
\end{array}
\qquad (6.2)
$$

Pour rendre la notation plus uniforme, renommons b_1, b_2,..., b_n comme a_{1n+1},..., a_{nn+1} . Alors le tableau peut être écrit

$$
\begin{array}{ccccccc}
a_{11} & a_{12} & a_{13} & \ldots & a_{1n} & a_{1n+1} & \\
a_{21} & a_{22} & a_{23} & \ldots & a_{2n} & a_{2n+1} & \\
a_{31} & a_{32} & a_{33} & \ldots & a_{3n} & a_{3n+1} & \\
. & & & & & & \\
. & & & & & & \\
. & & & & & & \\
a_{n1} & a_{n2} & a_{n3} & \ldots & a_{nn} & a_{nn+1} &
\end{array}
\qquad (6.3)
$$

Comme première étape dans le processus d'élimination, on peut diviser la

première équation par a_{11} pour que le coefficient de x_1 devienne 1, ce qui donne les équations représentées par

$$
\begin{array}{lllll}
1 & a_{12}/a_{11} & a_{13}/a_{11} & \ldots & a_{1n}/a_{11} & a_{1n+1}/a_{11} \\
a_{21} & a_{22} & a_{23} & \ldots & a_{2n} & a_{2n+1} \\
a_{31} & a_{32} & a_{33} & \ldots & a_{3n} & a_{3n+1} \\
 & . & & & & \\
 & . & & & & \\
 & . & & & & \\
a_{n1} & a_{n2} & a_{n3} & \ldots & a_{nn} & a_{nn+1}
\end{array}
$$

Maintenant nous pouvons éliminer la variable x_1 de chacune des autres i (i = 2, ..., n) équations en multipliant la première équation par a_{i1} et en soustrayant de la $i^{ième}$ équation, donnant :

$$
\begin{array}{llll}
1 & a_{12}/a_{11} & a_{13}/a_{11} & \ldots \\
0 & a_{22}-a_{21}\,(a_{12}/a_{11}) & a_{23}-a_{21}\,(a_{13}/a_{11}) & \ldots \\
0 & a_{32}-a_{31}\,(a_{12}/a_{11}) & a_{33}-a_{31}\,(a_{13}/a_{11}) & \ldots \\
 & . & & \\
 & . & & \\
 & . & & \\
0 & a_{n2}-a_{n1}\,(a_{12}/a_{11}) & a_{n3}-a_{n1}\,(a_{13}/a_{11}) & \ldots
\end{array}
\qquad
\begin{array}{ll}
a_{1n}/a_{11} & a_{1n+1}/a_{11} \\
a_{2n}-a_{21}\,(a_{1n}/a_{11}) & a_{2n+1}-a_{21}\,(a_{1n+1}/a_{11}) \\
a_{3n}-a_{31}\,(a_{1n}/a_{11}) & a_{3n+1}-a_{31}\,(a_{1n+1}/a_{11}) \\
 & \\
 & \\
 & \\
a_{nn}-a_{n1}\,(a_{1n}/a_{11}) & a_{nn+1}-a_{n1}\,(a_{1n+1}/a_{11})
\end{array}
$$

A ce stade, nous avons éliminé la variable x_1 de toutes les équations à l'exception de la première, en se servant de a_{11} comme coefficient pivot. Remarquez que dans l'ordinateur, les nouveaux coefficients peuvent être emmagasinés à l'endroit qui contenait les anciens coefficients : a_{12}/a_{11} remplace simplement a_{12}, etc. Si cela est fait, le tableau ci-haut devient

$$
\begin{array}{cccccc}
1 & a_{12} & a_{13} & \ldots & a_{1n} & a_{1n+1} \\
0 & a_{22} & a_{23} & \ldots & a_{2n} & a_{2n+1} \\
0 & a_{32} & a_{33} & \ldots & a_{3n} & a_{3n+1} \\
. & & & & & \\
. & & & & & \\
. & & & & & \\
0 & a_{n2} & a_{n3} & \ldots & a_{nn} & a_{nn+1}
\end{array}
$$

et le procédé qui donne ce tableau à partir du tableau original est décrit par

a_{1j}/a_{11} ---> a_{1j} pour j = 2, ..., n+1

$a_{ij} - a_{i1}.a_{1j}$ ---> a_{ij} pour i = 2, ..., n ; j = 2, ..., n+1

Remarquez que ces étapes ne mettront pas effectivement $a_{11} = 1$ et $a_{i1} = 0$ pour i>1, c'est-à-dire qu'elles ne mettront pas la première colonne à un et zéros. Cependant puisqu'on sait qu'ils devraient être là, on peut simplement se rappeler ce fait et ne pas obliger l'ordinateur à passer par plusieurs étapes pour effectivement réaliser cette opération.

Maintenant, on peut éliminer la variable x_2 des équations 3 à n et de l'équation 1 par un processus analogue. Les étapes sont décrites par :

a_{2j}/a_{22} ---> a_{2j} pour j = 3, ..., n+1

$a_{ij} - a_{i2}.a_{2j}$ ---> a_{ij} pour i = 3, ..., n et j = 3, ..., n+1

et produisent un tableau de la forme

$$
\begin{array}{cccccc}
1 & 0 & a_{13} & \ldots & a_{1n} & a_{1n+1} \\
0 & 1 & a_{23} & \ldots & a_{2n} & a_{2n+1} \\
0 & 0 & a_{33} & \ldots & a_{3n} & a_{3n+1}
\end{array}
$$

.

.

.

0 0 a_{n3} ... a_{nn} a_{nn+1}

Si le processus est répété, jusqu'à l'élimination de a_{nn} on obtient éventuellement le tableau :

1 0 0 ... 0 a_{1n+1}

0 1 0 ... 0 a_{2n+1}

0 0 1 ... 0 a_{3n+1}

.

.

.

0 0 0 ... 1 a_{nn+1}

La solution recherchée est alors simplement obtenue en mettant $x_1 = a_{1n+1}$, $x_2 = a_{2n+1}$, etc.

Le processus précédent peut être résumé dans l'algorithme suivant

Algorithme Gauss-Jordan

(entrée n : nombre d'équations,

A(i, j): matrice de coefficients des équations avec

i = 1..n et j = 1..n + 1

sortie

A(i, n+1): vecteur de résultats, avec i = 1..n);

```
i, j, k          : entier;

Début
Pour i =1 jusqu'à n          faire
        pour j = i+1 jusqu'à n+1   faire
                A(i,j) = A(i, j)/A(i,i)          (si A(i,i) ≠ 0)
                pour k = 1 jusqu'à n faire
                        Si(i ≠ k) alors
                                A(k, j) = A(k,j) - A(k, i)*A(i, j)
                        fin si
                fin faire
        fin faire
fin faire.
Fin
```

Analyse

Dans cet algorithme, il y a trois boucles imbriquées presque toujours sur le nombre d'équations n, ce qui donne un temps d'exécution d'ordre $O(n^3)$. Ici, on suppose que les temps d'exécution de l'affectation, de la soustraction, de la division et de la multiplication sont fixés pour chaque opération.

Donc $t(n) \propto O(n^3)$

L'algorithme donné ci-dessus va s'exécuter avec difficulté si au moins un coefficient pivot est nul, puisqu'il va tenter de faire une division par zéro. Une façon d'éviter ce problème est de réarranger les équations toutes les fois qu'un élément nul est rencontré sur la diagonale.

Une autre façon, pas plus difficile à exécuter, est de réarranger les équations à chaque étape pour que le coefficient pivot à chaque étape soit non seulement différent de zéro, mais soit effectivement le coefficient le plus grand. Cette approche

évite non seulement la division par zéro mais aussi a tendance à accroître la précision en minimisant les erreurs finales [61, 62]. Elle a comme inconvénient le regroupement des inconnues à la fin du processus de regroupement causé par le réarrangement.

6-3-2 La parallélisation de la méthode d'élimination :

On suppose qu'on dispose d'assez de processeurs pour la résolution du système à n équations, c'est-à-dire $N \geq n^2+n$, ou encore le nombre N de processeurs est supérieur ou égal au n*(n+1), où n est l'ordre de la matrice des coefficients. Remarquez qu'en utilisant la machine avec N processeurs, on s'est limité à des matrices dont l'ordre ne dépasse pas $\propto \sqrt{N}$.

On suppose qu'on a une matrice de processeurs d'ordre n*(n+1) où chaque processeur travaille sur un élément de la matrice de coefficients, comme par exemple le processeur P_{ij} traite l'élément a_{ij}.

L'algorithme correspondant est exécuté sur des machines SIMD avec mémoire partagée à accès concurrent en lecture et exclusif en écriture.

Algorithme parallèle de Gauss-Jordan (n, A)

entrée

n : l'ordre de la matrice

A(n, n+1) : la matrice de coefficients $|A(n,n)| \neq 0$

sortie

A(i, n+1) i=1..n : vecteur de résultats

{Pour ce problème, on utilise le modèle SIMD SM avec accès concurrent en lecture et exclusif en écriture. Nombre de processeurs : $N = n^2 + n$.}

```
Début
piv, n, N, lig, col          : entier;
pivot, A(n, n+1)             : réel;
A(n, n+1), pivot, piv, n     : variable de mémoire partagée
                                    en lecture concurrente;
Pour piv = 1 jusqu'à n faire
        pivot = A(piv, piv)

        {Première étape:}
        For col = piv+1 to n+1 do in parallel
                A(piv, col) = A(piv, col)/pivot;
        End for

        {deuxième étape:}
        For lig = 1 to n do in parallel
                for col = 1 to n+1 do in parallel
                        Si (lig ≠ piv)
                                A(lig, col) = A(lig, col)) - A(lig, piv)*A(piv, col);
                        fin si
                end for
        End for
Fin faire
Fin.
```

Cet algorithme reçoit en entrée: **n** qui est le nombre d'équations, A: une matrice de coefficients **n*n+1**. Le déterminant de A(i=1...n, j=1...n) est non nul, A(i, n+1) i=1...n : est le vecteur des valeurs données en entrée, et en sortie, c'est le vecteur solution.

La matrice A est placée dans une mémoire partagée à accès concurrent en lecture. Remarquez que la procédure nécessite des opérations en lecture concurrente puisque plus d'un processeur aura besoin de lire A(piv, col) simultanément.

Analyse

L'analyse de cet algorithme se fait comme suit : on voit qu'il y a la présence d'une boucle qui traite **n** fois la matrice. Pour chaque traitement, un élément pivot de la matrice de coefficients est choisi. Cet élément se trouve sur la diagonale (ligne = piv, colonne = piv).

Dans la première partie de la boucle, il se fait une division de chaque élément de la ligne pivot par l'élément pivot de la matrice A.

Dans la deuxième partie de la boucle, il se fait une élimination (c'est-à-dire une mise à zéro) de tous les éléments de la colonne pivot sauf celui de la ligne pivot.

Les opérations dans ces deux parties sont faites en parallèle. Donc chaque étape est dans l'ordre de O(1). Pour ce qui est du temps d'exécution de l'algorithme général, il est de l'ordre de

$t(n) = n^* (O(1) + O(1))$ d'où $t(n) \propto O(n)$.

