Implémentation d'un produit de matrices tolérant aux fautes

Georges Abou Haydar
georges.abou_haydar@etu.upmc.fr

Mikael Caçote
mikael.cacote@etu.upmc.fr

Encadrants : Jean-Luc Lamotte et Philippe Trébuchet

2 juin 2010

Résumé

Avec des gravures de plus en plus fine, les processeurs engendrent de plus en plus de fautes non détectées qui peuvent complètement invalider des calculs. Face à ce problème, différentes approches sont possibles. Dans le cadre de ce projet, nous nous intéresserons à l'approche ABFT (Algorithm-based Fault Tolerant) pour des algorithmes de calcul numérique et plus particulièrement pour le produit matriciel. La méthode s'appuie sur une extension de la taille des matrices afin de calculer des valeurs au fur et à mesure des calculs qui servent à vérifier l'intégrité des résultats. Un article fondateur de K.-H. Huang, J.A. Abraham [1] explique son principe.

Table des matières

Chapitre 1

Principe

1.1 Introduction

Depuis quelques années, les industriels voient monter en flèche un nouveau type de bug informatique, provoqué par ces particules qui sillonnent l'Univers à de vitesses proches de la lumière et percutent sans cesse la Terre. Il s'agit des rayons cosmiques.
"Quand ils pénètrent dans l'atmosphère, les noyaux d'atomes et les protons, qui forment l'essentiel des rayons cosmiques, se désintégrent en grandes cascades de particules", décrit Jim Ziebler. Ces particules arrivant en contact avec un noyau de silicium peuvent créer une charge éléctrique permettant d'inverser l'état logique d'un bit.
De plus, avec la miniaturisation des circuits, la sensibilité aux rayons cosmiques a connu une forte croissance. Ceci est dû au fait que les courants circulant dans ces circuits deviennent de plus en plus faible ainsi moins d'énergie est nécessaire pour basculer des bits. Ceci nous amène a croire que l'occurrence d'une erreur est plus probable de nos jours et que le nombre d'erreur à un instant t augmente à son tour.
Aucun circuit éléctronique n'est à l'abri de rayons cosmiques. Dans le cadre des ordinateurs on peut alors dire qu'une erreur peut subvenir dans n'importe quelle partie de l'ordinateur : le processeur, la mémoire vive, le bus ...

Pour bien présenter le problème nous devons d'abord définir quelques termes fondamentaux nécessaires à la compréhension du sujet.
La **sûreté de fonctionnement** d'un système permet aux utilisateurs de placer une confiance justifiée dans le service que délivre ce système. Cette sûreté dépend de plusieurs facteurs. Nous pouvons citer parmi ces facteurs la **fiabilité** et la **disponibilité** du système. La fiabilité repose sur le fait que le système est en état de rendre un service conforme à sa spécification alors que la disponibilité n'est autre que la fraction de temps pour laquelle il n'est pas défaillant et du coup disponible pour rendre son service.

Les erreurs amenées aux systèmes par des facteurs comme les rayons cosmiques, peuvent provoquer des défaillances et ainsi dégrader la sûreté des systèmes. Plusieurs techniques ont été développées afin d'essayer de prévenir ou d'éliminer

les fautes. Elles se partagent en deux grandes familles : la **compensation** ou *error masking* et le **recouvrement** ou *error recovery*. Toutes les techniques se basent sur un principe unique : la **redondance** .
Dans le cadre de la compensation nous pouvons citer par exemple le *TMR* ou *Triple Modular Redundancy* qui s'appuie sur la redondance matérielle (en effectuant le même calcul sur 3 processeurs en parallèle) et un système de vote fiable pour masquer une erreur. Bien que le TMR soit une technique assez générale applicable à tous les modules, elle est extrêmement coûteuse au niveau matériel.
Le traitement d'erreurs par recouvrement remplace l'état erroné du système par un état correct. Il se partage lui aussi en deux catégories : le recouvrement par **reprise** ou *backward recovery* et le recouvrement par **poursuite** ou *forward recovery* . Le recouvrement par reprise essaye de ramener le système à un état antérieur correct. Un bon exemple serait le *checkpointing* . Il s'agit de la technique la plus connue et utilisée pour obtenir une tolérance aux pannes dans les systèmes critiques ou les super calculateurs comme *road runner* qui tourne à une moyenne de 32 pannes par jour. Elle consiste à stocker un état cohérent d'un système réparti d'une manière périodique et puis revenir au dernier état correct en cas de panne. Le *checkpointing* est efficace dans quelques cas mais souffre d'un problème de mise à l'échelle. Avec l'augmentation du nombre de processeurs le nombre d'enregistrements devient de plus en plus important alors que ces derniers sont connus pour être le principal coût de cette technique. Ainsi, la disponibilité de système decroit en diminuant la sûreté du système. Or dans des systèmes désirant avoir une grande puissance calculatoire la disponibilité est un facteur très important.

Dans ce projet, nous n'entrerons pas dans les détails des différentes techniques de correction d'erreurs. Notre but est de réussir à implémenter un système tolérant aux fautes assez léger et accessible sur n'importe quel ordinateur récent. K.-H. Huang et J.A. Abraham présentent dans leur article fondateur intitulé : *"Algorithm-Based Fault Tolerance for Matrix Operations"* [1] un algorithme permettant de réaliser un système tolérant aux fautes répondant à nos besoins. Il s'agit d'une technique qui peut être classée dans la dernière famille que nous avons cité : le recouvrement par poursuite basé sur le reconstitution d'un état correct courant.