Le nombre de processeurs est de l'ordre de
$p(n) \propto O(n^2)$
d'où on déduit que le coût de cet algorithme est :
$c(n) = t(n)^* p(n) = O(n)^*O(n^2)$ ou encore
$c(n) \propto O(n^3)$

On peut trouver d'autres méthodes de parallélisation de la méthode d'élimination dans [57, 63, 64].

6-3-3 La méthode séquentielle optimale :

On peut résoudre le système d'équations AX = B en trouvant d'abord l'inverse de A (A^{-1}) et après en utilisant l'équation suivante :

$X = A^{-1} B.$

L'inverse de A peut être calculée de la façon suivante. Écrire d'abord [57] :

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} I & 0 \\ A_{21}A_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} A_{11} & 0 \\ 0 & C \end{bmatrix} \begin{bmatrix} I & A_{11}^{-1}A_{12} \\ 0 & I \end{bmatrix} \quad (6.4)$$

où les A_{ij} sont des sous-matrices (n/2)*(n/2) de A, et

$C = A_{22} - A_{21}.A^{-1}{}_{11}.A_{12}.$

Les matrices (n/2)*(n/2), "I" et "0" sont la matrice identité (dont les éléments de la diagonale principale sont '1' et tous les autres éléments sont zéros) et la matrice (zéro) nulle (dont les éléments sont zéros), respectivement. L'inverse de A est alors donné par le produit matriciel

$$A^{-1} = \begin{bmatrix} I & -A_{11}^{-1}A_{12} \\ 0 & I \end{bmatrix} \begin{bmatrix} A_{11}^{-1} & 0 \\ 0 & C^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -A_{21}A_{11}^{-1} & I \end{bmatrix} \quad (6.5)$$

où A^{-1} et C^{-1} sont calculés en appliquant le même processus récursif. Cela nécessite

deux inversions, six multiplications, et deux additions de matrices (n/2)*(n/2).

En notant le temps requis par ces opérations par les fonctions i(n/2), m(n/2), et a(n/2) respectivement, on obtient

$$i(n) = 2i(n/2) + 6m(n/2) + 2a(n/2).$$

puisque $a(n/2) = n^2/4$ et $m(n/2) = O((n/2)^x)$, où $2 < x < 2.5$ [57], on obtient $i(n) = O(n^x)$.

Alors, dans le calcul séquentiel, le temps requis pour calculer l'inverse d'une matrice n*n équivaut à un facteur multiplicatif, au temps requis pour multiplier deux matrices n*n. On sait aussi que le temps de la multiplication de A^{-1} et B est de l'ordre $O(n^x)$, $2 < x < 2.5$. Donc le temps total de résolution le système AX = B est dans l'ordre $O(n^x)$, où $2 < x < 2.5$.

6-3-4 La parallélisation de la méthode optimale :

A première vue, on peut penser qu'on peut diviser la matrice A en quelques sous-matrices. Par exemple, en utilisant l'équation (6.4), on voit que la matrice A est divisée en quatre sous-matrices qui s'expriment par la multiplication de trois sous-matrices. Mais, si on regarde l'équation (6.5), on voit que ce problème ne se subdivise pas en sous-problèmes distincts car l'inversion de la matrice A implique l'inversion des matrices A_{11} et C. Cependant, pour avoir la matrice C qu'on écrit à la suite, on doit avoir la matrice A_{11}^{-1}.

$$C = A_{22} - A_{21} \cdot A_{11}^{-1} \cdot A_{12}.$$

Donc, pour l'inversion de A, dans la première étape, on doit calculer l'inversion de A_{11} et dans la deuxième étape on doit calculer C et C^{-1}.

Dans l'équation (6.4), il est seulement possible de rendre parallèle les calculs matriciels de $A^{-1}_{11}.A_{12}$, $A_{21}.\ A^{-1}_{11}$ et de $C = A_{22} - A_2\ A_{11}^{-1}\ .\ A_{12}$.

Si on regarde l'équation (6.5), elle peut être écrite sous la forme suivante:

$$A^{-1} = \begin{bmatrix} I & -A_{11}^{-1}A_{12} \\ 0 & I \end{bmatrix} \begin{bmatrix} A_{11}^{-1} & 0 \\ 0 & C^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -A_{21}A_{11}^{-1} & I \end{bmatrix}$$

$$= \begin{bmatrix} I & -A_{11}^{-1}A_{12} \\ 0 & I \end{bmatrix} \begin{bmatrix} A_{11}^{-1} & 0 \\ -C^{-1}A_{21}A_{11}^{-1} & C^{-1} \end{bmatrix}$$

$$= \begin{bmatrix} A_{11}^{-1}+A_{11}^{-1}A_{12}C^{-1}A_{21}A_{11}^{-1} & -A_{11}^{-1}A_{12}C^{-1} \\ -C^{-1}A_{21}A_{11}^{-1} & C^{-1} \end{bmatrix}$$

Donc, pour arriver à l'inversion de la matrice A, après le calcul de A^{-1}_{11} et C^{-1}, on peut diviser le problème en au plus trois sous-parties qui peuvent être exécutées en parallèle. Ces parties sont les suivantes:

(a) $A^{-1}_{11} + A^{-1}_{11}\ .\ A_{12}\ .\ C^{-1}\ .\ A_{21}\ .\ A^{-1}_{11}$

(b) $-\ A^{-1}_{11}\ .\ A_{12}\ .\ C^{-1}$

(c) $-\ C^{-1}\ .\ A_{21}\ .\ A^{-1}_{11}$

Chaque partie est exécutée de manière séquentielle. Par exemple, pour la partie(a), on doit multiplier les matrices de la manière suivante:

tampon1 = $A_{21} \times A^{-1}_{11}$

tampon2 = $C^{-1} \times$ tampon1

tampon3 = $A_{12} \times$ tampon2

tampon4 = $A^{-1}_{11} \times$ tampon3

tampon5 = A^{-1}_{11} + tampon4

De manière générale, on peut expliquer les étapes d'une inversion de matrice à partir des séquences suivantes :

calculer A^{-1} :

- étape1 calculer A^{-1}_{11}
- étape2 calculer C et ($A_{22} - A_{21} . A^{-1}_{11} . A_{12}$)
- étape3 calculer C^{-1}
- étape4 calculer (a) (b) (c) en parallèle

Où

(a) $A^{-1}_{11} + A^{-1}_{11} . A_{12} . C^{-1} . A_{21} . A^{-1}_{11}$

(b) $- A^{-1}_{11} . A_{12} . C^{-1}$

(c) $- C^{-1} . A_{21} . A^{-1}_{11}$

De ces séquences, on voit bien que seule l'étape4 se fait en parallèle.

Mais, il est possible de faire chaque phase de multiplication en parallèle. Supposons qu'on utilise un algorithme de multiplication parallèle optimal avec les spécifications suivantes:

Algorithme : Multiplication_parallèle(n, matrice_1, matrice_2, matrice_resultat, N)

{entrée

n : l'ordre des matrices carrées: matrice_1

matrice_1 : première matrice de coefficients

matrice_2 : deuxième matrice de coefficients

N : le nombre de processeurs utilisés

sortie

matrice_resultat: résultat de la multiplication de matrices de coefficients}

Le temps d'exécution optimal de cet algorithme est dans l'ordre $O(n^x / N)$ avec $2 < x < 2.5$ [57]. On suppose aussi qu'on utilise un algorithme optimal d'addition et de soustraction parallèle avec les spécifications suivantes :

Algorithme : Addition_soustraction_parallèle(opération, n, matrice_1, matrice_2, matrice_resultat, N)

{entrée

opération : '+' addition
'-' soustraction

n : l'ordre des matrices carré:
matrice_1, matrice_2, matrice_resultat

matrice_1 : première matrice de coefficients

matrice_2 : deuxième matrice de coefficients

N : le nombre de processeurs utilisés

sortie

matrice_resultat: résultat d'addition ou de soustraction de matrices de coefficients}

L'ordre de cet algorithme est de $O(n^2 / N)$ [57]. À partir de ces deux algorithmes, on présente un algorithme d'inversion optimal et un algorithme de résolution de système d'équations optimal.

Algorithme : Résolution_système_équation_optimale_parallèle(n, A, B, N, X)

{entrée

n : l'ordre de la matrice de coefficients

A(n,n) : la matrice de coefficients

B(n) : le vecteur des valeurs données

N : le nombre de processeurs utilisés

sortie

X(n) : le vecteur de résultats}

Début

AI : matrice carrée d'ordre n.

Inversion_matricielle_optimale_parallèle(n, A, AI, N)

Multiplication_parallèle(n, AI, B, X, N)

Fin

Algorithme : Inversion_matricielle_optimale_parallèle(n, A, AI, N)

{entrée

n : l'ordre de la matrice de coefficients

A(n,n) : la matrice de coefficients

N : le nombre de processeurs utilisés

sortie

AI(n, n) : la matrice de résultats}

A_{11}, A_{12}, A_{21}, A_{22}, AI_{11}, AI_{12}, AI_{21}, AI_{22}, IA_{11}, CI, temp_1, temp_2, C
: matrice carrée d'ordre n/2.

IN : matrice identité négative et carrée d'ordre n/2 {pour changer le signe d'une matrice}

- Étape_1 : Partager la matrice A en quatre sous-matrices A_{11}, A_{12}, A_{21}, A_{22}.
- Étape_2 : {calculer la matrice $A^{-1}{}_{11}$}
  Inversion_matricielle_optimale_parallèle(n/2, A_{11}, IA_{11}, N)
- Étape_3 : {calculer la matrice C}
  Multiplication_parallèle(n/2, IA_{11}, A_{12}, temp_1, N)
  Multiplication_parallèle(n/2, A_{21}, temp_1, temp_2, N)
  Addition_soustraction_parallèle('-', n/2, A_{22}, temp_2, C, N)
- Étape_4 : {calculer la matrice C^{-1}}
  Inversion_matricielle_optimale_parallèle(n/2, C, CI, N)
- Étape_5 : {calculer AI_{11}, AI_{12}, AI_{21}, AI_{22}}

1. Étape_5-1 : {calculer AI_{11}}
   Multiplication_parallèle(n/2, $A_{21,}IA_{11}$, temp_1, N)
   Multiplication_parallèle(n/2, CI, temp_1, temp_2, N)
   Multiplication_parallèle(n/2, A_{12}, temp_2, temp_1, N)
   Multiplication_parallèle(n/2, IA_{11}, temp_1, temp_2, N)
   Addition_soustraction_parallèle('+', n/2, IA_{11}, temp_2, AI_{11}, N)
2. Étape_5-2 : {calculer AI_{12}}
   Multiplication_parallèle(n/2, A_{12}, CI, temp_1, N)
   Multiplication_parallèle(n/2, IN, IA_{11}, temp_2, N) {temp_2 = - IA_{11}}
   Multiplication_parallèle(n/2, temp_2, temp_1, AI_{12}, N)
3. Étape_5-3 : {calculer AI_{21}}

Multiplication_parallèle(n/2, A_{21}, IA_{11}, temp_1, N)

Multiplication_parallèle(n/2, IN, CI, temp_2, N)

Multiplication_parallèle(n/2, temp_2, temp_1,$AI_{21,}$ N)

4. Étape_5-4 : {calculer AI_{22}}

Multiplication_parallèle(n/2, IN, CI, AI_{22}, N)

- Étape_6 : Regrouper les sous-matrices AI_{11}, AI_{12}, AI_{21}, AI_{22} sous forme de matrice de sortie AI.

$$(AI = \begin{bmatrix} AI_{11} & AI_{12} \\ AI_{21} & AI_{22} \end{bmatrix})$$

Analyse de l'algorithme d'Inversion_matricielle_optimale_parallèle :

i. L'étape_1: Si de bonnes structures de données sont utilisées, il est possible d'avoir un temps d'exécution d'ordre O(1).

ii. L'étape_2: est dans l'ordre de O(t(n/2)) où t(n) est dans l'ordre d'exécution de l'algorithme Inversion matricielle optimale parallèle.

iii. L'étape_3: est dans l'ordre de $O((n/2)^x / N)$ $(2 < x < 2.5)$ parce qu'on effectue deux multiplications parallèles qui sont dans l'ordre de $O((n/2)^x / N)$ $(2 < x < 2.5)$ et une addition qui est dans l'ordre de $O((n/2)^2 / N)$.

iv. L'étape_4: est dans l'ordre de O(t(n/2))

v. L'étape_5: est dans l'ordre de $O((n/2)^x / N)$ $(2 < x < 2.5)$ parce qu'on effectue des opérations d'addition, de soustraction et de multiplication.

vi. L'étape_6: peut être dans l'ordre de O(1) en utilisant une bonne structure de données.