1.2 Encodage et Matrices

Nous nous intéresserons en particuliers à la détection et la correction d'erreurs dans le calcul matriciel au sein du processeur (et juste le processeur). Le principe de cette technique réside sur les propriétés d'un *checksum*. Celui-ci étant conventionnellement calculé au niveau d'un mot (dans ce cas un entier ou un flottant), nous ne pourrions pas détecter les erreurs affectant tous les bits du mot. Donc nous encoderons nos données à un plus haut niveau.
Dans la représentation choisie par K.-H. Huang et J.A. Abraham, les matrices, cela serait la ligne ou la colonne contenant les différents éléments de la matrice. Nous verrons par la suite que cet encodage permettra de localiser l'élément de la matrice affecté par le défaut et ceci grâce à la redondance dans l'encodage. Cette redondance est due au fait qu'un élément donné d'une matrice intervient dans le calcul du *checksum* au niveau de la ligne et de la colonne dans lesquelles il se trouve. Dans le cadre de ce projet on parlera alors d'une redondance au niveau de l'information.

L'encodage ou plus précisément le *checksum* d'une ligne ou d'une colonne est tout simplement la somme de ses éléments. Les *checksums* calculés successivement sur toutes les lignes ou colonnes forment un vecteur qu'on peut appeler dans ce cas un vecteur de sommation.
En se servant des deux vecteurs de *checksums* calculés pour chaque matrice (un vecteur de sommation pour les lignes et un pour les colonnes), nous pourrions étendre notre matrice d'information en réalisant l'une des trois combinaisons suivantes :

- la matrice d'information avec son propre vecteur de *checksums* de ligne,
- la matrice d'information avec son propre vecteur de *checksums* de colonne
- et finalement la matrice d'information avec ces deux vecteur de *checksums*.

Les auteurs de l'article baptisent les matrices obtenues respectivement *Row Checksum Matrix* (RCM), *Column Checksum Matrix* (CCM) et *Full Checksum Matrix* (FCM) ; en fait une FCM n'est autre qu'une CCM d'une RCM.
Ainsi, nous avons trois nouveaux types de matrices que nous rajoutons au type de base qui est la matrice d'information.

1.3 Extensions Vs. Opérations

Par la suite, K.-H. Huang et J.A. Abraham démontrent que 5 opérations élémentaires sur les matrices préservent la propriété que possèdent leurs checksums. Il s'agit des opérations suivantes :
La multiplication de deux matrices, la multiplication d'une matrice par un scalaire, la décomposition LU, l'addition de deux matrices et la transposition d'une matrice.

On retient de ces démonstrations les résultats suivants :

- Le résultat de la multiplication d'une *Column Checksum Matrix* A_c avec une *Row Checksum Matrix* B_r est une *Full Checksum Matrix* C_f tel que les matrices d'informations correspondantes ne sont pas affectées par l'extension (voir figure 1.1). Ainsi

$$A_c \times B_r = C_f \Leftrightarrow A \times B = C$$

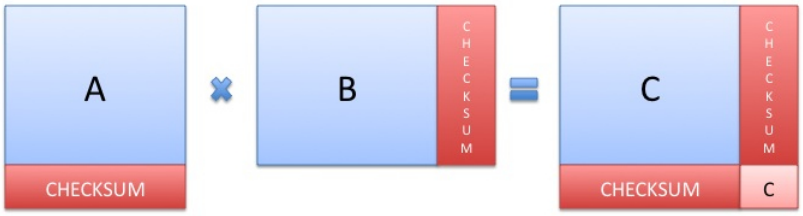

FIGURE 1.1 – Multiplication de deux matrices encodées

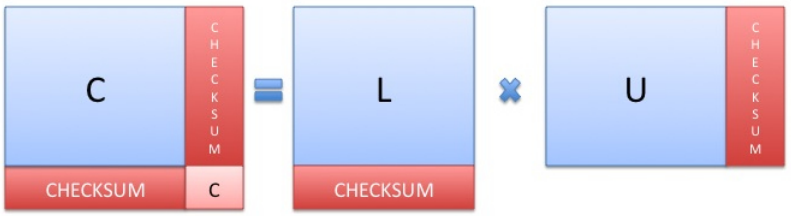

FIGURE 1.2 – Décomposition d'une matrice encodée

– Le même théorème peut s'appliquer pour la multiplication par un scalaire. Ainsi

$$A_f \times x = C_f \Leftrightarrow A \times x = C$$

– En ce qui concerne la décomposition LU, Si une matrice est décomposable en LU sans pivotage (c.-à-d. $C = LU$) la matrice C_f correspondante est elle aussi décomposable en deux matrices L_c et U_r dans lesquelles la matrice d'information n'est autre que la matrice L ou U correspondante dans la décomposition de C (voir figure 1.2). Ainsi

$$L_c \times U_r = C_f \Leftrightarrow L \times U = C$$

Dans ce cas seul une détection d'erreur peut être effectuée sur L_c et U_r afin d'éventuellement relancer le calcul mais les erreurs ne peuvent en aucun cas être corrigées sur des matrices ne contenant qu'un vecteur de sommation. Si la matrice n'est pas décomposable sans pivotement alors elle est seulement décomposable en $C = PLU$ où P est la matrice de pivotage, et on n'obtient plus des matrices L_c et U_r valides sur lesquelles on peut détecter des erreurs.