D'où un ordre d'exécution de l'algorithme est:

$t(n) = O(t(n/2)) + O((n/2)^x / N)$ $(2 < x < 2.5)$

$t(n) \propto O((n/2)^x / N)$ avec $(2 < x < 2.5)$.

Analyse de l'algorithme de Résolution_système_équation_optimale_parallèle :

L'ordre d'exécution de l'algorithme est:

$t(n) = O((n/2)^x / N) + O((n)^x / N)$ $(2 < x < 2.5)$

$t(n) \propto O((n)^x / N)$ avec $(2 < x < 2.5)$.

6-4 Parallélisation des algorithmes de génération de maillage en 2D :

Dans cette section, nous essayons de présenter les algorithmes parallélisables lors de la génération de maillage en deux dimensions.

Dans cette section, normalement on utilise le modèle SIMD SM avec accès concurrent en lecture et exclusif en écriture. On représente le nombre de processeurs par "N".

6-4-1 Modélisation géométrique de l'objet :

On rappelle l'algorithme suivant présente les étapes à suivre pour arriver à la représentation par le modèle B-spline cubique composite fermée (voir chapitre 4, section 4-2) :

1- Tripler les points aux coins.
2- Trouver les espacements nodaux $\{\nabla_i\}$:
   a. *Intervalles de support* : Les intervalles nodaux supportant la courbe peuvent être approximés raisonnablement par la méthode des longueurs des cordes entre points successifs i.e. :

      $\nabla_i = | P_{i+1} - P_i |$, $i = 0, 1, ..., n-1$.

   b. *Intervalles auxiliaires* : Les intervalles auxiliaires doivent être choisis de

la façon suivante :

$$\nabla_{-1} = \nabla_{n-2} , \nabla_{-2} = \nabla_{n-3} , \nabla_{n+1} = \nabla_2 , \ \nabla_n = \nabla_1$$

c. *Intervalles entre les points triples* : Les intervalles entre les points triples sont nuls mais pour éviter la division par zéro dans les calculs, on choisit une valeur petite, mais assez grande, pour éviter la division par zéro.

3- Construction des systèmes linéaires pour calculer les points de contrôle inconnus:

(voir la section 3-3-3-2 du chapitre 3)

4- Construction de la matrice de système (équation 3.33) :

(voir la section 3-3-3-2 du chapitre 3)

5- Résolution du système d'équations pour $\{V_i\}$:

On peut écrire l'algorithme parallèle pour arriver à la représentation par courbe cubique B-spline composite fermée comme suit :

Algorithme parallèle de modélisation géométrique par courbe cubique B-spline composite fermée :

{on utilise le modèle SIMD SM avec accès concurrent en lecture et exclusif en écriture.}

N : le nombre de processeurs.

i,j : entier

```
1- {Tripler les points aux coins.}
    For i = 1 to N do in parallel
        Pour j = 1 jusqu'à (nombre de points aux coins / N) faire
            Si (i-1)*N + j ≤ nombre de points aux coins
                Tripler le point (i-1)*N+j
            Fin si
        Fin faire
    End for
```

2- Trouver les espacements nodaux $\{\nabla_i\}$:

a. *Intervalles de support* :

Pour i = 1 jusqu'à (n / N) faire {n : le nombre de points donnés}

For j = 1 to N do in parallel

Si $(i-1)*N+j \leq n$

$\nabla_{(i-1)*N+j-1} = | P_{(i-1)*N+j-1} - P_{(i-1)*N+j} |$

Si $\nabla_{(i-1)*N+j-1} = 0$ (*Intervalles entre les points triples*) on choisit une valeur petite mais assez grande pour éviter la division par zéro.

Fin si

Fin si

End for

Fin faire

b. *Intervalles auxiliaires* : Les intervalles auxiliaires doivent être choisis de la façon suivante :

$\nabla_{-1} = \nabla_{n-2}$, $\nabla_{-2} = \nabla_{n-3}$, $\nabla_{n+1} = \nabla_2$, $\nabla_n = \nabla_1$

3- Construction des systèmes linéaires pour calculer les points de contrôle inconnus:

Pour i = 1 jusqu'à (n / N) faire {n : le nombre de points donnés}

For j = 1 to N do in parallel

Si $(i-1)*N+j \leq n$

Calculer $f_{(i-1)*N+j-1}$,$g_{(i-1)*N+j-1}$ et $h_{(i-1)*N+j-1}$.

(voir la section 3-3-3-2 du chapitre 3)

Fin si

End for

Fin faire

4- Construction de la matrice de système (équation 3.33) :

(voir la section 3-3-3-2 du chapitre 3)

5- Résolution du système d'équations pour $\{V_i\}$ en utilisant l'algorithme de

Résolution_système_équation_optimale_parallèle : (voir la section 6-3)

Analyse d'algorithme parallèle :

- L'étape 1 est d'ordre (nombre de points aux coins / N) ou O(n/N).
- L'étape 2 est dans l'ordre de O(n/N).
- L'étape 3 est dans l'ordre de O(n/ N).
- L'étape 4 est dans l'ordre de O(1)
- L'étape 5 est dans l'ordre de $O(n^x / N)$ $(2 < x < 2.5)$

D'où un ordre d'exécution de l'algorithme est:

$t(n) = 3O(n/N) + O(1) + O(n^x / N)$ $(2 < x < 2.5)$

$t(n) \propto O(n^x / N)$ avec $(2 < x < 2.5)$.

6-4-2 Création des quadrants et de l'arbre quadtree :

On peut présenter la procédure de création des quadrants et l'arbre quadtree sous forme d'algorithme comme suit (voir chapitre 4, section 4-3) :

Algorithme : Tester_subdiviser_quadrant (quadrant, niveau)

```
Début
    Si Max_paramètre_de_contrôle > ( niveau + 1 ) alors
        Diviser_quadrant_en_quatre;
        Pour i = 1 à 4 faire
            Tester_subdiviser_quadrant (quadrant(i), niveau+1)
        Finfaire;
    Sinon
```

Diviser_quadrant_en_quatre;

Finsi;

Fin;

On peut écrire l'algorithme parallèle pour la procédure de création des quadrants et l'arbre quadtree comme suit :

Algorithme : Tester_subdiviser_quadrant_parallèle (quadrant, niveau,no_proc)
{on utilise le modèle SIMD SM avec accès concurrent en lecture et exclusif en écriture.}

No_p : variable de mémoire partagée de type sémaphore initialisée à N (nombre de processeurs) qui désigne le nombre de processeurs disponibles à un instant donné.

Début

Si Max_paramètre_de_contrôle > (niveau + 1) alors

Diviser_quadrant_en_quatre;

S'il existe quatre processeurs libres (No_p ≥ 4)

Prendre quatre processeurs libres (No_p modifié à No_p - 4)

For i = 1 to 4 do in parallel {les quatre processeurs trouvés}

Tester_subdiviser_quadrant_parallèl (quadrant(i), niveau+1, i)

End for;

Sinon

Pour i = 1 à 4 faire

Tester_subdiviser_quadrant (quadrant(i), niveau+1)

{séquentiel}

Finfaire;

Finsi;

Sinon

```
            Diviser_quadrant_en_quatre;
        Finsi;
Fin;
```

Analyse des algorithmes :

Si on considère le temps d'exécution des opérations qui sont effectuées sur un quadrant est de l'ordre de O(1). Le temps d'exécution d'un algorithme séquentiel est de l'ordre du nombre d'éléments dans l'arbre soit $O(4^{niveaux})$. Pour l'algorithme parallèle, on doit arriver aux feuilles afin de terminer l'exécution d'algorithme. Donc si on avait assez de processeurs, cet algorithme serait de l'ordre du nombre de niveaux O(niveaux) parce que les éléments de chaque niveau de l'arbre exécutent en parallèle. Sinon cet algorithme est dans l'ordre $O(4^{niveaux} / N)$ (N est le nombre de processeurs).

6-4-3 Différence d'un niveau de subdivision :

L'algorithme suivant présente les étapes principales pour l'imposition d'un niveau de subdivision de différence (voir chapitre 4, section 4-4) :

Algorithme : Différence d'un niveau de subdivision de différence :

1. Construire la liste chaînée des quadrants feuilles de l'arbre quadtree.
2. Trouver les voisinages de chaque quadrant.
3. Pour tous les quadrants dans la liste chaînée :
    a. Si le niveau de subdivision du quadrant est plus petit que le niveau des quadrants voisins plus un alors :
        i. Subdiviser le quadrant en quatre quadrants
        ii. Modifier la liste chaînée des quadrants feuilles
            (1) Enlever le quadrant père

(2) Ajouter les quadrants fils à la fin de la liste chaînée

iii. Mise à jour des voisinages

On peut écrire l'algorithme parallèle de Différence d'un niveau de subdivision comme suit :

Algorithme : Différence d'un niveau de subdivision parallèle :

{on utilise le modèle SIMD SM avec accès concurrent en lecture et exclusif en écriture.}

N : le nombre de processeurs.

i,j : entier

Id=0 : variable de mémoire partagé avec accès concurrent en lecture et concurrent en écriture. On résout le conflit d'écriture de façon que tous les processeurs aient accès en écriture en autant que les valeurs à stocker sont identiques. Autrement, l'accès est refusé à chacun d'eux.