– Comme pour la multiplication, l'addition garde aussi le bon résultat dans la partie information de la matrice résultat (voir figure 1.3), c.-à-d. que le résultat de l'addition de deux *checksum* matrices (A_f et B_f) est la matrice C_f telle que :

$$A_f + B_f = C_f \Leftrightarrow A + B = C$$

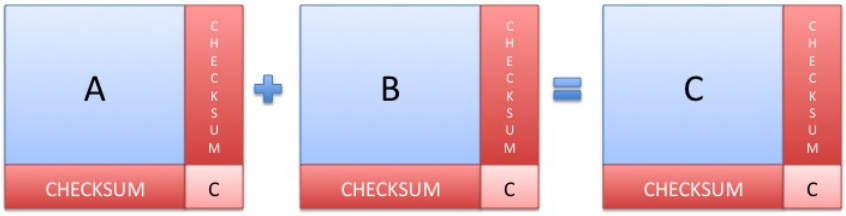

FIGURE 1.3 – Addition de deux matrices encodées

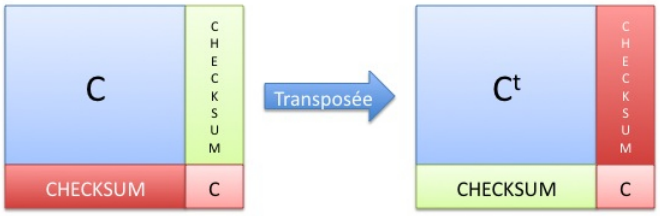

FIGURE 1.4 – Transposée d'une matrice encodée

– Et finalement après la transposition d'une *Full Checksum Matrix* la matrice d'information obtenue est la transposée de la matrice d'information correspondante (voir figure 1.4).

En conclusion, ces théorèmes nous permettent alors d'effectuer ces 5 opérations sans se soucier de l'influence que l'extension de la matrice peut avoir sur la matrice que nous étendons. En d'autres termes, les algorithmes de calculs de ces cinq opérations restent inchangés s'il s'agit d'une *Full Checksum Matrix*, d'une *Row Checksum Matrix*, d'une *Column Checksum Matrix* ou bien d'une matrice. Le passage d'une matrice de taille $m \times n$ à celle d'une taille $(m+1) \times (n+1)$ ne changera aucunement la complexité du calcul de ces cinq opérations. En effet, nous appliquons toujours le même algorithme et la taille de la matrice est du même ordre de la taille de la matrice d'origine.

Pour résumer, ce que nous avons à réaliser dans notre implémentation pour l'instant, serait de pouvoir à partir d'une matrice donnée l'étendre à un des trois autres types de matrices. Ensuite, la propriété du *checksum* étant préservée, on pourra implémenter ces cinq opérations sans se soucier du type de matrice en donnée.
Il reste à comprendre comment ce système de *checksum* permet, en s'appuyant sur la redondance, de détecter l'erreur ensuite d'essayer de la corriger.

1.4 Détection et Correction

La question posée à ce stade serait : "comment pourrait-t-on détecter une erreur en se basant sur cette extension ?" La réponse normalement serait de dire que par définition l'emploi d'un *checksum* a pour but principal de détecter

l'existence d'une erreur dans l'information qu'il encode. D'ailleurs il s'agit de l'étymologie même du mot dans lequel "check" signifie vérification en anglais et "sum" signifie somme ou sommation.

Ainsi, une erreur peut être détectée dans une ligne par exemple en effectuant la somme des éléments de cette ligne et comparant le résultat au *checksum* de cette ligne déjà calculé suite à l'extension de la matrice vers une *Full Checksum Matrix*. Une erreur dans ce cas correspondrait à une inégalité des deux nombres comparés. On détectera ainsi, que dans la ligne concernée, une erreur doit être signalée et effectivement, par la suite être traitée.

Un problème se pose. A moins qu'il ne s'agisse d'une ligne avec un seul élément (ce qui nous intéresse pas) l'encodage à un haut niveau pose l'inconvénient qu'une erreur dans une ligne ou une colonne ne donne aucune précision sur l'emplacement exact de l'erreur dans cette ligne ou colonne. Or nous désirons plus de précision sur l'emplacement exact de l'erreur.

Si une erreur survient dans la matrice d'information un *checksum* de ligne et *checksum* de colonnes la mettront en évidence.
Considérons qu'il s'agisse d'une et une seule erreur. L'élément du *checksum* de ligne ne correspondra pas à la somme des éléments calculés pour cette ligne, alors cette ligne contient l'erreur. De la même manière un élément dans le vecteur de sommation de colonne indiquera la colonne contenant une erreur. L'hypothèse sur le nombre d'erreurs suppose qu'il s'agit de la même erreur. On peut conclure que l'erreur se trouve alors à l'intersection des lignes et colonnes correspondantes.

Les auteurs de l'article démontrent que le nombre d'éléments différents entre deux *Full Checksum Matrix* différentes est supérieur ou égal à 4. En effet, si un élément dans la matrice d'information est modifié, pour réussir à garder les *checksums* valides il faudra modifier au moins un autre élément dans la ligne et un élément dans la colonne de l'élement en question, de sorte à masquer la différence. Il faudra en plus masquer ces derniers changements. Ceci est possible en modifiant l'élément à l'intersection de la colonne du premier changement et de la ligne du second changement. Donc les éléments de tout ensemble de 4 éléments appartenant deux à deux aux même lignes et même colonnes sont dépendants les uns des autres.

On peut déduire alors que ne pouvons détecter et corriger une erreur (rappelons qu'une erreur est une modification d'un élément dans la matrice) en supposant que les trois autres éléments avec lesquels un élément est lié restent inchangés. La correction se base sur la redondance de l'élément de la matrice d'information dans le calcul du *checksum* de sa ligne, de sa colonne et de la matrice entière. Après avoir détecter l'erreur à l'intersection de la ligne et de la colonne ou se trouve l'incohérence, pour corriger on ajoute à l'élément la différence entre la somme de la ligne ou colonne et le *checksum* de celle-ci, par contre si l'erreur est dans le *checksum* même on remplace l'élément erroné par la somme calculé pour sa ligne ou colonne concernée.

Chapitre 2

Implémentation

Après cette description détaillée du principe de l'algorithme présentée par K.-H. Huang et J.A. Abraham que nous appellerons par la suite ABFT (*Algorithm-Based Fault Tolerance*), notre objectif principal serait de concrétiser cette idée et ceci en implémentant un système qui met l'algorithme en action.

2.1 Choix du Langage

Plusieurs types de langages existent. Il faudra avant tout choisir le type de langage et puis sélectionner le langage le mieux adapté à nos besoins.

Vu que nous nous intéressons seulement au calcul matriciel, nous désirons manipuler des données de type matrice. Alors dans ce cas, la matrice est une structure sémantique indépendante qui rassemble des données et des traitements sur ces données. En plus nous devons stocker l'état d'une matrice avant les calculs et son évolution au cours de l'exécution de ces calculs et de la correction. Nous sommes en face d'un problème nécessitant alors le **paradigme objet**.

Mais quel langage objet choisir ? Deux langages de programmation orientés objets se présentent à l'appel : Java et C++. Java est issu de C++ et présente plus de sureté au niveau de la gestion de mémoire du à l'absence des pointeurs. Il offre le mécanisme des interfaces facilitant la tache du développeur. Par contre il est beaucoup plus lent que le C++. Dans le cadre du développement d'un système concernant une tache aussi répétitive et bas niveau que le calcul matriciel, nous désirons avoir en main le langage produisant le programme le plus efficace possible. Dans ce cas **C++** répondrait mieux à cet offre. Même s'il est moins tolérant que Java à une mauvaise conception c'est pas problème vu que nous tacherons de réaliser une conception qui a le gout d'être efficace tout en étant simple et logique.

2.2 Conception

2.2.1 Contrats

Dans cette partie nous proposons de réaliser une conception du système à réaliser. Sachant que ce dernier évoluera tout le long du développement du projet, il faudra que la conception permette les changements avec le moindre coût.

Pour cela il faut définir un **contrat d'utilisation** de chaque classe. Une classe utilisera une autre via son contrat d'implémentation sans se soucier de la manière dont une fonction a été implémentée. Différentes classes peuvent implémenter le même contrat et ainsi proposer les mêmes fonctionnalités. Quand c'est le cas l'appel d'une de ces classes se fera à l'initialisation ou l'assemblage. Le remplacement effectué aura un impact diminué par rapport à une implémentation sans contrat dans laquelle une modification se propage à divers niveaux.
En Java le mécanisme permettant une telle conception par contrat réside dans les **interfaces** Java. Malheureusement les interfaces dans le sens Java n'existent pas en C++. On sait qu'une interface en Java n'est autre qu'une classe abstraite pure dans laquelle aucune fonction n'est implémentée. On peut dire en quelque sorte qu'une interface au sens Java est une classe abstraite. Les classes abstraites étant un concept existant en C++, nous pourrions les utiliser pour simuler des interfaces représentant les contrats que nous voulons implémenter. Les méthodes déclarées dans cette classe sont toutes précédées du mot clé virtual qui fait d'elles des fonctions virtuelles permettant de les implémenter dans les classes filles.

Dans nos explications nous utiliserons alors le terme interface pour qualifier ces contrats. Tout le squelette de la conception se fera alors à travers ces interfaces. Elles sont toutes distinguées par un préfixe « I » précédent leur nom. A chaque interface (classe abstraite) correspondra au moins une implémentation (classe concrète) du même nom sans le préfixe « I ».

Notons que nous désirons obliger l'implémentation de toutes les fonctions de l'interface définie et interdire l'instanciation de ces interfaces sans passer par une implémentation. Ceci est possible si les fonctions que nous déclarons sont **virtuelles pures**. On atteint cet objectif juste en rajoutant « = 0 » à la déclaration de chaque fonction.

2.2.2 Types Génériques

Les données numériques à traiter au sein du calcul matriciel sont les flottants. Par contre plusieurs choix de représentation de flottants sont possibles. Le choix de la représentation dépend principalement de la précision avec laquelle nous voulons stocker nos données. On peut citer les types de base : les flottants a simple precision, double précision représentant des réels à 32 et 64 bits respectivement. Pour une précision encore plus importante il existe des bilibothèques permettant d'étendre arbitrairement cette précision comme GMP (Gnum Multiple Precision Arithmetic Library). Pour ne pas avoir à limiter notre système à un seul type de données, il faudra implémenter ce système pour effectuer un

calcul sur toutes les représentations de flottants possible. Le traitement de ces données (addition, multiplication, division, etc.) est invariant. Il est donc inutile d'implémenter deux versions d'un objet juste pour modifier le type de données en entrée. On veut réussir à factoriser le code pour éviter un tel problème, et paramétrer les fonctions d'une manière permettant de lui indiquer le type de donnée que nous désirons traiter. Le C++ permet de résoudre ce problème grâce aux paramètres génériques ou ***template***.

Nous tacherons alors à profiter des paramètres *template*, des fonctions *template* et des classes *template*. Pour cela, toutes nos classes ayant une certaine relation avec les données que nous manipulons sont des classes *template* ayant comme paramètre générique un type T qui peut être les entiers ou les flottants. La déclaration des paramètres *template* se fait de la manière suivante :

template <class T> class nom ou nom est le nom de la classe en question.