1. Construire la liste chaînée des quadrants feuilles de l'arbre quadtree.
2. Tant que Id = 0 faire
    Id = 1
    Pour i = 1 jusqu'à (nombre de quadrants dans la liste) / N faire
        For j = 1 to N do in parallel
            Si (i-1)*N+j $\leq$ (nombre de quadrants dans la liste)
                a.Trouver le voisinage du quadrant (i-1)*N+j
                b. Si le niveau de subdivision du quadrant est plus petit que le niveau des quadrants voisins plus un, alors :
                    i. Id = 0
                    ii. Subdiviser le quadrant en quatre quadrants
                    iii. Modifier la liste chaînée des quadrants feuilles
                        (1) Enlever le quadrant père

(2) Ajouter les quadrants fils dans la liste chaînée

Fin si

End for

Fin faire

Fin faire

Analyse des algorithmes :

Si on considère le temps d'exécution des opérations qui sont effectuées sur un quadrant est de l'ordre de O(1). Dans le pire cas, le temps d'exécution de l'algorithme séquentiel est de l'ordre O(maximum_niveau*nombre_des_quadrants) parce qu'un quadrant de niveau 1 peut être placé à coté d'un quadrant de niveau maximum. Pour l'algorithme parallèle, il y a aussi la possibilité du nombre de répétition égale à "maximum_niveau", quand un quadrant de niveau 1 est placé à coté d'un quadrant de niveau maximum donc cet algorithme est dans l'ordre de O(maximum_niveau*nombre_des_quadrants / N).

6-4-4 Classification des quadrants :

On peut présenter cette étape sous la forme de l'algorithme séquentiel suivant (voir chapitre 4, section 4-5) :

Algorithme de classification des quadrants :

1. Marquer tous les quadrants partiels.
2. Pour tous les quadrants qui ne sont pas marqués, faire :
    a. Liste_quadrants_inconnus = NUL;

b. test_int = 0;

c. test_ext = 0;

d. test_limit = 0;

e. Tester Quadrant

f. Si test_int == 1 et test_ext == 0 marquer tous les quadrants dans la liste "Liste_quadrants_inconnus" à quadrant extérieur;

g. Si test_int == 1 et test_ext == 1 marquer tous les quadrants dans la liste "Liste_quadrants_inconnus" à quadrant intérieur;

h. Si test_ext == 1 et test_limit == 1 marquer tous les quadrants dans la liste "Liste_quadrants_inconnus" à quadrant extérieur;

i. Autrement arrêter avec un message erreur.

Tester Quadrant :

1. Marquer le quadrant inconnu et ajouter dans la liste "Liste_quadrants_inconnus".
2. Si le quadrant a un quadrant voisin partiel de contour intérieur test_int = 1;
3. Si le quadrant a un quadrant voisin partiel de contour extérieur test_ext = 1;
4. Si le quadrant a un voisin de type limite test_limit = 1;
5. Pour tous les quadrants voisins de ce quadrant qui ne sont pas marqués faire:
    a. Tester Quadrant

On peut présenter l'algorithme parallèle sous la forme suivante :

Algorithme parallèle de classification des quadrants :

{on utilise le modèle SIMD SM avec accès concurrent en lecture et exclusif en écriture.}

i,j : entier

No_p : variable de mémoire partagée de type sémaphore initialisée à N (nombre de processeur) qui désigne le nombre de processeurs disponibles à un

instant donné.

1. {Marquer tous les quadrants partiels.}
    Pour i = 1 jusqu'à (nombre de quadrants dans la liste) / N faire
        For j = 1 to N do in parallel
            Si (i-1)*N+j ≤ (nombre de quadrants dans la liste)
                Vérifier si le quadrant (i-1)*N+j est partiel ou non, s'il est partiel marquer ce quadrant comme un quadrant partiel.
            Fin si

        End for
    Fin faire
2. Pour tous les quadrants qui ne sont pas marqués faire :
    a. Liste_quadrants_inconnus = NUL;
    b. test_int = 0;
    c. test_ext = 0;
    d. test_limit = 0;
    e. Tester Quadrant parallèle
    f. Si test_int == 1 et test_ext == 0 marquer tous les quadrants dans la liste "Liste_quadrants_inconnus" à quadrant extérieur;
    g. Si test_int == 1 et test_ext == 1 marquer tous les quadrants dans la liste "Liste_quadrants_inconnus" à quadrant intérieur;
    h. Si test_ext == 1 et test_limit == 1 marquer tous les quadrants dans la liste "Liste_quadrants_inconnus" à quadrant extérieur;
    i. Autrement arrêter avec un message erreur.

Tester Quadrant parallèle :

1. Marquer le quadrant inconnu et ajouter dans la liste "Liste_quadrants_inconnus".
2. Si le quadrant a un quadrant voisin partiel de contour intérieur test_int = 1;

3. Si le quadrant a un quadrant voisin partiel de contour extérieur test_ext = 1;
4. Si le quadrant a un voisin de type limite test_limit = 1;
5. S'il existe les processeurs libres (No_p ≥ nombre des quadrants voisins) prendre "nombre de voisins" processeurs libres (No_p modifié à No_p - "nombre des voisins")

    For "tous les quadrants voisins de ce quadrant qui ne sont pas marqués do in parallèle :

        Tester Quadrant parallèle

    End for

Sinon Pour tous les quadrants voisins de ce quadrant qui ne sont pas marqués

    Tester Quadrant

Fin si

Analyse des algorithmes :

Si on considère le temps d'exécution des opérations qui sont effectuées sur un quadrant est de l'ordre de O(1). Comme doit tester chaque quadrant une fois, donc l'algorithme séquentiel est de l'ordre O(n), n est le nombre de quadrants. Pour l'algorithme parallèle, dans un problème ordinaire à 3 ou 4 zones (une zone extérieure et deux ou trois zones intérieures), l'algorithme doit être exécuté zone après zone (séquentiellement). On a utilisé une structure d'arbre, donc les branches sont égales au nombre de voisins d'un quadrant. Donc si on avait assez de processeurs, le temps d'exécution serait dans l'ordre de O(hauteur de l'arbre = $\log_{\text{nombre\_de\_voisins}} n$).

6-4-5 Génération des quadrants frontières :

On peut présenter cette étape sous la forme de l'algorithme suivant (voir chapitre 4, section 4-6) :

Algorithme de génération des quadrants frontières :

1. Pour tous les quadrants intérieurs à côté des quadrants partiels faire :

    (Voir figure (4-10) du chapitre 4, la section 4-6)

    a. Pour tous les segments libres (l) du quadrant faire :

        i. Pour les deux points d'extrémité du segment (Q) faire :

            (1) Trouver le point (P) sur les courbes frontières de distance minimum avec le point Q.

            (2) Vérifier si le segment QP ne coupe pas les quadrants existants.

            (3) Si segment QP coupe un quadrant et il existe un autre point de projection, essayer un autre. Aller à l'étape (2).

            (4) S'il n'existe pas un autre point de projection, prendre le point le plus proche, se déplacer pas à pas sur la courbe frontière pour trouver le point (P) qui ne coupe pas les autres quadrants.

        ii. Générer le nouveau quadrant à partir des segments l, (Q_1 P_1), (P_1, P_2) et (Q_2, P_2).

2. Ajuster les coins.

On peut présenter l'algorithme parallèle de cette étape sous la forme suivante:

Algorithme parallèle de génération des quadrants frontières :

{on utilise le modèle SIMD SM avec accès concurrent en lecture et exclusif en écriture.}

i,i : entier

1. Pour i = 1 jusqu'à (nombre de quadrants intérieurs à côté des quadrants partiels) / N faire :

    For j = 1 to N do in parallel

Si (i-1)*N+j ≤ (nombre de quadrants intérieurs à côté des quadrants partiels)

a. Pour tous les segments libres (l) du quadrant faire :

i. Pour les deux points d'extrémité du segment (Q) faire :

(1) Trouver le point (P) sur les courbes frontières de distance minimum avec le point Q.

(2) Vérifier si le segment QP ne coupe pas les quadrants existants.

(3) Si segment QP coupe un quadrant et il existe un autre point de projection, essayer un autre. Aller à l'étape (2).

(4) S'il n'existe pas un autre point de projection, prendre le point le plus proche, redéplacer pas à pas sur la courbe frontière pour trouver le point (P) qui ne coupe pas les autres quadrants.

Fin faire

ii. Générer le nouveau quadrant à partir des segments l, (Q_1 P_1), (P_1, P_2) et (Q_2, P_2).

Fin si

End for

Fin faire

2. Pour i = 1 jusqu'à (nombre de coins) / N faire :

For j = 1 to N do in parallel

Si (i-1)*N+j ≤ (nombre de coins)

Ajuster les coins.

Fin si

End for

Fin faire

Analyse :

Si le temps d'exécution de l'algorithme séquentiel est présenté par O(t(n)). Le temps d'exécution de l'algorithme parallèle doit être dans l'ordre O(t(n)/N).

6-4-6 Lissage intérieur :

On peut présenter cette étape sous la forme de l'algorithme suivant (voir le chapitre 4, la section 4-7) :

Algorithme : lissage intérieur

1. Pour i = 1 à 4 faire :
    a. pour tous les quadrants intérieurs faire :
        i. Pour tous les points (A) de quadrant qui ne sont pas encore modifiés dans le cycle i faire :
            (1) Trouver les points directement reliés à A.
            (2) Trouver les points reliés, mais pas directement reliés à A.
            (3) Utiliser l'équation (4.3) et modifier les coordonnées du point.

On peut présenter l'algorithme parallèle sous la forme de l'algorithme suivante:

Algorithme parallèle de lissage intérieur :

{on utilise le modèle SIMD SM avec accès concurrent en lecture et exclusif en écriture.}

i, j, k : entier

1. Pour k = 1 à 4 faire :

a. Pour i = 1 jusqu'à (nombre de quadrants intérieurs) / N faire :

For j = 1 to N do in parallel

Si (i-1)*N+j ≤ (nombre de quadrants intérieurs)

i. Pour tous les points (A) du quadrant qui ne sont pas encore modifiés dans le cycle i faire :

(1) Trouver les points directement reliés à A.

(2) Trouver les points reliés, mais pas directement reliés à A.

(3) Utiliser l'équation (4.3) et modifier les coordonnées du point.

Fin faire

Fin si

End for

Fin faire

Fin faire

Analyse des algorithmes :

Si on considère le temps d'exécution des opérations qui sont effectuées sur un quadrant est de l'ordre de O(1). Le temps d'exécution de l'algorithme séquentiel est de l'ordre O(n) (n : nombre de quadrants intérieurs) et le temps d'exécution de l'algorithme parallèle est dans l'ordre O(n/N).

6-4-7 Génération du maillage en éléments finis :

On peut présenter l'algorithme général de cette étape sous la forme suivante (voir chapitre 4, section 4-8) :

Algorithme de génération de maillage à partir des quadrants générés

1. Pour Tous les quadrants faire :
    a. Si le quadrant est dégénéré et correspond à un triangle
        i. Le triangle est un élément fini.
    Sinon
        i. Trouver la séquence de quadrant
        ii. Sélectionner le patron de maillage.
        iii. Projeter le patron sur le quadrant et trouver les éléments finis.

On peut présenter l'algorithme parallèle de cette étape sous la forme suivant :

Algorithme parallèle de génération de maillage à partir des quadrants générés :

{on utilise le modèle SIMD SM avec accès concurrent en lecture et exclusif en écriture.}

1. Pour i = 1 jusqu'à (nombre de quadrants) / N faire :
    For j = 1 to N do in parallel
        Si (i-1)*N+j $\leq$ (nombre de quadrants)
            a. Si le quadrant est dégénéré et correspond à un triangle
                i. Le triangle est un élément fini.
            Sinon
                i. Trouver la séquence de quadrant
                ii. Sélectionner le patron de maillage.
                iii. Projeter le patron sur le quadrant et trouver les éléments.
            Fin si
        Fin si
    End for
Fin faire

Analyse des algorithmes :

Si on considère le temps d'exécution des opérations qui sont effectuées sur un quadrant est de l'ordre de O(1). Le temps d'exécution de l'algorithme séquentiel est de l'ordre O(n) (n : nombre de quadrants) et le temps d'exécution de l'algorithme parallèle est dans l'ordre O(n/N).

6-4-8 Algorithme général :

L'algorithme général du programme peut se présenter comme suit (voir le chapitre 4, section 4-10-1) :

1. Entrée des données géométriques et les paramètres de maillage.
2. Génération du modèle géométrique unifié via le modèle NURBS.
   (Si on a besoin d'éléments finis de frontière de type CAO "geometric boundary elements", on va à l'étape 5)
3. Génération des quadrants :
   a. Subdivision récursive et construction de l'arbre quadtree.
   b. Imposition de la différence d'un niveau de subdivision.
   c. Classification et identification des quadrants intérieurs.
   d. Génération des quadrants frontières
4. Application du lissage intérieur.
5. Génération du maillage en éléments finis.
   (Classique ou éléments finis de frontière)
6. Sortie et construction des fichiers résultats.

Ces étapes doivent être exécutées séquentiellement donc on ne peut pas paralléliser l'algorithme général du programme. Mais on peut paralléliser chacune des étapes présentées ci-dessus.

6-5 Parallélisation des algorithmes de génération de maillage en 3D :

Dans cette section nous essayons de présenter les algorithmes parallélisables de génération de maillage en trois dimensions.

Dans cette section, normalement on utilise le modèle SIMD SM avec accès concurrent en lecture et exclusif en écriture. On représente le nombre de processeurs par “N”.

6-5-1 Modélisation géométrique de l'objet :

L'algorithme qui suit présente les étapes à suivre pour obtenir une représentation par surface B-spline composite bicubique lorsqu’on dispose de points (sous forme de tableau) sur cette surface (voir le chapitre 5, la section 5-2) :

1- Trouver les espacements nodaux $\{\nabla_i$ et $\nabla_j\}$:

a. *Intervalles de support* : Une estimation raisonnable de la taille des espacements nodaux est de rendre ces espacements égaux à la longueur des cordes entre les points donnés. Dans ce cas, les espacements nodaux de support dans les directions u et w seront définis respectivement par :

$$\nabla_i = \sum_{j=0}^{n} |P_{i+1,j} - P_{i,j}| \qquad pour\ i = 0,1,...,m-1$$

$$\nabla_j = \sum_{i=0}^{m} |P_{i,j+1} - P_{i,j}| \qquad pour\ j = 0,1,...,n-1 \tag{5.3}$$

b. *Intervalles auxiliaires* : Le vecteur des espacements nodaux étendu est

complété en fixant les extrémités du vecteur à zéro i.e. :

$$\nabla_{-2} = \nabla_{-1} = 0 = \nabla_m = \nabla_{m+1} = \nabla_n = \nabla_{n+1} \qquad (5.4)$$

c. *Intervalles entre les points triples* : Les intervalles entre les lignes ou les colonnes triples sont nuls mais pour éviter la division par zéro dans les calculs, on choisit une valeur petite mais assez grande.

2- Déterminer les points de contrôle intermédiaires C_{ij} :

Pour obtenir les vecteurs des points de contrôle intermédiaires, chaque colonne j des données d'entrée est interpolée par une courbe B-spline composite non-uniforme. C'est-à-dire que pour chaque colonne j, il faut interpoler une courbe à partir des points donnés. À cette fin, il suffit de résoudre le système d'équations linéaires (3.59), pour (j = 0, 1, ..., n-1).

3- Déterminer les vecteurs de frontière d_i et e_j :

Les vecteurs de frontière seront calculés de la même manière que les points de contrôle intermédiaires le sont, mais en modifiant l'équation (3.59), et en remplaçant les points P par les vecteurs tangents T et les vecteurs tangents S par les vecteurs de torsion X (les vecteurs de frontière d_i et e_i remplacent à ce moment la matrice de C_{ij}). De cette façon, la première rangée et la dernière rangée de la matrice C contiendront les éléments d_i et e_i respectivement.

4- Déterminer les points de contrôle B-spline V_{ij} :

Pour ce faire, il suffit d'interpoler une courbe B-spline pour chaque rangée i de la matrice C, c'est-à-dire de la matrice des points de contrôle intermédiaires. La méthode utilisée ici est la même (résolution d'un système d'équations linéaires), mais cette fois, l'opération est effectuée dans le sens orthogonale (dans la direction des rangées, et non des colonnes comme

précédemment). Pour chaque rangée i (i = 0, ..., m+2), il suffit de résoudre le système d'équations linéaires (3.64).

On peut écrire l'algorithme parallèle pour arriver à la représentation par modèle bicubique B-spline composite comme suit :

Algorithme parallèle de modélisation géométrique par surface B-spline composite bicubique :

{on utilise le modèle SIMD SM avec accès concurrent en lecture et exclusif en écriture.}

N : le nombre de processeurs.

k, i, j, l : entier

1- Trouver les espacements nodaux {∇_i et ∇_j} :

a. *Intervalles de support (direction u) :*

Pour k = 1 jusqu'à (m/N) faire {m :le nombre de lignes des points donnés}

For l = 1 to N do in parallel

i = (k-1)*N+l-1

Si i < m

$$\nabla_i = \sum_{j=0}^{n} |P_{i+1,j} - P_{i,j}| \qquad \text{(direction u)}$$

Si $\nabla_i = 0$ (*Intervalles entre les points triples*) on choisit une valeur petite mais assez grande pour éviter la division par zéro.

Fin si

Fin si

End for

Fin faire

b. *Intervalles de support (direction w)* :

Pour k = 1 jusqu'à (n/N) faire{n :le nombre de colonnes de points donnés}

For l = 1 to N do in parallel

j = (k-1)*N+l-1

Si j < n

$$\nabla_j = \sum_{i=0}^{m} |P_{i,j+1} - P_{i,j}| \qquad \text{(direction w)}$$

Si $\nabla_j = 0$ (*Intervalles entre les points triples*) on choisit une valeur petite mais assez grande pour éviter la division par zéro.

Fin si

Fin si

End for

Fin faire

c. *Intervalles auxiliaires* : Le vecteur des espacements nodaux étendu est complété en fixant les extrémités du vecteur à zéro i.e. :

$\nabla_{-2} = \nabla_{-1} = 0 = \nabla_m = \nabla_{m+1} = \nabla_n = \nabla_{n+1}$

2- Déterminer les points de contrôle intermédiaires C_{ij} :

Chaque colonne j des données d'entrée est interpolée par une courbe B-spline composite non-uniforme. Il suffit d'utiliser un algorithme parallèle semblable à l'algorithme qui est présenté dans la section 6-4-1 et résoudre le

système d'équations linéaires (3.59), pour (j = 0, 1, ..., n-1).

3- Déterminer les vecteurs de frontière d_i et e_j :

Les vecteurs de frontière seront calculés de la même manière que les points de contrôle intermédiaires (étape 2), mais en modifiant l'équation (3.59), et en remplaçant les points P par les vecteurs tangents T et les vecteurs tangents S par les vecteurs de torsion X (les vecteurs de frontière d_i et e_j remplacent à ce moment la matrice de C_{ij}). De cette façon, la première rangée et la dernière rangée de la matrice C contiendront les éléments d_i et e_i respectivement.

4- Déterminer les points de contrôle B-spline V_{ij} :

Il suffit d'utiliser un algorithme parallèle semblable à l'algorithme qui est présenté dans la section 6-4-1 et interpoler une courbe B-spline pour chaque rangée i de la matrice C, c'est-à-dire de la matrice des points de contrôle intermédiaires. La méthode utilisée ici est la même (résolution d'un système d'équations linéaires), mais cette fois, l'opération est effectuée dans le sens orthogonale (dans la direction des rangées, et non des colonnes comme précédemment). Pour chaque rangée i (i = 0, ..., m+2), il suffit de résoudre le système d'équations linéaires (3.64).

Analyse de l'algorithme parallèle :

- L'étape 1 est dans l'ordre de O(n/N) (n = max(nb_lignes,nb_colonnes).
- L'étape 2 est dans l'ordre de $n \times O(n^x / N)$ avec $(2 < x < 2.5)$ ou dans l'ordre de $O(n^x / N)$ avec $(3 < x < 3.5)$.
- L'étape 3 est dans l'ordre de $O(n^x / N)$ avec $(2 < x < 2.5)$.
- L'étape 4 est dans l'ordre de $n \times O(n^x / N)$ avec $(2 < x < 2.5)$ ou dans l'ordre de $O(n^x / N)$ avec $(3 < x < 3.5)$.

D'où on déduit que l'ordre d'exécution de l'algorithme est:

$t(n) = O(n/N) + O(n^x / N)\ (2 < x < 2.5) + O(n^x / N)\ (3 < x < 3.5)$

$t(n) \in O(n^x / N)$ avec $(3 < x < 3.5)$.

6-5-2 Création des octants, de l'arbre octree et de la différence d'un niveau de subdivision :

On peut présenter la procédure de génération de l'arbre octree qui permet de tester et subdiviser un octant pour s'assurer de la condition de la différence d'un niveau de subdivision sous forme d'un algorithme comme suit (voir le chapitre 5, la section 5-4) :

Algorithme : Tester_subdiviser_octant (octant, niveau)

```
Début
    Si Max_paramètre_de_contrôle > niveau  alors
        Diviser_octant_en_huit;
        Compléter_voisinage;
        Pour tous les octants voisins faire
            Si vérifier_un_niveau_diférence(octant_voisin) != ok alors
                Tester_subdiviser_octant (octant_voisin,
                niveau_octant_voisin);
            Finsi;
        Finfaire;
        Pour i = 1 to 8 faire
            Tester_subdiviser_octant (octant(i), niveau+1);
        Finfaire;
```

Finsi;

Fin;

On peut présenter l'algorithme parallèle de procédure de génération de l'arbre octree qui permet de tester et subdiviser un octant pour s'assurer de la condition de la différence d'un niveau de subdivision sous forme d'un algorithme comme suit :

Algorithme : Tester_subdiviser_octant_parallèle (octant, niveau)

{on utilise le modèle SIMD SM avec accès concurrent en lecture et exclusif en écriture.}

{les informations sur les voisins doivent être stockées dans la zone de mémoire partagée en lecture concurrent}

No_p : variable de mémoire partagée de type sémaphore initialisée à N (nombre de processeur) qui désigne le nombre de processeurs disponibles à un instant donné.