Bien que le *template* soit un mécanisme très pratique, il soulève un problème d'instanciation assez important. Les *templates* doivent impérativement être définis lors de leur instanciation pour que le compilateur puisse générer le code de l'instance. Ceci signifie que la déclaration et la définition des différentes fonctions membres doit être réalisé au sein d'un même fichier. En temps normal, moins nous avons de fichiers sources mieux on se porte. Pour garder leur savoir faire à la distribution d'une bibliothèque les développeurs fournissent juste les déclarations de leur code. Par contre avec les *templates* cela n'est pas possible étant donné que les déclarations et les définitions sont dans le même fichier.
La solution proposée à ce problème est d'effectuer l'instanciation explicite du *template* dans la partie définition. Cette instanciation se fait en donnant une valeur par défaut à T. On rajoute alors au fichier contenant les définitions « .cpp » la ligne ci-dessous si on veut instancier le *template* pour des types de données réelles :

template class NomDeLaClasse<double> ;

2.2.3 Matrices

Nous avons conclus dans la partie **1.2** du chapitre **1** que nous manipulerons quatre types de matrices dans le cadre de ce projet : une matrice sans extensions, une *Row Checksum Matrix*, une *Column Checksum Matrix* et finalement une *Full Checksum Matrix*. Tous ces types de matrices ont des points communs qui forment les propriétés de bases d'une matrice (sans extensions), cependant présentent des différences au niveau de leurs extensions.

Il est tout à fait naturel de vouloir créer un type d'objet pour chaque type de matrice. Ainsi nous avons créé quatre interfaces définissant les contrats de ces quatre objets. Vu que nous sommes dans le monde objet de C++, nous profiterons ainsi de l'héritage que C++ offre pour réunir les propriétés et les traitements communs dans une interface. Cette interface doit être la plus générale dans la hiérarchie des matrices. Celle-ci sera relative à la matrice sans extension, dans ce cas appelée ***IMatrix***. On remarque de même qu'une *Full Checksum*

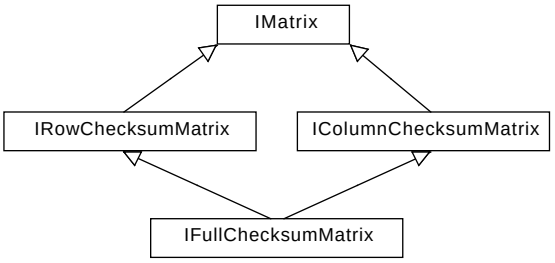

FIGURE 2.1 – Diagramme des Interfaces des Matrices

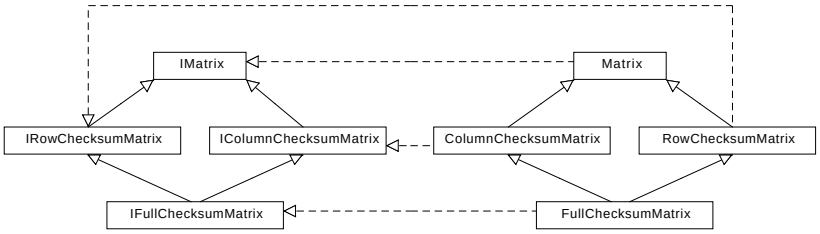

FIGURE 2.2 – Diagramme des Interfaces des Matrices et de leurs implémentations

Matrix est une *Column Checksum Matrix* d'une *Row Checksum Matrix*. L'interface ***IFullChecksumMatrix*** peut alors hériter des interfaces ***IRowChecksumMatrix*** et ***IColumnChecksumMatrix***. Ainsi, nous aurons l'arbre de la figure 2.1.

Avant de nous arrêter sur le problème que peut poser une pareille structure, nous voulons juste dire que chacune des interfaces sera implémentée par une classe qui respectera le contrat, ainsi au cas où une classe doit être remplacée dans la suite du projet on aura pas des problèmes de compatibilité avec les autres classes. L'arbre des classes d'implémentation de ces interfaces est calqué sur l'arbre des interfaces et nous obtenons alors, le diagramme de la figure 2.2. Dans ce diagramme les flèches pointillées reflètent les liens d'implémentation des interfaces même si en C++ ce lien doit être le même que celui de l'héritage, nous avons voulus faire la distinction.

Le Problème du diamant

La structure de l'arbre faite ci-dessus est dite une structure d'héritage en diamant. Elle pose quelques problèmes à la compilation.
Vu que FCM hérite de CCM et RCM et les deux héritent à leur tour de *Matrix*, FCM peut contenir deux copies des membres de classes *Matrix* dont elle hérite indirectement (1 copie des membres hérités par CCM et un autre des membre

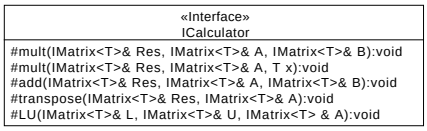

FIGURE 2.3 – Diagramme de l'interface ICalculator

hérités par RCM). Quand nous faisons appel à une fonction de la classe *Matrix* à partir d'une instance FCM, celle-ci possédant deux sous objets *Matrix*, le compilateur aura un choix ambiguë à faire et ne saura alors laquelle des deux choisir. Alors il signalera cette ambigüité.

Encore une fois le mot clé ***virtual*** est la solution au problème. En utilisant *virtual* dans la déclaration de CCM et RCM. Une seule copie de *Matrix* se trouvera dans FCM, il n'y aura alors aucune ambigüité pour le compilateur.
En effet, *virtual* fait en sorte que CCM et RCM n'instancient pas *Matrix* et laissent cette tache à FCM qui instancie sa copie de *Matrix*. Cela veut dire en d'autres termes que le constructeur de FCM doit faire appel au constructeur de *Matrix* pour que son instanciation soit complète sinon une erreur sera produite.