Début

Si Max_paramètre_de_contrôle > niveau alors

Diviser_octant_en_huit;

Compléter_voisinage;

S'il existe des processeurs libres (No_p ≥ nombre des octants voisins)

prendre "nombre de voisins" processeurs libres (No_p modifié à No_p - "nombre des voisins")

For "tous les octants voisins" do in parallel

Si vérifier_un_niveau_diférence(octant_voisin) != ok alors

Tester_subdiviser_octant_parallèle (octant_voisin, niveau_octant_voisin);

```
                    Finsi;
                End do
            Sinon pour tous les octants voisins faire
                Si vérifier_un_niveau_diférence(octant_voisin) != ok alors
                    Tester_subdiviser_octant_parallèle (octant_voisin,
                    niveau_octant_voisin);
                Finsi;
            Finfaire;
            S'il existe des processeurs libres (No_p ≥ 8)
                prendre 8 processeurs libres (No_p modifié à No_p - 8)
                 For i = 1 to 8 do in parallel
                    Tester_subdiviser_octant_parallèle (octant(i), niveau+1);
                End for;
            Sinon pour i = 1 à 8 faire
                    Tester_subdiviser_octant (octant(i), niveau+1);
                Finfaire;
            Fin si;
        Fin si;
Fin;
```

Analyse des algorithmes :

Si on considère le temps d'exécution des opérations qui sont effectuées sur un octant est de l'ordre de O(1). Le temps d'exécution de l'algorithme séquentiel est de l'ordre du nombre d'éléments dans l'arbre soit $O(8^{niveaux})$. Pour un algorithme parallèle on doit arriver aux feuilles afin de terminer l'exécution de l'algorithme. Donc si on avait assez de processeurs, cet algorithme serait de l'ordre du nombre de niveaux O(niveaux) parce que les éléments de chaque niveau de l'arbre exécutent en parallèle. Sinon cet algorithme est dans l'ordre $O(8^{niveaux} / N)$ (N est le nombre de processeurs).

6-5-3 Classification des octants :

On peut présenter cette étape sous la forme d'un algorithme comme suit (voir le chapitre 5, la section 5-5) :

Algorithme de classification des octants :

1. Marquer tous les octants partiels.
2. Pour tous les octants qui ne sont pas marqués faire :
    a. Liste_octants_inconnus = NUL;
    b. test_int = 0;
    c. test_ext = 0;
    d. test_limit = 0;
    e. Tester Octant ;
    f. Si test_int == 1 et test_ext == 0 marquer tous les octants dans la liste "Liste_octants_inconnus" à octant extérieur;
    g. Si test_int == 1 et test_ext == 1 marquer tous les octants dans la liste "Liste_octants_inconnus" à octant intérieur;
    h. Si test_ext == 1 et test_limit == 1 marquer tous les octants dans la liste "Liste_octants_inconnus" à octant extérieur;
    i. Autrement arrêter avec un message erreur.

Tester Octant :

1. Marquer l’octant inconnu et ajouter dans la liste "Liste_octants_inconnus";
2. Si l’octant a un octant voisin partiel de la surface intérieure test_int = 1;
3. Si l’octant a un octant voisin partiel de la surface extérieure test_ext = 1;
4. Si l’octant a un voisin de type limite test_limit = 1;

5. Pour tous les octants voisins de cet octant qui ne sont pas marqués faire:
    a. Tester Octant;

On peut présenter l'algorithme parallèle correspondant sous la forme suivante:

Algorithme parallèle de classification des octants :

{on utilise le modèle SIMD SM avec accès concurrent en lecture et exclusif en écriture.}

i,j : entier

No_p : variable de mémoire partagée de type sémaphore initialisée à N (nombre de processeur) qui désigne le nombre de processeurs disponibles à un instant donné.

1. {Marquer tous les octants partiels.}
    Pour i = 1 jusqu'à (nombre d'octants dans la liste) / N faire
        For j = 1 to N do in parallel
            Si (i-1)*N+j ≤ (nombre d'octants dans la liste)
                Vérifier si l'octant (i-1)*N+j est partiel ou non, s'il est partiel marquer cet octant comme un octant partiel.
            Fin si
        End for
    Fin faire
2. Pour tous les octants qui ne sont pas marqués faire :
    a. Liste_octants_inconnus = NUL;
    b. test_int = 0;
    c. test_ext = 0;
    d. test_limit = 0;
    e. Tester Octant parallèle ;

f. Si test_int == 1 et test_ext == 0 marquer tous les octants dans la liste "Liste_octants_inconnus" à octant extérieur;

g. Si test_int == 1 et test_ext == 1 marquer tous les octants dans la liste "Liste_octants_inconnus" à octant intérieur;

h. Si test_ext == 1 et test_limit == 1 marquer tous les octants dans la liste "Liste_octants_inconnus" à octant extérieur;

i. Autrement arrêter avec un message erreur.

Tester Octant parallèle :

1. Marquer l'octant inconnu et ajouter dans la liste "Liste_octants_inconnus";
2. Si l'octant a un octant voisin partiel de la surface intérieure test_int = 1;
3. Si l'octant a un octant voisin partiel de la surface extérieure test_ext = 1;
4. Si l'octant a un voisin de type limite test_limit = 1;
5. S'il existe des processeurs libres (No_p ≥ nombre des octants voisins)
   prendre "nombre de voisins" processeurs libres (No_p modifié à No_p - "nombre des voisins")
   For tous les octants voisins de cet octant qui ne sont pas marqués
   do in parallel :
   Tester Octant parallel
   End for
   Sinon Pour tous les octants voisins de cet octant qui ne sont pas marqués
   Tester Octant
   Fin si

Analyse des algorithmes :

Si on considère le temps d'exécution des opérations qui sont effectuées sur un octant est de l'ordre de O(1). Comme doit tester chaque octant une fois, donc l'algorithme séquentiel est de l'ordre O(n), n est le nombre d'octants. Pour l'algorithme

parallèle, dans un problème ordinaire à 3 ou 4 zones (une zone extérieure et deux ou trois zones intérieures), l'algorithme doit être exécuté zone après zone (séquentiellement). On a utilisé une structure d'arbre, donc les branches sont égales au nombre de voisins d'un octant. Donc si on avait assez de processeurs, le temps d'exécution serait dans l'ordre de O(hauteur de l'arbre = $\log_{nombre\_de\_voisins}$ n).

6-5-4 Génération des octants frontières :

On peut présenter cette étape sous la forme d'un algorithme comme suit (voir le chapitre 5, la section 5-6) :

Algorithme séquentiel de génération des octants frontières :

1. Pour tous les octants intérieurs voisins des octants partiels faire :
    a. Pour toutes les faces libres (f) d'octant faire :
        i. Si la face est entièrement libre
            Pour les quatre sommets (Q) aux coins et les quatre points (Q) au milieu de chaque côté de la face et le point (Q) au centre de la face
        Ou Si la face n'est pas entièrement libre
            Pour les points (Q) libres aux coins et au milieu de chaque côté de la face et le point (Q) au centre de la face
        faire :
            (1) Trouver le point (P) sur les surfaces frontières situé à une distance minimum avec le point Q.
            (2) Vérifier si le segment QP ainsi trouvé ne coupe pas les octants existants.

(3) Si segment QP coupe un octant et qu'il existe un autre point de projection essayer l'autre point. Aller à l'étape (2).

(4) S'il n'existe pas un autre point de projection prendre le point plus proche, et marcher pas à pas sur la surface frontière pour trouver le point (P) qui ne coupe pas les autres octants.

ii. Générer jusqu'à quatre nouveaux octants à partir de la partie libre de la face (f) entre les points Q_i et leurs points de projection $_iP$ sur les surfaces frontière.

2. Ajuster les coins et les bords.

a. Pour toutes les surfaces frontières faire :

i. Pour tous les côtés de la surface sur les bords de l'objet faire :

(1) suivre le côté pas à pas et à chaque pas, trouver le segment d'un octant frontière généré sur les frontières proche à ce côté et déplacer le segment sur le côté.

On peut présente l'algorithme parallèle de cette étape sous la forme d'un algorithme comme suit :

Algorithme parallèle de génération des octants frontières :

{on utilise le modèle SIMD SM avec accès concurrent en lecture et exclusif en écriture.}

i,j : entier

1. Pour i = 1 jusqu'à (nombre d'octants intérieurs voisins des octants partiels) / N faire :

For j = 1 to N do in parallèle

Si (i-1)*N $\leq$ (nombre d'octants intérieurs voisins des octants

partiels)

a. Pour toutes les faces libres (f) d'octant faire :

i. Si la face est entièrement libre

Pour les quatre sommets (Q) aux coins et les quatre points (Q) au milieu de chaque côté de la face et le point (Q) au centre de la face

Ou Si la face n'est pas entièrement libre

Pour les points (Q) libres aux coins et au milieu de chaque côté de la face et le point (Q) au centre de la face

faire :

(1) Trouver le point (P) sur les surfaces frontières situé à une distance minimum avec le point Q.

(2) Vérifier si le segment QP ainsi trouvé ne coupe pas les octants existants.

(3) Si segment QP coupe un octant et qu'il existe un autre point de projection essayer l'autre point. Aller à l'étape (2).

(4) S'il n'existe pas un autre point de projection prendre le point plus proche, et marcher pas à pas sur la surface frontière pour trouver le point (P) qui ne coupe pas les autres octants.

Fin faire

ii. Générer jusqu'à quatre nouveaux octants à partir de la partie libre de la face (f) entre les points Q_i et leurs points de projection P_i sur les surfaces frontière.

Fin si

End for

Fin faire

2. Pour i = 1 jusqu'à (nombre de coins et de bords de l'objet) / N faire :

For j = 1 to N do in parallel

Si (i-1)*N+j ≤ (nombre de coins et de bords)

suivre le côté pas à pas et à chaque pas, trouver le segment d'un octant frontière généré sur les frontières proche à ce côté et déplacer le segment sur le côté.

Fin si

End for

Fin faire

Analyse :

Si le temps d'exécution de l'algorithme séquentiel est présenté par O(t(n)). Le temps d'exécution de l'algorithme parallèle doit être dans l'ordre O(t(n)/N).

6-5-5 Lissage intérieur :

On peut présenter cette étape sous la forme d'un algorithme comme suit (voir le chapitre 5, la section 7) :

Algorithme : lissage intérieur

1. Pour i = 1 à 4 faire :

a. pour tous les octants intérieurs faire :

i. Pour tous les points (A) d'un octant qui ne sont pas encore modifiés dans le cycle i faire :

(1) Trouver les points directement reliés à A.

(2) Trouver les points reliés, mais pas directement reliés à A.

(3) Utiliser l'équation (5.5) et modifier les coordonnées du point.

On peut présenter l'algorithme parallèle sous la forme d'un algorithme suivant:

Algorithme : lissage intérieur parallèle

i, j, k : entier

1. Pour k = 1 à 4 faire :
    a. Pour i = 1 jusqu'à (nombre d'octants intérieurs) / N faire :
        For j = 1 to N do in parallel
            Si (i-1)*N+j $\leq$ (nombre d'octants intérieurs)
                i. Pour tous les points (A) d'octant qui ne sont pas encore modifiés dans le cycle i faire :
                    (1) Trouver les points directement reliés à A.
                    (2) Trouver les points reliés, mais pas directement reliés à A.
                    (3) Utiliser l'équation (5.5) et modifier les coordonnées du point.
                Fin faire
            Fin si
        End for
    Fin faire

Fin faire

Analyse des algorithmes :

Si on considère le temps d'exécution des opérations qui sont effectuées sur un octant est de l'ordre de O(1). Le temps d'exécution de l'algorithme séquentiel est de l'ordre O(n) (n : nombre d'octants intérieurs) et le temps d'exécution de l'algorithme parallèle est dans l'ordre O(n/N).

6-5-6 Génération du maillage pour éléments finis :

On peut présenter l'algorithme général de cette étape sous la forme suivante (voir le chapitre 5, la section 5-8) :

Algorithme de génération de maillage

1. Pour Tous les octants faire :
    a. Si élément fini désiré est de type hexaèdre
        i. Si les voisins de face et d'arête de l'octant ayant un niveau de subdivision inférieur ou égal au niveau de subdivision de l'octant
            (1) déclarer cet octant comme un élément fini.
            (2) calculer le volume de l'élément.
            (3) Si le volume est égal à zéro éliminer l'élément
                Sinon ajouter l'élément dans la liste des éléments finis.
        Sinon
            (1) diviser l'octant en six pyramides
            (2) Pour chaque pyramide
                (a) Sélectionner le patron de maillage.
                (b) Projeter le patron sur l'octant et trouver les éléments.
                (c) Pour chaque élément
                    (i) calculer le volume de l'élément.
                    (ii) Si le volume est égal à zéro éliminer l'élément
                        Sinon ajouter l'élément dans la liste des éléments finis.
    b. Si l'élément fini désiré est de type pyramide
            (1) diviser l'octant en six pyramides
            (2) Pour chaque pyramide

(a) Sélectionner le patron de maillage.

(b) Projeter le patron sur l'octant et trouver les éléments.

(c) Pour chaque élément

(i) calculer le volume de l'élément.

(ii) Si le volume est égal à zéro éliminer l'élément

Sinon ajouter l'élément dans la liste des éléments finis.

c. Si l'élément fini désiré est de type tétraèdre

i. Si les voisins de face et d'arête de l'octant ayant un niveau de subdivision inférieur ou égal au niveau de subdivision de l'octant

(1) diviser l'octant en cinq tétraèdres

(2) Pour chaque tétraèdre

(a) calculer le volume du tétraèdre.

(b) Si le volume est égal à zéro éliminer le tétraèdre

Sinon ajouter le tétraèdre dans la liste des éléments finis.

Sinon

(1) diviser l'octant en six pyramides

(2) Pour chaque pyramide

(a) Sélectionner le patron de maillage.

(b) Projeter le patron sur l'octant et trouver les éléments.

(c) Pour chaque élément

(i) calculer le volume de l'élément.

(ii) Si le volume est égal à zéro éliminer l'élément

Sinon ajouter l'élément dans la liste des éléments finis.

On peut présenter l'algorithme parallèle de cette étape sous la forme suivante:

Algorithme parallèle de génération de maillage à partir les octants générés :

{on utilise le modèle SIMD SM avec accès concurrent en lecture et exclusif en écriture.}

1. Pour i = 1 jusqu'à (nombre d'octants) / N faire :
    For j = 1 to N do in parallel
        Si (i-1)*N+j ≤ (nombre d'octants)
            Comme algorithme séquentiel faire étape a ou b ou c.
        Fin si
    End for

Fin faire

Analyse des algorithmes :

Si on considère le temps d'exécution des opérations qui sont effectuées sur un octant est de l'ordre de O(1). Le temps d'exécution de l'algorithme séquentiel est de l'ordre O(n) (n : nombre d'octants) et le temps d'exécution de l'algorithme parallèle est dans l'ordre O(n/N).

On peut présenter l'algorithme de génération des éléments finis de frontière sous la forme suivante :

Algorithme de génération des éléments finis de frontière

1. Pour Toutes les surfaces faire :
    a. Si la surface est plane
        i. En utilisant la méthode quadtree modifiée subdiviser le plan.
        ii. Chaque élément généré dans l'étape 1-a-i est un élément fini de frontière
    b. Sinon

i. Pour chaque carreau de surface B-spline faire

(1) Trouver Δu et Δw, en utilisant l'équation (5, 6)

(2) u = 0

(3) Tant que u entre 0 et (1-Δu) faire

(a) w = 0

(b) Tant que w entre 0 et (1-Δu) faire

(i) calculer les points sur le carreau de surface correspondant aux (u, w), (u+Δu, w), (u+Δu, w+Δu) et (u, w+Δu)

(ii) Si la surface entre les quatre points n'est pas zéro, construire l'élément fini de frontière entre les quatre points et ajouter l'élément dans la liste des éléments finis de frontière.

Fin faire

Fin faire

Fin faire

Fin faire

On peut présenter l'algorithme parallèle de cette étape sous la forme suivante:

Algorithme parallèle de génération des éléments finis de frontière:

{on utilise le modèle SIMD SM avec accès concurrent en lecture et exclusif en écriture.}

1. Pour i = 1 jusqu'à (nombre de surfaces) / N faire :

For j = 1 to N do in parallel

Si (i-1)*N+j ≤ (nombre de surfaces)

Comme l'algorithme séquentiel faire étape a ou b.

Fin si

End for

Fin faire

Analyse des algorithmes :

Le temps d'exécution de l'algorithme séquentiel est de l'ordre O(n*t(m)) où n est le nombre de surfaces et t(m) est le temps d'exécution de l'algorithme mesh2d (génération de maillage sur une surface). Donc le temps d'exécution de l'algorithme parallèle est de l'ordre O(n*t(m)/N).

6-5-7 Algorithme général :

L'algorithme général du programme MESH3D peut se présenter comme suit (voir le chapitre 5, la section 5-10-1) :

1. Entrée des données géométriques et des paramètres de maillage.
2. Génération du modèle géométrique via le modèle NURB.
   (Si on a besoin d'éléments finis de frontière de type CAO "geometric boundary elements", on va à l'étape 5)
3. Génération des octants :
   a. Subdivision récursive et construction de l'arbre octree.
      i. Imposer la condition sur la différence d'un niveau de subdivision.
   b. Classification et identification des octants intérieurs.
   c. Génération des octants frontières
4. Application du lissage intérieur.
5. Génération du maillage pour éléments finis.
   (Classique ou éléments finis de frontière)
6. Sortie et construction des fichiers des résultats.

Ces étapes doivent être exécutées séquentiellement, donc on ne peut pas paralléliser l'algorithme général du programme. Mais on peut paralléllliser chacune de ces étapes qui est présenté en haut.

6-6 Implantation de l'algorithme de résolution du système d'équations sur la machine VolVox :

L'algorithme choisi pour un essai d'implantation est celui de Gauss-Jordan présenté dans la section 6-3. Cette implantation a nécessité la simulation des machines parallèles SIMD avec mémoire partagée sur les machines MIMD sans mémoire partagée (machines VOLVOX du département d'informatique de l'université Laval). Cette simulation a besoin d'une bonne structure de données pour la matrice de coefficients, afin d'envoyer et recevoir de l'information de la manière la plus facile et la plus rapide possible.

6-6-1 Choix d'une topologie :

Étant donné que le nombre de processeurs est en général inférieur au nombre d'éléments de la matrice, on ne peut pas utiliser la topologie matricielle parce qu'il n'est pas possible d'avoir un processeur par élément. De plus, la simulation d'un modèle avec mémoire partagée sur une machine sans mémoire partagée a engendré des problèmes de communication. Tout ceci réduit l'efficacité de la topologie matricielle. Comme nous sommes physiquement limités à quatre liens par processeur (pour la machine VOLVOX), pour avoir une meilleure communication entre le programme principal (maître) et les sous programmes parallèles (esclaves), le plus court chemin entre un point d'origine et tous les autres points, la structure d'arbre ternaire a été choisie.

6-6-2 Algorithme de Gauss-Jordan implanté :

Dans la première partie de l'algorithme de Gauss-Jordan, il se fait une division de chaque élément de la ligne pivot par l'élément pivot. Dans la deuxième partie de l'algorithme de Gauss-Jordan, il se fait une élimination (c'est-à-dire une mise à zéro) de tous les éléments de la colonne pivot sauf celui de la ligne pivot.

L'algorithme d'implantation est constitué d'un algorithme maître et d'un algorithme esclave. Lors de l'initialisation, l'algorithme maître envoie, le nombre de processeurs N, l'ordre de la matrice n, le nombre de processeurs utilisés dans l'étape2 et le nombre d'itérations dans l'étape2, à chaque processeur esclave.

Dans la première étape, le processeur maître envoie l'élément pivot ainsi qu'une partie de la ligne pivot qui contient nb_op_ch_p_etape1 (nombre d'opération de chaque processeur dans l'étape 1) éléments à chaque processeur esclave. Chaque processeur esclave divise la partie reçue de la ligne pivot par l'élément pivot et l'envoie au processeur maître.

Dans la deuxième étape, le processeur maître, dans chaque itération et pour chaque processeur, trouve la ligne qu'il doit traiter et envoie l'élément de la colonne pivot de cette ligne ainsi qu'une partie de cette ligne qui contient nb_op_r_ch_p_etape2 (nombre d'opération de chaque processeur dans l'étape 2) éléments, et la partie de la ligne pivot dans la même colonne que les éléments envoyés. Le processeur esclave multiplie l'élément de la colonne pivot par chaque élément de la ligne pivot qui lui est envoyé, et fait la soustraction de l'élément envoyé et du résultat de cette multiplication qu'il envoie ensuite au processeur maître.

Algorithme Gauss-Jordan_maître(n, A)

entrée

n : l'ordre de la matrice

A(n, n+1) : la matrice de coefficients

sortie

A(i, n+1) i=1..n : vecteur de résultats

{N : le nombre de processeurs}

n, N : entier

piv : entier

nb_op_ch_p_etape1 : entier {nombre d'opérations de chaque processeur de l'étape1}

nb_op_ch_p_etape2 : entier {première estimation du nombre d'opérations de chaque processeur de l'étape2}

nb_op_r_ch_p_etape2 : entier {nombre réel d'opérations de chaque processeur de l'étape2}

nb_div_lig : entier {nombre de divisions de chaque ligne dans l'étape2}

nb_iterations : entier {nombre d'itérations dans l'étape2}

nb_p_etape1 : entier {nombre de processeurs utilisés dans l'étape1}

nb_p_etape2 : entier {nombre de processeurs utilisés dans l'étape2}

element_col_piv : réel {élément dans la colonne pivot}

element_pivot : réel

i : entier {compteur}

no_i : entier

no_l, no_c : entier {numéro de ligne et numéro de colonne}

Id[nb_p_etape2][2] : entier {indice de ligne et de colonne du premier élément

envoyé à un processeur à l'étape2}

```
Début

nb_op_ch_p_etape1 = (n+1)/N

nb_op_ch_p_etape2 = (n-1)*(n+1)/N

Si nb_op_ch_p_etape2 > n+1
alors
        nb_div_lig = 1
sinon
        nb_div_lig = (n+1)/nb_op_ch_p_etape2
fin si

nb_op_r_ch_p_etape2 = (n+1)/nb_div_lig

si (N > n+1) alors
        nb_p_etape1 = n+1
sinon
        nb_p_etape1 = N
fin si

nb_iterations =   nb_div_lig*(n-1)/N

si nb_iterations < 1 alors
        nb_p_etape2 = nb_div_lig*(n - 1)
sinon
        nb_p_etape2 = N
```

```
fin si

Pour i = 1 jusqu'à N
faire
        envoyer(N) à tache(i)
        envoyer(n) à tache(i)
        envoyer(nb_p_etape2) à tache(i)
        envoyer(nb_iterations) à tache(i)
Fin pour

pour piv = 1 jusqu'à n
faire
        element_pivot = A(piv, piv)

        {première étape}
        pour i = 1 jusqu'à nb_p_etape1
        faire
                envoyer(element_pivot) à tache(i)
                envoyer(A(piv, (i - 1)*nb_op_ch_p_etape1+1) à
                  A(piv, i*nb_op_ch_p_etape1)) à tache(i)
        fin faire

        pour i = 1 jusqu'à nb_p_etape1
        faire
                recevoir(A(piv, (i - 1)*nb_op_ch_p_etape1 + 1) à
                  A(piv, i*nb_op_ch_p_etape1)) à tache(i)
        fin faire
```

```
        {deuxième étape}

no_l = 1
no_c = 1
element_col_piv = A(no_l, piv)
pour no_i = 1 jusqu'à nb_iterations
        pour i = 1 to nb_p_etape2
        faire
                si no_l = piv
                alors
                        no_l = no_l +1
                        element_col_piv = A(no_l, piv)
                fin si

                {distribution de l'élément de la colonne pivot}
                envoyer(element_col_piv) à tache(i)

                {distribution de l'élément de la ligne pivot}
                envoyer(A(piv, no_c) à
                        A(piv, nb_op_r_ch_p_etape2 + no_c - 1)) à la tache(i)

                {distribution de l'élément de la ligne no_l}
                Id[i][1] = no_l
                Id[i][2] = no_c
                envoyer(A(Id[i][1], Id[i][2]) à A(Id[i][1], Id[i][2] +
                        nb_op_r_ch_p_etape2 - 1)) à tache(i)

                no_c = no_c + nb_op_r_ch_p_etape2
                si (no_c > n+1) alors
```

```
                    no_c = 1
                    no_l = no_l +1
                    element_col_piv = A(no_l, piv)
                fin si

            fin faire{boucle i}

            pour i = 1 jusqu'à nb_p_etape2
            faire
                recevoir(A(Id[i][1], Id[i][2]) à
                    A(Id[i][1], Id[i][2]+nb_op_r_ch_p_etape2-1)) à tache(i)
            fin faire
        fin faire {no_i}
fin faire {boucle piv}
Fin.
```

Algorithme-Gauss-Jordan_esclave()

N	: entier {nombre de taches qui travaillent}
Id	: entier {identificateur de tache}
n	: entier {nombre de lignes ou de colonnes de la matrice de coefficients}
nb_elem_dist	: entier {nombre d'éléments distribués}
piv	: entier {indice de la ligne ou de la colonne de l'élément pivot}
nb_p_etape2	: entier {nombre de processeurs utilisés dans l'étape2}
no_i	: entier {numéro d'itération}
nb_iterations	: entier {nombre d'itérations dans l'étape2}
element_pivot	: réel {élément pivot}

```
element_col_piv      : réel {élément dans la colonne pivot}
vecteur(n+1)          : réel {ligne de la matrice}
vecteur_piv(n+1)     : réel {ligne pivot}
i                          : entier {compteur}

Début
Trouver l'identificateur de tâche(Id)

recevoir(N) de la tâche maître
recevoir(n) de la tâche maître
recevoir(nb_p_etape2) de la tâche maître
recevoir(nb_iterations) de la tâche maître

pour piv = 1 jusqu'à n
faire
        si Id  ≤ n {exécute l'étape1}
        alors
                recevoir(element_pivot) de la tâche maître
                recevoir(nb_elem_dist et vecteur) à la tâche maître

                pour i = 1 jusqu'à nb_elem_dist
                faire
                        vecteur(i) = vecteur(i)/element_pivot
                fin pour

                envoyer(vecteur) à la tâche maître
        fin si
```

```
        si Id ≤ nb_p_etape2
                pour no_i = 1 jusqu'à nb_iterations
                faire
                        recevoir(element_col_piv) de la tache maître
                        recevoir(nb_elem_dist, vecteur_piv) de la tache maître
                        recevoir(nb_elem_dist, vecteur) de la tache maître

                        pour i = 1 jusqu'à nb_elem_dist
                        faire
                                vecteur(i) = vecteur(i) - elem_col_piv*vecteur_piv(i)
                        fin pour

                        envoyer(vecteur) à la tâche maître

                fin pour {no_i}
        fin si

fin pour {piv}
Fin.
```

6-6-3 Résultats d'implantation :

La méthode de Gauss-Jordan a été implantée sous formes:

- un programme séquentiel et
- un programme parallèle.

Les résultats de ces programmes sont présentés à la suite

Tableau 6-1 : Temps d'exécution du programme séquentiel

n	temps(seconde)
10	.00066
20	.0522
40	.0419
60	1.410
80	3.388
100	6.610
120	11.412
140	18.384
160	27.440

Les temps d'exécution du programme parallèle sont présentés au tableau suivant :

Tableau 6-2 : Temps d'exécution pour 2 transputers (processeurs)

n	Var: double	Var: float
10	.246	.232
20	1.365	1.009
40	7.352	5.584
60	20.777	13.734
80	44.438	30.193
100	86.970	56.233
120	142.075	85.598
140	216.341	134.227
160	326.045	183.501

La deuxième colonne présente le temps d'exécution en utilisant les variables de type double précision pour les données et en troisième colonne, on trouve le temps d'exécution de la même implantation utilisant les variables de type 'float' pour les données. Les tableaux présentés diffèrent selon le nombre de processeurs utilisés.

Tableau 6-3 : Temps d'exécution pour un nombre de transputers = 3

n	Var: double	Var: float
10	.271	.261
20	1.3655	1.014
40	7.036	5.326
60	19.480	12.405
80	41.088	27.787
100	81.850	52.124
120	132.873	77.306
140	201.253	122.234
160	288.856	163.478

Tableau 6-4 : Temps d'exécution pour un nombre de transputers = 4

n	Var: double	Var: float
10	.268	.260
20	1.487	1.11
40	7.017	5.345
60	19.968	12.749
80	44.516	28.653
100	82.598	52.679
120	135.436	78.997
140	206.471	125.362
160	295.573	165.828

Tableau 6-5 : Temps d'exécution pour un nombre de transputers = 13

n	Var: double	Var: float
10	0.568	0.552
20	2.851	2.356
40	13.033	10.162
60	29.494	19.605

80	63.446	43.672
100	126.867	82.234
120	190.022	133.151
140	294.959	181.543
160	431.945	248.64

Tableau 6-6 : Temps d'exécution pour un nombre de transputers = 40

n	Var: double	Var: float
10	1.863	1.807
20	6.255	6.021
40	20.953	18.092
60	60.113	45.564
80	129.649	96.998
100	193.652	137.361
120	335.841	219.32
140	428.556	485.570

Tableau 6-7 : Temps d'exécution pour une matrice d'ordre 100 et un nombre variable de processeurs :

N	Var: double	Var: float
2	86.974	56.233
3	81.851	52.124
4	82.598	54.680
8	114.297	73.066
13	126.867	82.234
26	143.481	98.436
40	193.652	173.361

En comparant les temps d'exécution d'implantations différentes, on voit bien que le temps d'exécution du programme séquentiel est inférieur à celui des programmes parallèles. Cette situation est due à la lenteur du temps de communication entre les processeurs par rapport au temps d'exécution des quelques opérations mathématiques élémentaires de chaque processeur (remarquer que cette répartition d'opérations entre les processeurs nous est presque imposée par la structure de l'algorithme Gauss-Jordan). Dans la deuxième étape de l'algorithme Gauss-Jordan, pour chaque élément, on doit faire seulement une multiplication et une soustraction, ce qui est inférieur au temps d'envoi et de réception d'un élément.

En comparant les temps donnés par un tableau avec seulement deux transputers et sachant qu'il existe un processeur maître qui envoie l'information à un seul processeur esclave qui se charge d'exécuter tous les calculs, les différences de temps présentes entre ce tableau et celui des temps d'exécution du programme séquentiel sont les temps de communication entre les deux processeurs. Ces temps de communication sont supérieurs aux temps d'exécution du programme séquentiel.

Tableau 6-8 : Différences de temps d'exécution d'un programme parallèle utilisant deux transputers et un programme séquentiel ($T_{parallèle} - T_{séquentiel}$)

n	Var: double	Var: float
10	.245	.231
20	1.314	0.957
40	6.933	5.165
60	19.367	12.324
80	41.051	26.807
100	80.363	49.624
120	130.665	74.187
140	197.957	115.843
160	298.605	156.061

La deuxième colonne présente la différence du temps d'exécution en utilisant les variables de type double précision pour les données et en troisième colonne, on trouve la différence du temps d'exécution de la même implantation utilisant les variables de type 'float' pour les données.

En comparant les tableaux dont le nombre de transputers est différent, on constate qu'après un seuil, plus le nombre de transputers augmente, plus le temps d'exécution augmente et ceci est dû au temps de communication élevé.

La comparaison des colonnes deux et trois des tableaux 6.2 à 6.8 montre que pour une même implantation, le temps d'exécution augmente si la précision du résultat augmente. Et parfois même, le temps d'exécution d'une double précision peut aller jusqu'à doubler celui d'une simple précision, ce qui montre l'importance du temps de communication pour envoyer et recevoir les données.

6-7 Conclusions et rétrospectives :

Les processus de génération de maillage en deux et trois dimensions peuvent être laborieux et longs, dans ce chapitre on a présenté les algorithmes parallèles de différentes étapes de génération de maillage afin de diminuer le temps de génération de maillage en éléments finis.

Pour présenter les algorithmes, nous avons utilisé le modèle SIMD SM avec accès concurrent en lecture et exclusif en écriture. Les algorithmes sont aussi utilisables pour les machines MIMD avec un peu de modifications.

Les étapes générales de processus de génération de maillage doivent être exécutées séquentiellement. Mais chaque étape est parallélisable.

Dans ce chapitre, on constate que pour l'implantation d'un meilleur algorithme

séquentiel pour la résolution d'un système d'équations (algorithme séquentiel optimal) sur une machine parallèle, on a besoin d'un algorithme parallèle de multiplication matricielle efficace. Aussi, on sait que dans l'algorithme Gauss-Jordan, on a seulement quelques opérations mathématiques par élément, au plus deux: une multiplication et une soustraction. Donc, pour des machines parallèles adéquates pour l'implantation de cet algorithme, nous devons avoir des temps de communication d'un élément inférieurs au temps d'exécution des opérations pour cet élément, ce qui n'est pas le cas des machines(VOLVOX) sur lesquelles notre implantation a été faite. La vitesse de transmission de données est très lente pour les machines VOLVOX, elle est de 1Mbits/sec ce qui est 10 fois plus lent que le réseau ordinaire Ethernet à 10Mbits/sec. Pour le cas des machines VOLVOX, après un seuil, le temps d'exécution est d'autant plus grand que le nombre de processeurs utilisés augmente à cause d'un besoin plus grand de communication.

Dans le chapitre suivant, on présente quelques exemples de génération de maillage en deux et trois dimensions.