Vecteurs de Sommation

Toutes les matrices étendues possèdent au moins un vecteur de sommation. Pour cela, nous avons choisis de faire de ce vecteur un objet. Nous créons donc l'interface ***IVector*** qui représentera ce vecteur. Celui-ci sera naturellement un champ des classes d'implémentation relatives aux interfaces des matrices étendues. Ce champ aura la possibilité d'être visité et modifié dans ces classes. Un vecteur étant tout simplement une matrice à une seule colonne ou bien une seule ligne, *IVector* héritera alors de *IMatrix*.

2.2.4 Processeurs et Calculs

On sait que dans une machine les calculs sont effectués dans le processeur ou CPU. Autre que la tache des calculs arithmétiques celui ci gère la tolérance aux fautes. Le système de tolérance aux fautes que nous réalisons est un système logiciel. Donc nous sommes dans l'obligation de simuler les différentes tâches du processeur : du calcul, en passant par la détection d'erreur, la correction de ces erreurs, arrivant même à la génération d'erreurs. Ces fonctionnalités sont distribuées sur tout le système. Ainsi, c'est la classe ***FullChecksumMatrix*** qui s'occupera du test sur l'existence d'erreurs dans la matrice qu'elle encode. Si une erreur est trouvée une tentative de correction peut être mise en place. Cette matrice est bien le résultat d'un calcul réalisé par le processeur. Alors, nous implémentons les algorithmes de calculs cités dans la partie **1.3** pour réaliser les calculs sur les matrices. Pour cela, il faut implémenter le contrat que nous définissons dans l'interface ***ICalculator*** de la figure 2.3.

L'implémentation de cette interface peut être effectuée de plusieurs manières différentes. On pourra utiliser des bibliothèques déjà faites pour l'implémenter et ainsi profiter des propriétés et des avantages présentés par ces bibliothèques - Parmi ces bibliothèques nous citons **ATLAS**, **GotoBlas** et **IntelMkl** - ou bien essayer d'implémenter nos propres algorithmes. Puis on pourra comparer les implémentations entre-elles pour choisir la plus performante. Pour réaliser ce but nous devons avoir la possibilité de choisir l'implémentation que nous voulons à l'assemblage. Les différentes implémentations de *ICalculator* auront un nom préfixé par *Calculator*. Notre solution utilise le mécanisme de **délégation** et le patron de conception (design pattern) ***Strategy***. Il s'agit d'un patron de conception de type comportemental grâce auquel des algorithmes peuvent être sélectionnés à la volée au cours du temps d'exécution selon certaines conditions, comme les stratégies utilisées en temps de guerre d'ou son nom.
Ainsi une classe ***Processor***, dont le nom reflète la fonctionnalité se chargera du calcul matriciel sans vraiment faire le travail. On déclare dans cette classe un champ de type *ICalculator*. Ce champ pouvant être n'importe quel type de calculateur on le choisit à la construction du processeur. A chaque fois que nous appelons une fonction de calcul dans la classe ***Processor***, celle-ci procèdera à l'extension des matrices avant de délèguer le calcul à la classe stockée dans ce champ qui s'occupera du reste. Puis, le processeur appellera la fonction de vérification et correction d'erreurs sur la matrice résultat. Toutes les matrices résultats sont stockées dans leur forme de FCM, c'est pour cette raison que cette fonction est dans la classe *FullChecksumMatrix*.

2.2.5 Génération d'erreurs

Bien que le vrai processeur de l'ordinateur puisse produire des erreurs, on peut attendre indéfiniment jusqu'à ce qu'une erreur soit produite. Cette attente n'est pas le seul problème. Si on n'est pas notifié qu'une faute a été induite à un calcul, on peut croire que le système que nous avons réalisé fonctionne parfaitement tant qu'aucune erreur n'apparaisse, et que ce système est systématiquement entrain de corriger les éventuelles erreurs qui apparaissent d'une manière silencieuse. Ayant déjà simuler les calculs qu'effectue un processeur rien nous interdit de simuler la génération des erreurs. Si nous avons ce mécanisme en main, nous pourrions avoir le contrôle de la génération d'erreurs. On peut ainsi injecter une faute dans le calcul quand nous le désirons et observer la réaction du système pour traiter cette erreur. On peut aussi injecter un nombre d'erreurs bien précis avec des emplacements précis comme : deux erreurs dans la même ligne ou bien la même colonne, une erreur dans le checksum, deux erreurs aléatoires, 4 erreurs ...et ainsi de suite on peut énumérer tous les différents cas auxquels on peut penser.

Nous choisissons alors de réaliser une interface ***IErrorGenerator*** pour s'occuper de cette tâche. Dans cette interface nous avons une méthode ***generateError()*** qui génère un certain nombre d'erreurs (précisé par un paramètre) dans la matrice passée en argument, dans des intervalles de lignes et de colonnes bien précises.

Cet outil que nous implémentons est largement suffisant pour injecter des fautes dans les matrices. Pour réussir à le faire constamment sans arrêter les

calculs, en d'autres termes faire de la génération d'erreurs simulée un événement vraisemblable, il faudra que son exécution s'effectue en tâche de fond.
Ceci est possible grâce à ce que nous appelons les ***Threads***. Pour présenter rapidement les *threads*, on peut dire que c'est un mécanisme de lancement de pseudo processus très légers qui partagent la même mémoire virtuelle que le processus principal. Leur intérêt réside dans le fait que plusieurs threads ou fils d'exécution peuvent être lancés en même temps. Ils s'exécutent en parallèle et communiquent entre eux grâce à l'API du langage grâce auquel nous les lançons. La fonction de génération d'erreurs sera alors lancée pour s'exécuter en tache de fond grâce à un thread. La tache principale étant le calcul lui-même.

Remarque sur l'implémentation de la méthode de génération d'erreurs

La méthode de génération d'erreurs generateError() prend 6 arguments :
- **M** la matrice dans laquelle la méthode injecte l'erreur
- **nb** le nombre d'erreurs à générer
- **iMin** la borne inférieure de l'intervalle de ligne
- **iMax** la borne supérieure de l'intervalle de ligne
- **jMin** la borne inférieure de l'intervalle de colonne
- **jMax** la borne supérieure de l'intervalle de colonne

Les erreurs sont générées aléatoirement et itérativement dans la sous matrice déterminée par les deux intervalles de ligne et de colonne. Ainsi, deux erreurs peuvent tomber à la suite dans la même case et le nombre d'erreurs que nous voulons injecté sera réduit de la sorte. Pour éviter ce problème nous stockons les indices des cases changés aux itérations précédentes.

Chapitre 3

Autour de la correction d'erreur

Nous avons déjà expliqué le principe qui réside derrière la correction d'erreurs. Cette correction utilise la propriété du *checksum* pour détecter une erreur et profite de la redondance d'information pour tenter une correction de l'erreur détectée. Dans la présentation du principe nous avons considéré qu'il existe une et une seule erreur dans la matrice. Dans cette partie du rapport nous aborderons les différents cas présentés par la détection d'erreur. On verra que la correction d'une case erronée n'est autre que la solution d'un système d'équation. Nous discuterons aussi des cas ou la correction n'est pas certaine ou bien ne peut pas être effectuée.

On sait que pour détecter une faute dans une ligne ou une colonne, on calcule la somme de celle-ci et on la compare à la valeur du *checksum* stocké dans la matrice étendue. Si les deux sont égaux alors y a éventuellement de très grandes chances qu'il n' y pas d'erreur dans cette ligne. Cette incertitude est due au fait que plusieurs erreurs peuvent se masquer les unes les autres c.-à-d. que plusieurs éléments sont faux mais leur somme reste inchangée. Soit $x_1, x_2, \dots x_n$ les valeurs des éléments présents dans une ligne ou bien une colonne et c la valeur du *checksum* de cette ligne ou colonne. nous devons vérifier l'égalité suivante :

$$x_1 + x_2 + \ldots + x_n = c$$

Cette dernière peut être réécrite de la manière suivante :

$$x_1 + x_2 + \ldots + x_n - c = 0$$

Bien que les deux écritures soient complètement équivalentes la deuxième présente un avantage important. On suppose qu'une erreur peut survenir dans n'importe quelle case de la matrice, les *checksums* inclus. Or avec la première écriture on devra effectuer plusieurs tests pour savoir si l'erreur est dans le *checksum* ou bien dans la matrice d'information, alors que l'utilisation de la deuxième ne fait aucune hypothèse sur le type de l'élément erroné. Quand l'égalité n'est pas respectée la ligne ou la colonne entière est considérée fausse. C'est l'intersection de cette ligne ou colonne avec les autres qui déterminera l'emplacement de l'erreur.

L'erreur par la suite sera traitée selon cet emplacement facilement trouvé. S'il s'agit du *checksum* on le remplacera par la somme de la ligne, sinon on ajoutera la différence trouvée par l'égalité à l'élément en erreur.

D'une manière générale le travail consiste à résoudre un système contenant les équations des lignes et colonnes non valides. Le nombre d'équations est alors égal au nombre de ces lignes et colonnes et le nombre d'inconnues est celui des intersections entre ces lignes et ces colonnes.
Déterminer le nombre de solutions pour ce système permet de discuter de la possibilité ou pas d'une correction.

3.1 Une erreur

S'il existe une seule erreur, nous avons alors une ligne et une colonne non valides. Vu que nous avons une intersection alors, il s'agit dans ce cas de résoudre un système de deux équations à une inconnue.
Tant que le nombre d'inconnues est strictement inférieur au nombre d'équations le système a une solution unique.Donc dans le cas d'une seule erreur la correction est possible.

3.2 Deux erreurs

3.2.1 Deux erreurs dans la même ligne ou la même colonne

Soit une matrice à m lignes et n colonnes. Supposons que nous avons deux erreurs dans la même ligne de la matrice. Deux cas sont possibles :

- Ces deux erreurs s'annulent : Si c'est le cas, nous n'avons pas une équation concernant une ligne erronée, par contre nous avons deux équations pour des fausses colonnes. Dans ce cas il n'y a alors aucune intersection. Donc le nombre d'inconnues et alors égal à deux fois le nombre de lignes de la matrices : ici $2 \times m$.
  Le système à résoudre est un système de deux équations à $2m$ inconnues. $2m$ étant supérieure à deux alors la correction dans ce cas est impossible. Heureusement, K.-H. Huang et J.A. Abraham démontrent dans l'article que le pourcentage de détection d'une erreur générée aléatoirement dans une FCM croit très vite en fonction de la taille de la matrice vers les 99.99%. Donc le masquage d'erreur est très peu probable.
- Les deux erreurs ne s'annulent pas : Dans ce cas nous avons alors trois équations à 2 inconnues. Une équation pour la ligne erronée et deux équations pour les deux colonnes erronées et une intersection entre chaque colonne et la ligne (d'ou les deux inconnues). Le nombre d'équations étant supérieur au nombre d'inconnues le système admet une solution unique. On pourra corriger les deux erreurs.

Le cas de deux erreurs sur une même colonne est un cas analogue au précédent. C'est la raison pour laquelle nous ne le détaillerons pas.
On peut généraliser ce cas pour n erreurs sur une même ligne ou colonne. Supposons que les erreurs ne s'annulent pas alors la ligne contenant ces erreurs indique une erreur. Alors l'équation de cette ligne est une équation du système qu'on doit résoudre. De même les équations de n colonnes ne seront pas valides.

Ces équations font partie elles aussi du système. Le système est ainsi formé de $n + 1$ équations. Chaque colonne est intersectée par la ligne qui présente l'inégalité. Nous avons alors n intersections d'où n inconnues.
On sait qu'un système de $n + 1$ équations à n inconnues admet une solution unique. Alors dans ce cas la correction de la matrice est possible.

3.2.2 Deux erreurs pas sur la même ligne ou la même colonne

Dans le cas où les deux erreurs ne sont pas dans la même ligne ou bien la même colonne, celles-ci ne peuvent pas s'annuler. Alors nous aurons deux inégalités au niveau des lignes et deux autres au niveau des colonnes. Ces inégalités permettent alors d'avoir un système à quatre équations. Grâce aux quatre intersections entre les lignes et les colonnes en questions nous obtenons quatre inconnues.
Ce système de quatre équations à quatre inconnues admet un ensemble de solution et non une solution unique. Quelle solution choisir ? Pour répondre à cette question il faudra rajouter plus de contraintes sur les données et ceci grâce à une hypothèse. Le choix de ces contraintes est assez subjectif mais nous essayerons de le faire d'une manière logique. Les erreurs dans le calcul sont très peu fréquentes. Leur nombre est négligeable par rapport au nombre d'instructions effectuées par le processeur entre deux erreurs. Alors la probabilité d'avoir deux erreurs dans une matrice est supérieure à celle de quatre erreurs. Partant de ce postulat, le choix de la solution serait un choix optimal se basant sur la minimisation du nombre d'erreurs.

Donc, la solution que nous proposons est la suivante :
On suppose que deux des quatre éléments aux intersections sont faux. On fixe ainsi les deux autres à leur valeur dans la matrice. On tente une correction de ces deux erreurs dans la matrice, puis on teste si tous les vecteurs de sommation des lignes et des colonnes correspondantes à ces éléments sont valides. Si c'est le cas, on s'arrete ici, car la correction est valide. Sinon on fixe l'autre couple d'éléments, puis on tente une correction. Si aucun couple ne permet d'avoir une correction valide, on effectue le même travail mais cette fois-ci en fixant des triplets d'éléments. Dans tous les cas, on s'arrete à la premiere correction valide et celle-ci sera la solution optimale. Si par malheur aucune correction est valide on dit que la correction n'est pas possible car le système admet plusieurs solutions pour quatre inconnues.

Nous avons alors énuméré tous les cas possibles jusqu'à deux erreurs.

3.3 Taille des conteneurs de données

Nous remarquons que le nombre d'erreurs détéctées est parfois plus grand que le nombre d'erreurs réel.

Ceci est dû à la taille des conteneurs des valeurs des éléments et surtout à la valeur du *checksum* qui ne correspond pas toujours à la somme de sa ligne ou sa colonne. Les flottants doubles sont stockés sur 64 bits en C++ alors si jamais un nombre doit etre stockée sur plus de 64 bits sa valeur est tronquée et ne correspond donc plus à la vrai valeur calculée
Par exemple, dans un calcul en flottants IEEE double précision, $(2^{60}+1)-2^{60}$ ne donne pas 1, mais 0. La raison est que $2^{60}+1$ n'est pas représentable exactement et est approché par 2^{60}.

Pour contourner cette erreur, il suffit d'avoir plus de bits pour stocker les nombres. Dans la deuxième partie de notre projet nous utiliserons une bibliothèque de sur-précision pour stocker nos résultats. Nous examinons pour l'instant la bibliothèque GMP.

Conclusion

Nous avons abordé dans ce rapport, d'abord le principe de l'ABFT présenté par K.-H. Huang et J.A. Abraham. Ensuite, nous avons discuté de nos choix d'implémentation de cet algorithme partant du choix du langage, passant par la conception par contrats, les types génériques et la simulation de génération d'erreurs ainsi que les solutions possibles au problèmes que nous pose cette implémentation comme le patron de conception *Strategy*, et les fonctions virtuelles pures. Enfin, nous avons étudié en détail la capacité de cet algorithme à détecter et à corriger les erreurs.

Cette dernière partie reflète les limites de cette méthode. Ainsi on peut corriger avec certitude seulement une erreur, cependant la correction concernant deux erreurs dépend de plusieurs facteurs dont certains que nous avons fixé pour permettre cette correction. Il s'agit de la limite de l'utilisation des propriétés du *checksum*.

Nous savons maintenant que plus il y a de la redondance plus nous pouvons corriger des erreurs. Nous remplacerons alors dans la deuxième partie de notre projet le *checksum* par une méthode conduisant à plus de redondances de manière à avoir un système qui tolère plus de fautes.

Bibliographie

[1] KUANG-HUA HUANG, MEMBER, IEEE, AND JACOB A. ABRAHAM, Algorithm-Based Fault Tolerance for Matnx Operations. IEEE TRANSACTIONS ON COMPUTERS, VOL. c-33, NO. 6, June 1984.

[2] CppUnit, http ://sourceforge.net/projects/cppunit/.

[3] ATLAS (Automatically Tuned Linear Algebra Software), http ://math-atlas.sourceforge.net/.

[4] GotoBLAS2, http ://www.tacc.utexas.edu/tacc-projects/.

[5] Intel® Math Kernel Library (Intel® MKL), http ://software.intel.com/en-us/intel-mkl/.

[6] The MPACK : Multiple precision arithmetic BLAS (MBLAS) and LAPACK (MLAPACK), http ://mplapack.sourceforge.net/.

[7] Bruce Dawson, Comparing floating point numbers, http ://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm.