CONSERVATOIRE NATIONAL DES ARTS ET METIERS
CENTRE REGIONAL DE LORRAINE

ACCELERATION DE LA CLASSIFICATION DE SOURCES SONORES POUR LE TRAITEMENT DE SONS DE TRES LONGUES DUREES

MEMOIRE EN INFORMATIQUE

OPTION RESEAUX, SYSTEMES ET MULTIMEDIA (IRSM)

Présenté par

Kevin DEHLINGER

Soutenu le 14 décembre 2012

JURY

Président :	J.P. ARNAUD	Professeur de Chaire du CNAM et président du jury
Membres :	S. VIALLE	Responsable Régional de Spécialité informatique au CNAM de Lorraine, et tuteur du mémoire
	P. MERCIER	Professeur à Supélec et responsable des ressources informatiques et multimédia
	J. GUSTEDT	Directeur de recherche INRIA et responsable de l'Equipe Projet INRIA AlGorille
	G. VECCHIO	Ingénieur NTIC à SIB

Remerciements

Je tiens en premier lieu à remercier Monsieur Stéphane Vialle, enseignant-chercheur, ainsi que la direction de l'école pour m'avoir permis d'effectuer mon stage à Supélec.

Je remercie aussi Monsieur Stéphane Rossignol, enseignant-chercheur, pour sa disponibilité et ses explications, ainsi que l'ensemble des salariés de Supélec pour leur accueil et leur bonne humeur.

Un grand merci également à Madame Brigitte de Metz-Noblat, directrice de la Caisse d'Allocations Familiales de la Moselle, pour m'avoir permis de m'absenter pendant près de neuf mois malgré un contexte délicat.

Mille mercis au FAF Sécurité Sociale ainsi qu'à Uniformation, organismes paritaires collecteurs agréés, pour avoir financé mon congé individuel de formation.

Table des matières

Chapitre 1

Introduction

La classification de sources sonores est une technologie qu'il devient important de maîtriser dans le monde numérique actuel. Son but est de pouvoir étiqueter par type de sons des plages d'un signal audio ou vidéo, à des fins d'indexation ou pour effectuer ensuite des traitements spécifiques sur le signal. Les types de sons peuvent être très généraux (parole, musique, animaux…) ou plus précis (chant, jazz, classique, chien, chat…). Les possibilités d'applications offertes par un tel système sont très variées. On peut par exemple citer des travaux réalisés par l'institut océanographique de Woods Hole aux Etats-Unis qui permettent de classifier les sons provenant de mammifères marins en fonction de leur espèce et de certains types de comportements [1]. D'autres applications apparaissent couramment dans la littérature, telles que l'indexation d'archives audio ou vidéo en fonction du contenu sonore [2] [3], le ciblage afin de n'exécuter des traitements comme la reconnaissance vocale que sur des plages de son contenant effectivement de la voix [2] [4] [5] [6], la reconnaissance de bruits [7], ou encore l'encodage dynamique, pour utiliser le codec[1] audio offrant la meilleure performance en fonction du son actuellement diffusé [5].

1. Contexte de travail

Ce travail de mémoire s'est déroulé sur le campus de Metz de Supélec, au sein du groupe IDMaD (Informatique Distribuée et Masses de Données) de l'équipe IMS.

La base de ce travail de mémoire d'ingénieur CNAM est un outil de classification de sources sonores développé par Stéphane Rossignol, enseignant-chercheur à Supélec. Ce programme, initialement écrit en C parallélisé sur un seul ordinateur multi-cœur, permet de baliser un fichier audio afin d'indiquer les plages temporelles durant lesquelles le fichier analysé est constitué de musique ou de voix.

Parmi l'ensemble des activités de Supélec, se trouve la recherche dans le domaine du calcul parallèle. Le groupe IDMaD s'intéresse ainsi à la parallélisation d'applications réelles (industrielles ou académiques) pour guider une recherche amont sur de nouveaux modèles et outils de programmation parallèle vers les besoins des utilisateurs. Dans ce cadre, l'étude d'applications parallèles comportant de nombreuses entrées/sorties est un objectif du groupe IDMaD depuis 2010. Les entrées/sorties sont en effet de plus en plus fréquentes

[1] Bibliothèque permettant de compresser/décompresser des sons ou des vidéos dans un format particulier.

dans les applications parallèles mais restent difficiles à exprimer facilement avec les outils de programmation parallèle. Des études de cas sont encore nécessaires, comme la « classification de sources sonores » étudiée dans ce mémoire.

Pour mener ces recherches, Supélec dispose d'un cluster de production de 256 PCs, ainsi que de plusieurs petits clusters d'expérimentation, et réalise chaque année des études académiques et industrielles sur ces plateformes.

2. Présentation de l'outil original de classification de sources sonores

Le fonctionnement de ce programme repose sur l'utilisation simultanée de quatre algorithmes de discrimination afin d'avoir un résultat d'une grande précision. Ceux-ci fournissent des résultats que nous nommerons *caractéristiques à court terme*.

Les différentes caractéristiques à court terme sont :

- le **flux spectral**, c'est-à-dire la variation du spectre entre deux échantillons de longueur identique, séparés de 5ms ;
- le **taux de passage par zéro** est une représentation du nombre de changements de signe du signal ;
- le **cepstre**, défini comme étant la transformée de Fourier inverse du logarithme du spectre du signal [8] ;
- le **centroide spectral** est le centre de gravité du spectre du signal [9].

Ces caractéristiques sont communément utilisées dans des programmes de reconnaissance vocale ou de classification de sources sonores. Dans notre cas elles sont calculées sur des échantillons de 20ms, et représentent ce que nous appellerons *une trame*. Les trames, dessinées en rouge sur le schéma suivant, commencent 5ms après le début de la précédente de manière à avoir un taux de chevauchement de 75%.

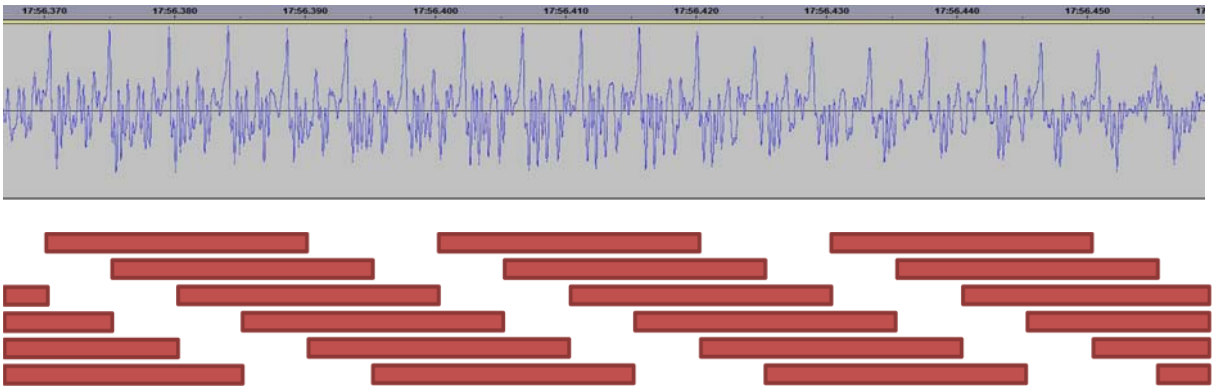


Figure 1.1 : représentation visuelle des trames.

En utilisant les trames, nous pouvons calculer les *caractéristiques à long terme*, qui sont au nombre de huit :

- la moyenne des flux spectraux ;
- la variance des flux spectraux ;
- la moyenne des taux de passage par zéro ;
- la variance des taux de passage par zéro ;
- la moyenne des cepstres ;
- la variance des cepstres ;
- la moyenne des centroides spectraux ;
- la variance des centroides spectraux.

Ces caractéristiques à long terme sont calculées en utilisant le nombre nécessaire de trames pour couvrir une période de 1s et représentent *un segment*. Les segments, représentés en bleu dans le schéma suivant, commencent 250ms après le début du segment précédent afin d'avoir un taux de chevauchement de 75%.

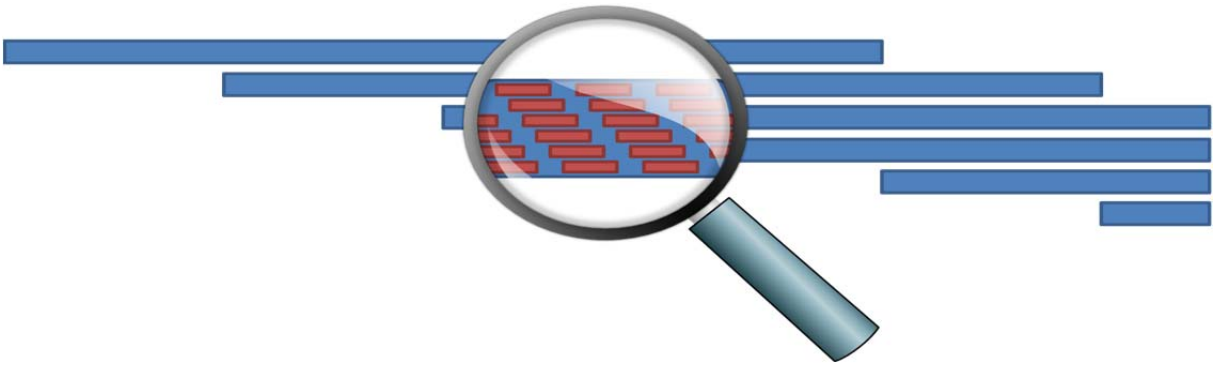


Figure 1.2 : représentation visuelle des segments.

Pour déterminer s'il s'agit de musique ou de voix, une comparaison est effectuée pour chaque caractéristique à long terme, entre les segments calculés et deux bases précalculées pour la musique et la parole. L'ensemble dont les données d'entraînement sont les plus proches des données calculées est déclaré vainqueur de la comparaison et le segment en question est étiqueté en conséquence.

L'intérêt d'un tel outil dépend à la fois de la qualité des résultats obtenus, mais également de son évolutivité, de sa rapidité d'exécution, et de la taille des données qu'il est capable de traiter. Les résultats obtenus par l'application originale sont considérés comme corrects. Celle-ci ne permet toutefois pas de traiter des fichiers de taille importante en raison de limitations inhérentes aux mécanismes de lecture implantés et n'est pas très performante en termes de rapidité d'exécution. De plus, l'ajout de nouvelles caractéristiques n'est pas chose aisée à cause de la forte imbrication de leurs algorithmes de calcul dans les boucles de traitement.

3. Problème de limitation de taille du fichier de données audio

Après étude du programme initial, il s'avère que l'algorithme global est composé de quatre grandes étapes. La taille maximale du fichier à traiter n'est pas explicitement fixée dans le code, mais dépend de la machine sur laquelle l'outil est exécuté.

La bibliothèque utilisée pour faire la lecture est libsndfile. Elle permet de décompresser un grand nombre de formats audio tels que ogg, flac ou PCM. Cette bibliothèque libre est également capable de convertir des fichiers d'un format vers un autre, ou simplement d'écrire de nouveaux fichiers audio[2]. Elle est reconnue et présente dans les « dépôts d'application » de la majorité des grandes distributions de linux, mais fonctionne également sous Windows.

Dans notre cas, libsndfile sert à récupérer des informations du fichier audio, tels que le taux d'échantillonnage, le nombre de canaux, ou encore la taille en nombre d'échantillons et à réaliser la lecture afin de stocker les données brutes provenant du fichier dans un tableau.

Cette bibliothèque fournit les fonctions nécessaires aux entrées/sorties avec des fichiers audio :

```
SNDFILE* sf_open (const char *path, int mode, SF_INFO *sfinfo);
```

Renvoie un descripteur et alimente une structure avec les informations sur le fichier.

```
sf_count_t sf_seek (SNDFILE *sndfile, sf_count_t frames, int whence);
```

Permet de déplacer la position du pointeur dans le fichier et renvoie l'offset depuis le début.

```
sf_count_t sf_read_double (SNDFILE *sndfile, double *ptr,
sf_count_t items);
```

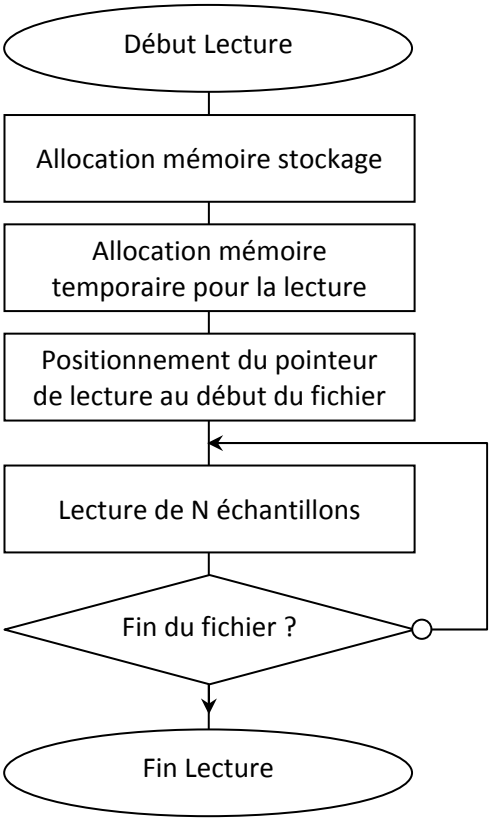
Remplit le tableau pointé par `ptr` avec les données lues et renvoie le nombre d'éléments lus. Des fonctions similaires existent pour d'autres types comme les `float` ou les `int`.

```
int sf_close (SNDFILE *sndfile);
```

Ferme le fichier, renvoie la valeur 0 s'il n'y a pas eu de problème, et libère les variables utilisées en interne.

[2] Site officiel contenant la documentation ainsi que le code source : http://www.mega-nerd.com/libsndfile/

La lecture dans le programme original fonctionne de la manière suivante :



La partie « Lecture » débute par l'allocation dynamique d'une zone mémoire contiguë permettant de stocker intégralement la première piste du fichier audio à traiter.

Une seconde allocation mémoire dynamique permet de stocker temporairement N échantillons pour l'ensemble des pistes. La première piste est extraite de ces données et enregistrée dans le buffer de stockage.

Un bouclage est ensuite effectué tant que la fin du fichier n'est pas atteinte. Lorsque la lecture est finie, le buffer de stockage contient l'intégralité de la première piste du fichier audio à traiter.

Figure 1.3 : fonctionnement de la lecture dans le programme original.

La taille maximale du fichier dépend donc de la quantité de mémoire disponible lors de l'exécution. De plus, la zone mémoire allouée pour le stockage et traitement de la première piste est déclarée en tant que `double` alors que la résolution maximale des fichiers à traiter n'est que de 32 bits. L'utilisation de `float` permet déjà de doubler la taille maximale du fichier. Toutefois, ce gain ne résout pas le problème : la limite est repoussée, mais existe encore. Un autre axe d'amélioration est donc de ne plus lire le fichier d'une traite, mais de l'analyser morceau par morceau en séquençant le traitement.

Le tableau donne l'équivalent de la durée du signal stockable en 100 Mo de données, en fonction des traitements effectués et de l'obtention de résultats de plus en plus synthétiques.

Elément	**Commentaire**	**Stockage en 100Mo**
Echantillons bruts	44100 ech/s de 1 `float`	9m54s
Trames	4 `double` pour 20ms de signal avec chevauchement de 75% des trames	4h33m
Segments	8 `double` pour 1s de signal avec chevauchement de 75% des segments	4j 17h46m

Il est donc possible, dans un buffer de 100 Mo, de stocker les segments calculés à partir d'un fichier audio dont la durée dépasserait les 4 jours et dont la taille serait de près de 70 Go, ce

qui excède largement les spécifications du format WAVE (4 Go). Mais le format RF64 qui est né du besoin de stockage et de transmission de fichiers audio multi canaux non compressés plus volumineux permet d'utiliser des fichiers de très grande taille [10].

Pour repousser cette limite, nous allons nous attacher à morceler la lecture du fichier. Les échantillons bruts qui ne servent que pour le calcul des trames ne seront pas conservés durant toute l'exécution du programme. Il en va de même pour les trames dont le seul intérêt est de permettre de calculer les segments. Elles seront « oubliées » dès le prochain cycle de lecture pour faire place aux nouvelles. Les segments, quant à eux, seront tous conservés pour permettre une éventuelle normalisation dynamique des valeurs avant la classification.

L'algorithme permettant de faire cela peut donc être représenté de la manière suivante :

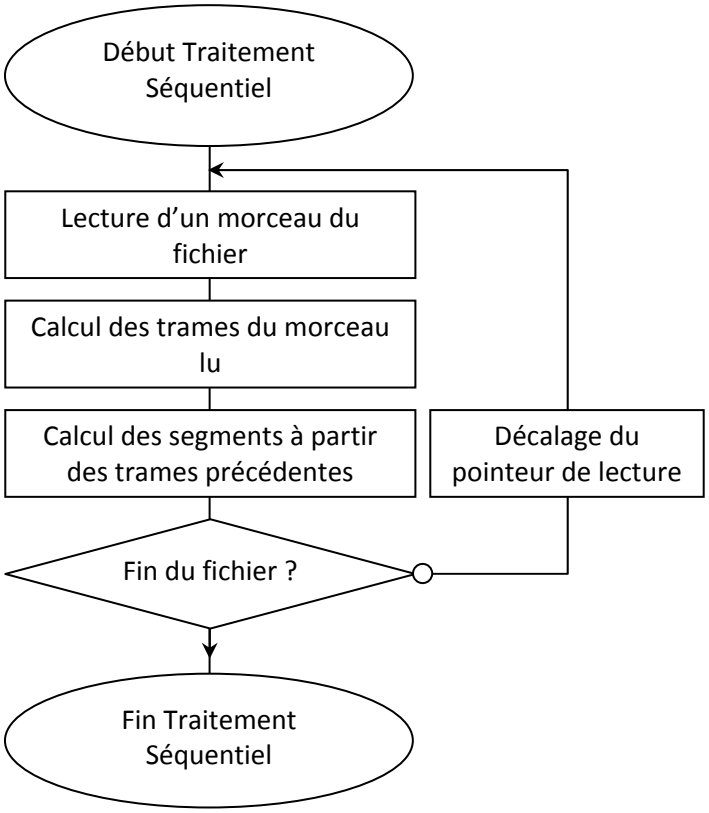


Figure 1.4 : fonctionnement amélioré de la lecture.

Le principe de cet algorithme est de lire une partie du fichier, d'extraire l'intégralité des caractéristiques des échantillons lus et de continuer ainsi jusqu'à la fin du fichier.

Il existe deux problèmes cependant :

- *Pour n'oublier aucun segment, il faut impérativement que la lecture commence par le début des échantillons nécessaires au traitement d'un segment, et s'arrête, de la même manière, sur les données correspondant à la fin d'un segment.*
- *Le début des données nécessaires au calcul du segment N+1 est situé à environ un quart des données correspondant au segment N.*

Cela n'est toutefois pas aussi trivial en pratique, à cause du chevauchement nécessaire entre deux segments successifs, deux trames successives, mais également du fait qu'il est nécessaire pour le flux spectral d'avoir les échantillons commençant 5ms avant la trame que

l'on veut traiter. Il apparaît donc nécessaire de faire un replacement du pointeur de fichier sur le début du segment suivant, en tenant compte des 5ms supplémentaires nécessaires au calcul du flux spectral. Une autre solution serait de copier ces données qui ont déjà été lues lors du cycle précédent vers un buffer temporaire pour les récupérer au besoin. Avec un buffer de lecture de taille importante et une quantité de données à relire de taille minime (environ 133ko pour un fichier audio échantillonné à 44.1kHz) le coût engendré par cette relecture est négligeable : les deux solutions sont acceptables. Pour simplifier la future distribution de l'outil sur cluster, et ainsi éviter des communications supplémentaires entre les nœuds de calcul, le choix du replacement de l'index du fichier a été fait.

4. Besoin d'optimisation du temps de traitement

Le temps d'exécution est un facteur impactant de manière importante le confort d'utilisation du produit, ainsi il apparaît primordial de faire le maximum pour réduire la durée du traitement.

Le programme initial est prévu pour fonctionner sur des ordinateurs disposant de plusieurs cœurs en utilisant la technologie OpenMP [11]. Cela permet de rendre parallèle un programme purement séquentiel constitué de grosses boucles de calculs en utilisant un thread par cœur.

Les zones parallèles dans le programme initial étaient dans la boucle de traitement des trames et dans les boucles de calcul de deux petites fonctions appelées à plusieurs reprises lors du calcul des segments. La phase de classification était effectuée de manière séquentielle.

L'algorithme parallèle de calcul des segments doit être revu du fait de la segmentation de la lecture du fichier, afin de ne pas créer de threads pour faire de petits calculs, puis d'en recréer d'autres quelques instants après qui vont effectuer les mêmes petits calculs sur d'autres données. Le but est de regrouper au sein d'une grande boucle de calcul le plus d'opérations possible afin de réduire les pertes de performances liées à la création/destruction de threads. Dans le domaine du calcul parallèle, on dit que l'on adapte la granularité de la parallélisation.

Paralléliser l'algorithme de classification avec des threads OpenMP est également un axe d'amélioration à suivre afin d'optimiser davantage la vitesse d'exécution.

Un autre axe d'amélioration intéressant est d'utiliser des bibliothèques optimisées pour effectuer certains calculs qui ont été codés en dur dans le programme initial, comme le calcul de la transformée de Fourier par exemple. Un challenge supplémentaire était de pouvoir choisir facilement entre l'algorithme initial de calcul de la FFT, ou celui de la bibliothèque choisie, alors que les interfaces de ces deux solutions sont totalement différentes, et d'éviter de dupliquer le code.

Une fois ces axes d'améliorations parcourus et mis en œuvre, l'optimisation pour l'exécution de l'outil sur une seule machine touchera à ses limites. L'étape logique suivante sera donc de distribuer tout ou partie de ces calculs sur plusieurs machines.

Cependant, un des défis majeurs de la distribution des calculs dans ce projet est lié à la quantité d'entrées/sorties. Le matériel informatique utilisé à Supélec n'est pas prévu pour réaliser de grandes quantités de lectures en parallèle et les fichiers audio à analyser ne sont pas compressés ce qui rend leur taille très importante. Trouver une solution peu coûteuse et fonctionnelle est un paramètre important pour le bon fonctionnement de la distribution de l'application. La façon d'accéder au fichier à traiter par l'outil est un paramètre important à prendre en compte. Il est possible de déployer un fichier sur l'ensemble des machines d'un cluster pour que chaque tâche puisse y accéder directement sur le disque local, malheureusement le déploiement du fichier est très coûteux en temps. En résumé, cette solution permet de disposer facilement d'un jeu de données de test sur chaque nœud, mais n'est pas judicieuse en production à cause du temps de déploiement des fichiers et parce que les nœuds n'ont pas besoin d'avoir accès à l'intégralité du fichier. Elle l'est cependant lors de périodes de développement de nouveaux algorithmes de classification car cela permet de s'affranchir d'une lecture via le réseau lors de tests répétitifs. Il est donc très important de mettre au point une solution permettant d'avoir de bonnes performances soit en lisant le fichier à partir du disque local, soit à partir d'un emplacement réseau centralisé.

Après réflexion, l'architecture adoptée consistera en la spécialisation d'un certain nombre de processus qui ne seront chargés que des entrées/sorties (voir page 36). Ces processus « lecteurs » lisent le fichier à traiter directement à l'endroit qui les concerne et envoient ces données à leurs processus « travailleurs » et qui seront chargés d'analyser ce fragment de fichier.

Nous appellerons un « groupe » la formation d'un lecteur et de ses travailleurs. Les groupes doivent être homogènes pour éviter que certains finissent longtemps avant d'autres. La gestion optimale de ces groupes est un défi à part entière, car il est possible de jouer sur un grand nombre de paramètres, tels que :

- le nombre de processus travailleurs dans un groupe ;
- le nombre de nœuds multi-cœur utilisés pour un groupe ;
- le nombre de processus par nœud ;
- le nombre de threads OpenMP par processus ;
- la possibilité de partager un nœud entre deux groupes.

De plus, nous sommes en droit d'émettre l'hypothèse qu'il existe un paramétrage optimal différent pour chaque configuration possible. Les différences de configuration peuvent être liées, entre autres :

- au nombre de nœuds alloués ;
- au mode de lecture (en local, ou sur réseau) ;
- au réseau (vitesse, latence, charge, …) ;
- à la puissance des nœuds du cluster (CPU, RAM, …).

Il n'est pas impossible que l'utilisateur final n'ait qu'une connaissance minime de la configuration du cluster et ne soit donc pas en mesure de connaître ou de déduire les paramètres optimaux pour celui-ci. Il est donc primordial de proposer un mode d'exécution simple avec une implication minimale de l'utilisateur, tout en garantissant de bonnes performances quelle que soit la configuration (notamment le nombre de nœuds, et le mode de lecture), ainsi qu'un mode laissant aux utilisateurs avancés la possibilité de configurer l'ensemble des paramètres manuellement.

Nous voyons donc clairement apparaître plusieurs scénarios d'utilisation différents :

- Le chercheur en traitement de signal qui développe de nouvelles fonctionnalités : Celui-ci sera amené à tester régulièrement le programme pour vérifier que les évolutions qu'il apporte ne pénalisent pas la qualité des résultats ou les performances. Il est intéressant qu'il puisse donc tester rapidement et régulièrement, sans affecter l'ensemble du réseau. Les fichiers audio sur lesquels il va effectuer ses tests seront souvent les mêmes, c'est pourquoi il est aisé de copier, une fois pour toutes, sa base de fichiers de tests sur l'ensemble des machines du cluster et de le laisser faire ses essais via des lectures en local.

- L'exploitant qui traite de nouveaux fichiers audio : Le plus simple dans ce cas est de traiter les nouveaux fichiers en y accédant par le réseau. De plus, c'est également la méthode la plus rapide en considérant le temps de copie qui serait nécessaire pour déployer chaque nouveau fichier sur l'ensemble des machines. La configuration pour obtenir des performances optimales doit être transparente pour cet utilisateur. Dans l'idéal, il devra lancer la même commande quel que soit le cluster sur lequel il exécute les traitements et obtenir des temps de traitements proches de ceux qui auraient été obtenus avec la meilleure configuration manuelle.

- L'expert en calcul parallèle qui peut être amené à « benchmarker » son cluster pour en améliorer les performances et éventuellement à tester un bon nombre de configurations différentes pour offrir à l'exploitant une heuristique de configuration la plus pertinente possible lors de l'acquisition d'un nouveau cluster ou de l'évolution d'une architecture existante.

5. Manque d'évolutivité de l'application

Il apparaît important de faciliter l'ajout de nouveaux algorithmes de calculs de caractéristiques sonores. Initialement le code relatif au calcul des caractéristiques est réparti dans l'ensemble du code.

Après analyse, une grande partie du code concernant chaque caractéristique est semblable et l'ordre des grandes opérations est le même. La solution envisagée consiste donc à créer une classe pour chaque caractéristique avec son code propre en la faisant hériter d'une classe mère contenant les outils communs à chacune d'elles.

Le travail de ce mémoire d'ingénieur porte donc sur la suppression de la limite relative à la taille du fichier à analyser, l'optimisation du temps de traitement, et l'amélioration de l'évolutivité du code.

Chapitre 2

Etat de l'art applicatif et des mécanismes d'IO intensives

1. La classification de sources sonores

Il existe de nombreux algorithmes pour la classification de sources sonores, mais la majorité des solutions décrites dans la littérature utilisent un fonctionnement en deux étapes :

- l'extraction de caractéristiques permettant de déterminer la nature du son ;
- la classification de ces caractéristiques.

Les efforts dans ce domaine semblent axés dans deux principales directions. La majorité des études se focalisent sur la recherche de nouvelles caractéristiques extraites du signal audio, et permettant de différencier des types de sons, ou bien tentent d'améliorer ou de développer de nouveaux algorithmes de classification. Le but de la classification est de de déterminer, avec la plus faible marge d'erreur possible, la nature du son en fonction des valeurs des caractéristiques pour une plage temporelle donnée.

a. Travaux principalement orientés sur les caractéristiques

La qualité des résultats obtenus dépend en grande partie de la recherche d'indicateurs dans un son qui montreraient que celui-ci est plutôt d'un type que d'un autre. Ces types peuvent être très généraux (parole, musique, …) ou plus spécifiques (classique, jazz, chant, …) en fonction des besoins.

Des chercheurs des Bell Laboratories, dans leur papier « Robust Singing Detection in Speech/Music Discriminator Design », [2] choisissent d'utiliser des caractéristiques inédites et dérivées d'une caractéristique qu'ils ont précédemment développée et nommée « harmonic coefficient ». Leur méthode est basée sur la recherche des spécificités des structures harmoniques d'un son voisé[3]. En plus de ces caractéristiques originales, ils en utilisent d'autres, déjà connues, comme la modulation de l'énergie à 4Hz, qui est plus importante dans un signal parlé et peu présente dans un signal chanté. Ces différentes méthodes leur permettent de distinguer 3 classes de sons : la voix parlée, la voix chantée, et

[3] Le voisement est caractérisé par la présence d'un son harmonique dû à la vibration des cordes vocales (source : http://cnrtl.fr).

la musique. Ils indiquent également l'existence de deux algorithmes de classification couramment utilisés : un utilisant un modèle de mélanges gaussiens (GMM) et un autre basé sur la méthode des k plus proches voisins (k-NN), mais préfèrent GMM pour la suite de leurs travaux.

Un groupe de travail composé d'un chercheur de la Navy américaine et d'une entreprise danoise propose dans le papier « *Speech Music Discrimination Using Class-Specific Features* » [7] l'utilisation de caractéristiques propres aux types de sons classifiables. Leur approche nécessite l'extraction d'une grande quantité de caractéristiques différentes. Cette quantité est étroitement liée au nombre de types de sons que l'on souhaite classifier. En effet, si certaines caractéristiques peuvent être communes, la méthode de ces chercheurs repose essentiellement sur l'emploi de caractéristiques différentes entre les diverses classes de sons. La classification est faite en comparant les scores obtenus après calculs de fonctions de densité de probabilités en sortie des différentes branches de l'algorithme (une branche par type de son).

Un groupe de chercheurs d'une entreprise californienne propose quant à eux d'étudier un grand nombre de caractéristiques basées sur la recherche de certaines propriétés spécifiques à la voix ou à la musique. Ces travaux sont décrits dans leur papier intitulé « Construction And Evaluation Of A Robust Multifeature Speech/Music Discriminator » [4]. Une des caractéristiques, nommée « pulse metric », est nouvelle et permet de déterminer si un son contient un rythme fort souvent retrouvé dans certains styles de musique. Les caractéristiques qu'ils utilisent ont des temps de latence différents, c'est-à-dire que certaines donnent un résultat utilisable après avoir traité seulement quelques millisecondes de son, alors que d'autres nécessitent un traitement sur plusieurs secondes. Ils proposent également plusieurs algorithmes de classification régulièrement utilisés : un modèle de mélanges gaussiens et des variantes de la méthode des plus proches voisins. Dans le cadre de leurs travaux, ils cherchent à obtenir les résultats les plus fiables possibles et réalisent pour cela une batterie de tests. Les meilleurs résultats apparaissent en utilisant simultanément les trois caractéristiques donnant les meilleurs résultats lorsqu'elles sont utilisées individuellement. Les classifieurs, quant à eux, semblent globalement donner la même qualité de résultats.

Une équipe de l'Université de Montréal considère que les méthodes habituelles de classifications ne sont pas compatibles avec un fonctionnement en temps réel à cause du temps de latence nécessaire à certaines caractéristiques. Ils proposent leur solution dans leur papier nommé « Speech/Music Discrimination For Multimedia Applications » [5]. Le travail de cette équipe repose sur la combinaison de fréquences spectrales de lignes (LSF) et du taux de passage par zéro (ZCR), et a pour objectif d'obtenir de bons résultats avec le plus de réactivité possible, c'est-à-dire en classifiant sur des plages glissantes de quelques dizaines de millisecondes seulement. Ils développent pour cela une caractéristique nouvelle basée sur une analyse par prédiction linéaire du taux de passage par zéro qu'ils ont baptisée

« LP-ZCR ». Ils choisissent d'analyser les résultats en utilisant deux classifieurs connus : les k plus proches voisins, et un classifieur quadratique gaussien. Les meilleurs résultats sont obtenus avec la méthode des k plus proches voisins, mais celle-ci nécessite une grande quantité de données d'entraînement. Ces travaux proposent également l'utilisation d'un filtrage post-classification qui permet de lisser les résultats afin d'éviter des changements trop rapides de classe pour être corrects. Cette fonctionnalité donne au système de filtrage la possibilité de changer, pour la trame en cours, la décision du classifieur en fonction des résultats antérieurs. Cela n'affecte pas le fonctionnement temps réel car seuls les résultats précédents interviennent dans la décision.

b. Travaux principalement orientés sur les classifieurs

Le but de la classification est d'interpréter les résultats pour déterminer le type de son de la manière la plus précise possible. La majorité des classifieurs, tels que la méthode des k plus proches voisins ou un modèle de mélanges gaussiens, nécessitent une importante base d'entraînement. Cette base contient des résultats réputés corrects pour les différents types de sons à classifier, mais il existe d'autres classifieurs qui peuvent s'en passer.

Hadi Harb, et Liming Chen, de l'Ecole Centrale de Lyon proposent une solution de classification nouvelle dans leur papier « *Robust Speech Music Discrimination Using Spectrum's First Order Statistics And Neural Networks* » [3]. Celle-ci utilise des statistiques du premier ordre du spectre comme caractéristique et se sert, entre autres, d'un réseau de neurones pour la classification. Les principaux avantages de cette méthode, mis en avant par les auteurs, sont l'obtention de bons résultats avec peu d'entraînement et une adaptabilité aux besoins. Ils relèvent que, dans certains cas, il peut être intéressant de traiter un son correspondant à une personne qui parle sur un fond musical comme étant de la voix, et dans d'autres cas comme étant de la musique, chose que permet de faire leur solution. La classification est réalisée grâce à des modèles de mélange de densité de probabilités couplés à un réseau de neurones chargé d'estimer la présence de ces modèles dans la classe de la voix ou dans celle de la musique.

Actuellement, il semble y avoir peu d'évolution dans le domaine de la classification. La majorité des travaux sont plutôt axés sur la recherche de caractéristiques plus discriminantes et utilisent ensuite des classifieurs reconnus.

c. Les autres travaux

Certains travaux proposent des méthodes autres que le modèle en deux étapes, ou se concentrent à la fois sur la recherche de nouvelles caractéristiques et de nouveaux algorithmes de classification.

Ainsi, une équipe de l'IDIAP en Suisse propose un modèle à 3 étages dans le papier « Robust HMM-Based Speech/Music Segmentation » [6]. Cette solution s'appuie sur un étage de

calcul de probabilités a posteriori de présence de phonèmes grâce à un type de réseaux de neurones très basique en connexionnisme : le perceptron multicouche (MLP). L'étage suivant est chargé de l'extraction de deux caractéristiques : « l'entropie » et le « dynamisme », à partir des résultats générés par le réseau de neurones. La classification, quant à elle, est réalisée dans le troisième étage grâce à un modèle de Markov caché traitant des distributions générées précédemment. Leur méthode semble robuste et permet un traitement avec peu de latence. Il est toutefois nécessaire d'entrainer le classifieur, mais cela est réalisé de manière automatique grâce à l'algorithme de Baum-Welch.

Des chercheurs de l'université de Sheffield en Angleterre et de Berkeley aux Etats-Unis soumettaient en partie la même idée, quelques années auparavant, dans leur papier « Speech/Music Discrimination Based On Posterior Probability Features » [12]. Leur solution utilise un réseau de neurones afin d'estimer a posteriori si chaque partie du son devrait être identifiée comme étant un parmi une cinquantaine de phonèmes reconnus. Les caractéristiques qu'ils retiennent sont quatre statistiques calculées par trames de 100ms : l'entropie moyenne, le dynamisme moyen, l'énergie du fond sonore, et la distribution des phonèmes.

2. La gestion de grandes quantités d'IO sur cluster

Les traitements de classification sont effectués sur des fichiers son non compressés ce qui génère une grande quantité de lectures. Celles-ci peuvent s'effectuer sans problème lorsque le fichier son existe localement sur chaque machine du cluster, mais il est nécessaire de trouver une solution intelligente dans le cas d'un accès via le réseau. En effet, il n'est pas concevable d'espérer obtenir de bonnes performances si un grand nombre de machines accèdent simultanément au fichier à traiter sur le même serveur de fichiers.

Des recherches publiées par l'équipe *KerData* composée de chercheurs de l'antenne bretonne de l'ENS Cachan, de l'INRIA, et de l'université de l'Illinois proposent dans « Damaris : Leveraging Multicore Parallelism to Mask I/O Jitter » [13] une approche consistant à dédier un cœur par nœud au traitement des I/O plutôt que d'effectuer ces opérations directement dans les nombreuses taches de calcul. Les I/O peuvent se faire en parallèle des calculs lorsque l'algorithme le permet. Leurs travaux portent principalement sur l'écriture de données pour permettre la création rapide de points de reprise lors de l'exécution de traitements importants sur de gros clusters. Ils utilisent pour se faire également différents types de systèmes de fichiers parallèles tels que Lustre, GPFS, PVFS.

Deux chercheurs de l'université de l'Illinois présentent dans leur papier « Towards Efficient and Simplified Distributed Data Intensive Computing » [14] une solution différente de l'habituelle association de PFS, ordonnanceur et framework. Leur solution s'apparente à GFS/MapReduce développé en interne par Google mais est plus souple car elle n'est pas limitée à quelques traitements spécifiques. Leur architecture s'appuie sur un système de

stockage extensible qu'ils ont développé et baptisé « Sector » et un framework de traitement de données parallèle qu'ils ont nommé « Sphere ». Le couple Sector/Sphere a spécifiquement été conçu pour être fortement lié. Il permet aux utilisateurs de forcer la position des données au sein du système de stockage s'ils le souhaitent, ou de laisser ce choix à Sector et Sphere. Leur système intègre un ordonnanceur propre, un mécanisme de partage de charge et de tolérance aux pannes. Les chercheurs disent de leur système qu'il est « application-aware » dans le sens où il peut placer les données de manière intelligente en fonction des besoins des différentes applications. Ils le qualifient également de « network-aware » car il a une connaissance de la topologie du réseau et permet de faire des réplications avisées entre sites distants. Techniquement, le matériel nécessaire pour opérer ce système se limite à un certain nombre de serveurs classiques avec des disques internes. L'architecture nécessite un serveur dédié à la sécurité qui sera chargé d'authentifier les utilisateurs et les machines, quelques serveurs « maitres » dont le rôle est de maintenir l'intégrité des données et d'assurer l'ordonnancement des travaux, et des serveurs esclaves qui traitent et stockent les données. Les performances qu'ils obtiennent en traitant de grandes quantités de données sont intéressantes et de l'ordre de 2 à 4 fois plus rapides qu'Hadoop, une implémentation libre de GFS/MapReduce, tout en offrant davantage de fonctionnalités.

Une équipe de chercheurs du MIT propose, quant à elle, un système permettant de distribuer le stockage et la récupération de fichiers sur plusieurs serveurs NFS de manière cohérente. Ils décrivent ce système et son implémentation dans leur papier nommé « MORRIS : A Distributed File Système for Read-Intensive Applications » [15]. L'objectif de MORRIS est de tirer parti au mieux des serveurs de fichiers pour permettre de multiples lectures concurrentes sans perte de performance. Cela est réalisé en éclatant les fichiers en blocs de taille fixe qui seront répartis sur les serveurs de données. L'avantage de leur système est qu'il fonctionne comme un serveur NFS classique du point de vue de l'utilisateur. Cela nécessite toutefois qu'un composant, qu'ils ont baptisé « NFSStripe », soit exécuté sur chaque client. NFSStripe communique ainsi directement avec les serveurs de fichiers qu'ils ont nommés « StripeServer ». Par contre, MORRIS n'offre pas davantage de disponibilité qu'un serveur NFS classique dans le sens où si un des serveurs venait à faillir, l'ensemble du système ne pourrait plus fonctionner. Les performances qu'ils présentent sont effectivement régulières au fur et à mesure que des clients sont ajoutés, mais, pour un faible nombre de clients, elles sont bien inférieures à celles obtenues par un serveur NFS unique (de l'ordre de 3 fois pour un seul client). Ils concluent en disant que leur approche est prometteuse, et, qu'avec davantage de recherche, elle pourrait être en mesure d'être plus performante que les systèmes de fichiers pour lectures intensives existant. Il ne semble toutefois pas avoir eu de suite depuis ce papier paru en 2005.

Le papier « Comparison of Leading Parallel NAS File Systems on Commodity Hardware » [16] résultant de recherches d'une équipe du « Lawrence Livermore National Laboratory » en Californie semble être l'unique étude comparant les systèmes de fichiers distribués GPFS et

Lustre sur le même matériel. L'architecture qu'ils utilisent pour faire leurs tests est composée d'une baie de stockage capable de fournir 2.4Go/s, et de 5 serveurs à base de processeurs Nehalem reliés avec la baie via une connexion InfiniBand. Le cluster jouant le rôle des clients est composé de 1152 nœuds avec une interconnexion « fat tree » InfiniBand. Les performances qu'ils présentent donnent un avantage à GPFS pour tout ce qui est lecture/écriture de fichiers. Dans le cas où un seul nœud écrit un fichier, les performances obtenues sont proches de 1.2GB/s pour GPFS alors qu'elles sont légèrement inférieures à 500Mo/s pour Lustre. De la même manière, pour des écritures dans un fichier partagé par plusieurs nœuds, GPFS arrive à saturer la bande passante de la baie disque (2.4Go/s) à partir de 4 clients simultanés, alors que Lustre semble atteindre son maximum à partir de 64 nœuds en offrant tout juste 2Go/s. Lustre offre les meilleures performances pour tout ce qui a trait à des opérations sur les métadonnées des fichiers ou des répertoires (création de fichier, de répertoires, suppression, lecture d'informations telles que la taille, la date de création,…), mais les chercheurs conviennent qu'il s'agit là d'opérations relativement rares dans le traitement de données. En plus des mesures de performances, ils ont réalisé des essais en cas de pannes et concluaient que GPFS avait passé ces tests sans problèmes, alors qu'ils ont rencontré des difficultés avec Lustre. D'autre part, ils concluent leur papier en indiquant que les outils livrés par IBM pour l'administration de GPFS sont très aboutis et simplifient grandement toute les opérations nécessaires au quotidien dans de gros environnement de production.

3. Bilan

Les différents travaux sur les classifieurs de sources sonores et surtout la quantité différente de méthodes d'extraction de caractéristiques et de classification permettent de mieux apprécier le fait que l'évolutivité du programme est un aspect très important de ce projet. Il est primordial de permettre facilement d'ajouter, de modifier, ou de supprimer des fonctionnalités ou des algorithmes de traitement.

Les recherches publiées par l'équipe *KerData* sur l'optimisation des I/O sur cluster nous confortent dans l'idée que la spécialisation de certains processus dans la gestion des I/O est une idée intéressante. Dans notre cas, puisque nous ne disposons pas d'une architecture proposant un système de fichier parallèle, il est impensable de dédier un processus par nœud à ces taches, car cela mènerait dans le cas du cluster Intercell de Supélec avec ses 256 machines, à 256 lectures simultanées sur NFS avec un système de fichiers classique. Les processus spécialisés dans les I/O seront donc partagés par un ensemble de nœuds afin de limiter le nombre d'accès simultanés à NFS. La recherche de la bonne granularité des I/O et des configurations permettant d'offrir de bonnes performances constituera une part importante de ce travail de mémoire d'Ingénieur.

Chapitre 3

Reconception « objet » et optimisations des traitements

Avant d'aborder les optimisations en matière de vitesse, il était important de revoir l'utilisation mémoire de l'outil. En effet, celui-ci allouait dynamiquement une zone mémoire contiguë dont la taille permettait de stocker l'ensemble de la piste à traiter. Cette méthode, simple à mettre en œuvre, génère plusieurs problèmes comme la possibilité de saturation de la mémoire de la machine, et donc une limite de la taille maximum du fichier traitable fortement liée aux caractéristiques de la machine.

1. Reconception « Objet »

La modularité et la facilité de maintenance sont des aspects importants dans ce projet. Ils ont été pris en compte en redéveloppant une partie importante de l'outil en « technologie objet ». Cela permet de regrouper au maximum des outils communs à l'ensemble des caractéristiques dans une classe mère (*Feature*) dont hérite une classe fille composée seulement des algorithmes propres au traitement de sa caractéristique. L'utilisation d'objets permet de nous assurer de l'initialisation de l'ensemble des variables de l'objet dans le constructeur de celui-ci. Cela permet également de créer une classe de gestion de tableaux à une ou deux dimensions (*Array2D*) détruisant automatiquement le tableau et libérant ainsi la mémoire dès la fin de sa portée.

a. Notre classe Array2D

Son but est de simplifier l'allocation mémoire de tableaux à deux dimensions, de vérifier, en phase de développement, si le programme ne tente pas d'accéder à une zone mémoire non allouée en faisant des tests supplémentaires sur les index, mais aussi d'assurer avec certitude que la mémoire est bien libérée quand le tableau n'est plus utilisé.

Si la macro *DEBUG* est définie, on réalise une série de tests lors d'accès à un index pour vérifier que le tableau a bien été initialisé et que l'index accédé est bien compris dans l'intervalle autorisé. En production, la macro *DEBUG* n'est pas définie ce qui empêche la compilation de ces tests pour ne pas pénaliser le temps de traitement.

```
#define DEBUG

#include <iostream>
#include <string>
#include <Array2D.h>


int main()
{
 Array2D <double> Test;

 if(!Test.Is_Initialized())
 {
  std::cout<<"Not Initialized - 1"<< std::endl;
 }

 Test.Initialize(4,2);

 if(!Test.Is_Initialized())
 {
  std::cout<<"Not Initialized - 2"<< std::endl;
 }

 Test.index(0,0)=85.5;
 Test.index(1,0)=15.3;
 Test.index(2,0)=12.0;
 Test.index(3,0)=32.7;

 for(unsigned int i=0;i<=4;i++)
 {
  Test.index(i,1)=Test.index(i,0)*Test.index(i,0);
  std::cout << Test.index(i,1) << std::endl;
 }

 return 0;
}
```

Un petit bug s'est glissé dans le code, dans la boucle nous tentons d'accéder à l'index 4, qui correspondrait à la 5e case du tableau.

Le tableau est libéré immédiatement et automatiquement dès qu'il n'est plus dans la portée.

Exemple de fonctionnement de notre classe Array2D.

On peut considérer le code précédent comme un exemple simple d'utilisation permettant de voir rapidement les possibilités offertes par la classe. La trace d'exécution qui suit illustre le fonctionnement de notre classe Array2D.

```
-bash-3.2$ ./TestArray2D
Not Initialized – 1
7310.25
234.09
144
1069.29
TestArray2D: ./Array2D.h:96: T& Array2D<T>::index(unsigned int,
unsigned int) [with T = double]: Assertion `x >= 0 && x < this-
>size_x' failed.
Abandon
-bash-3.2$
```

Il est facile de vérifier si le tableau est déjà initialisé ou non.

Le programme nous informe lors de la 5e itération qu'il y a une tentative de dépassement de la zone mémoire allouée.

Trace d'exécution du programme précédent.

Cette classe permet d'offrir une certaine souplesse lors du développement en s'affranchissant d'une grande partie de la gestion de la mémoire et permet de mettre en évidence certaines erreurs dans le code, telles qu'un accès à une zone mémoire non allouée. Elle utilise également des templates ce qui la rend compatible avec n'importe quel type de données.

b. Les classes de traitement des caractéristiques

L'intérêt de ces classes est d'avoir le code le plus générique possible pour rajouter facilement de nouvelles caractéristiques. Cela permet également de rendre le code source beaucoup plus lisible en uniformisant les noms des fonctions des caractéristiques et en regroupant les algorithmes propres à chacune d'elles.

Les classes de traitement sont au nombre de quatre et reprennent les algorithmes utilisés par le programme original. Elles héritent de la même classe mère contenant le code commun. Elles s'appuient également sur les fonctionnalités offertes par *Array2D* pour le stockage des données intermédiaires, et celles d'une classe nommée *Training* permettant d'utiliser facilement les données d'entrainement qui sont propres aux caractéristiques extraites et traitées par ces classes.

Les fonctions communes car héritées de la classe mère sont les suivantes :

```
void Set_b_Compute_Mean(bool b);
void Set_b_Compute_Variance(bool b);
```

Elles permettent de spécifier s'il faut calculer la moyenne ou la variance lors du calcul des segments.

```
bool Get_b_Compute_Mean();
bool Get_b_Compute_Variance();
```

Elles permettent de connaitre s'il faut ou non calculer la moyenne ou la variance lors du calcul des segments.

```
void Initialize_LTFeatures(unsigned int Data_Size);
```

Elle permet de spécifier le nombre maximum de segments (caractéristiques à long terme) calculés par un nœud, en un cycle. Cette fonction initialise un tableau *Array2D* contentant les résultats des segments ainsi calculés.

```
void Initialize(unsigned int Sizex);
```

Elle permet de spécifier le nombre maximum de trames (caractéristiques à court terme) calculées par un nœud, en un cycle. Cette fonction initialise un tableau *Array2D* contentant les résultats des trames ainsi calculées.

```
void Save_Mean(string path, string filename, char *ext);
void Save_Variance(string path, string filename, char *ext);
```

Ces fonctions permettent de sauvegarder les résultats des calculs de segments, pour la moyenne ou la variance.

```
void Load_Mean_Training_Files(string speech, string music, string
ext, int *n_pts_res);
void Load_Variance_Training_Files(string speech, string music, string
ext, int *n_pts_res);
```

Ces fonctions permettent de charger les données d'entrainement nécessaires à la classification pour la moyenne ou la variance.

```
void Normalize_All();
```

Cette fonction normalise les résultats des segments, à la fois pour la moyenne et la variance.

```
double Mean_Distance_Points(int segment, int training, int source);
double Variance_Distance_Points(int segment, int training, int
source);
```

Ces fonctions permettent de calculer la distance entre un segment calculé et un segment d'entrainement, pour la moyenne ou la variance.

Une seule fonction de l'interface n'est pas héritée de la classe mère, il s'agit de :

```
void Compute(...);
```

Cette fonction est la seule propre à chaque caractéristique, elle peut prendre des arguments différents et contient l'algorithme permettant de calculer les trames pour la caractéristique en question.

c. Architecture logicielle développée

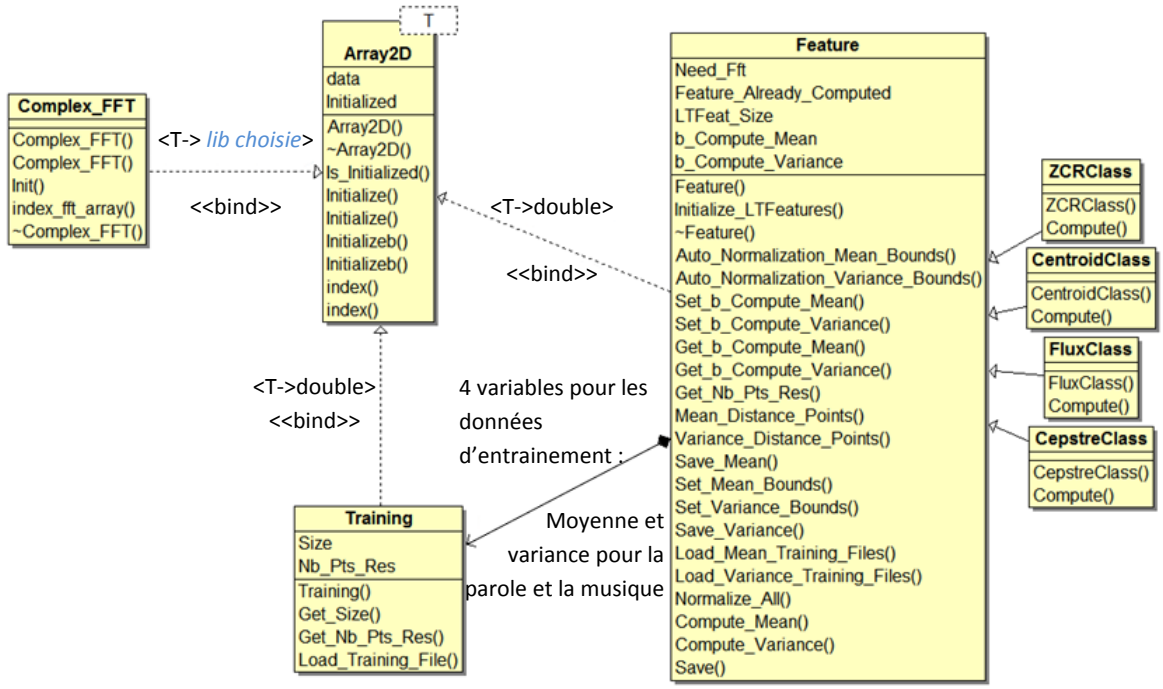


Figure 3.1 : diagramme de classe de la nouvelle architecture logicielle.

L'extraction des trames est effectuée uniquement grâce aux fonctions *Compute()* fournies par chacune des classes de caractéristiques. Le calcul des segments est réalisé en utilisant principalement les fonctions *Compute_Mean()* et *Compute_Variance()* héritées de la classe mère *Feature*. La classification, quant à elle, est faite en utilisant les fonctions fournies par la classe Training qui permettent de charger les données d'entrainement correspondant à la variance et à la moyenne de la voix et de la musique pour chaque caractéristique. Les fonctions *Mean_Distance_Points()* et *Variance_Distance_Points()* de la classe mère permettent de calculer les plus proches voisins lors de la phase de classification.

La classe *Complex_FFT,* quant à elle, propose un tableau dynamique Array2D dont le type dépend de la bibliothèque de calcul de transformées de Fourrier choisie afin de permettre le calcul des trames pour les caractéristiques qui ont besoin de la FFT.

2. Optimisations des traitements

a. L'utilisation de la mémoire

Les problèmes liés à la lecture du fichier audio en plusieurs fois proviennent du taux de recouvrement des trames, mais aussi du fait que, pour calculer le flux spectral, il est nécessaire de connaitre un certain nombre d'échantillons antérieurs au premier échantillon de la trame à traiter.

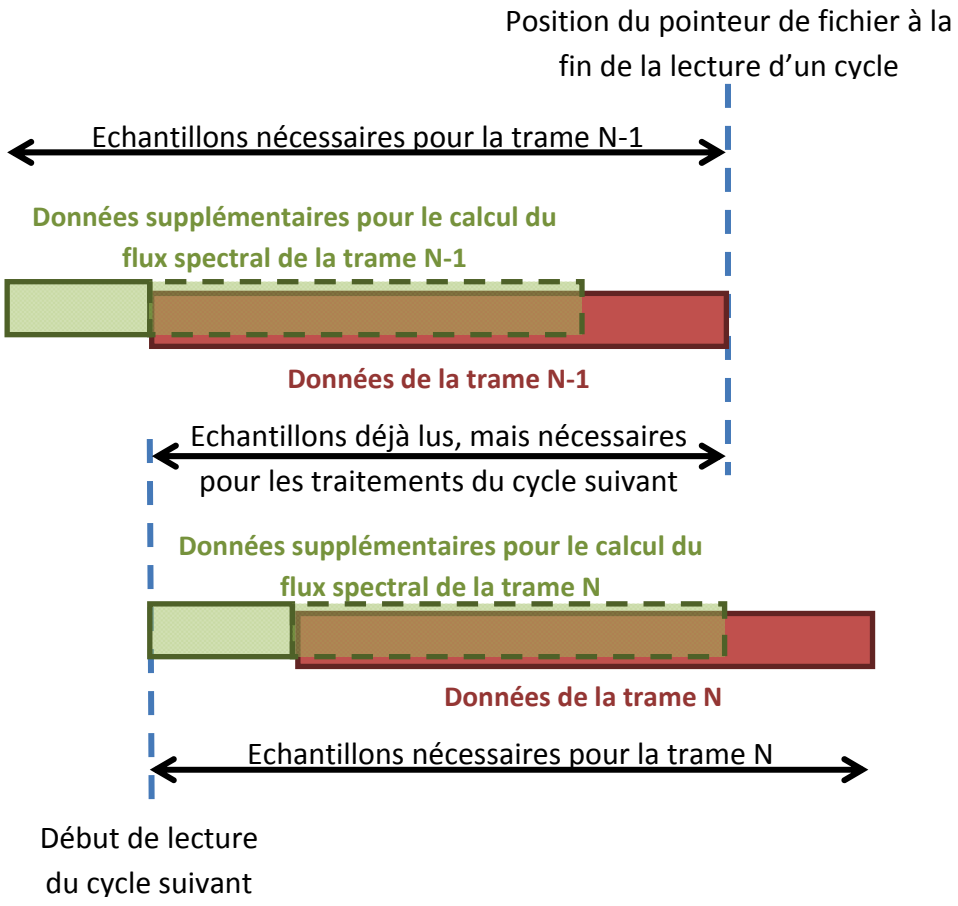


Figure 3.2 : représentation visuelle de la nécessité du replacement du pointeur de fichier.

Il est donc essentiel de sauvegarder temporairement une quantité de valeurs déjà lues pour calculer les premières trames du cycle suivant (voir Figure 1.4), ou de replacer le pointeur de fichier.

Ayant en tête la future distribution de l'outil, la solution consistant à repositionner le pointeur de fichier sur le début de la trame suivante apparait comme étant la plus judicieuse (voir page 38), d'autant plus que des tests effectués spécialement dans ce but ne permettent pas de mettre en évidence une différence de performance mesurable entre ces deux méthodes.

Nous pouvons supposer à présent que le volume lu à chaque cycle a un impact sur les performances globales de l'outil, toutefois, en pratique, nous nous apercevons que la plage pour laquelle les performances sont semblables est très large.

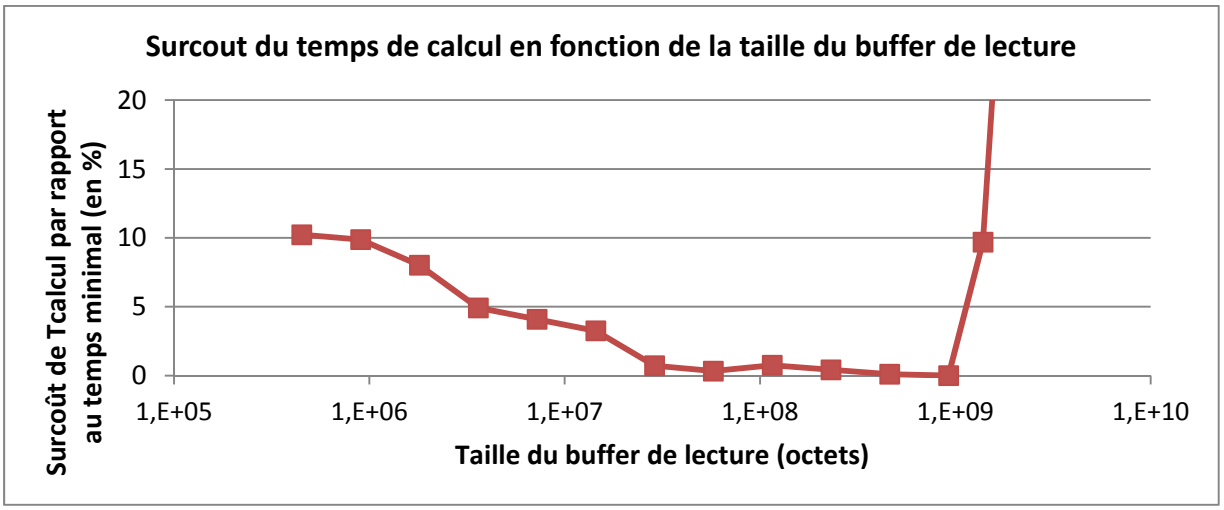


Figure 3.3 : surcout du temps de calcul en fonction de la taille du buffer de lecture.

Ainsi sur la machine utilisée lors du développement (Intel Core(TM) i7 870 @ 2.93GHz avec 6 Go de RAM), la taille de buffer possible pour un surcout de temps de calcul inférieur à 1% par rapport au meilleur temps obtenu est comprise entre environ 25 et 900 Mo, ce qui autorise une certaine souplesse de fonctionnement.

On choisira pour l'instant arbitrairement un buffer d'environ 84Mo, permettant d'accueillir de quoi calculer exactement 50000 trames.

En analysant l'exécution de l'outil avec Valgrind[4], un certain nombre d'autres problèmes liés à la mémoire ont été découverts dans le logiciel original, tels que des oublis de libération de zones mémoire allouées dynamiquement, ou encore l'accès à des zones mémoires incohérentes à cause d'un calcul d'index erronés dans certains cas. Ce bug avait d'ailleurs un impact sur les résultats obtenus, et après analyse avec le responsable de l'application, les nouveaux résultats furent considérés comme corrects et utilisés comme référence lors de tests de non-régression dans la suite du développement.

Un problème entrainant le crash de l'application, sous Windows, dans le cas d'un fonctionnement utilisant un seul thread OpenMP a également été isolé et corrigé. Il était lié à des comparaisons effectuées sur certaines variables ayant trait à la gestion des threads avant leur initialisation.

Les problèmes de gestion de la mémoire étant réglés, il restait deux travaux importants à réaliser en séquentiel : permettre une simplification d'ajout de caractéristiques et de traitements, et optimiser le code pour une exécution plus rapide.

[4] http://valgrind.org/docs/manual/mc-manual.html

b. Utilisation d'une bibliothèque rapide pour le calcul de la FFT

L'outil original utilise un code propre pour le calcul des transformées de Fourrier rapides. Un rapide profilage indique que la majeure partie du temps de traitement d'un fichier audio est passée à calculer les FFT. Malgré tout, le code utilisé pour ces calculs ne semble pas grandement optimisable, d'où l'idée d'utiliser une bibliothèque de calcul scientifique réputée pour ces performances : FFTW[5]. Cette bibliothèque tire parti au mieux de l'architecture des machines en utilisant, si disponible, des jeux d'instructions tels que SSE ou SSE2.

Un benchmark de performance entre le code original et FFTW montre que, pour un grand nombre de calculs de FFT successifs, FFTW est environ 70% plus rapide sur la machine de développement, ce qui correspond à une accélération de 3.33x.

Malheureusement, l'intégration de FFTW dans le programme n'est pas chose aisée, car les types de données utilisés sont propres à la bibliothèque. De plus, une consigne supplémentaire fut de pouvoir choisir, à la compilation, s'il faut utiliser FFTW ou le code d'origine.

L'idée pour offrir ce choix est de créer une classe permettant de mettre en forme les données pour la bibliothèque qui a été choisie à la compilation. Cette solution consiste en deux macros d'accès aux données ayant les mêmes arguments et renvoyant les valeurs pertinentes quel que soit le choix qui a été fait, et un *typedef* de nom identique correspondant au type utilisé par la bibliothèque sélectionnée.

Un avantage supplémentaire de cette méthode est lié au fait que FFTW requiert l'utilisation de méthodes d'allocation mémoire livrées avec la bibliothèque. Avec l'utilisation de la classe permettant l'interface, on peut facilement réaliser l'allocation dans le constructeur et la libération dans le destructeur et ne plus se soucier de la gestion de la mémoire par la suite.

Le code d'interfaçage est situé dans un fichier .h et se présente de la manière suivante :

```
#ifdef FFTW
    Code nécessaire au fonctionnement avec FFTW
#else
    Code nécessaire au fonctionnement avec le code original
    pour le calcul de la FFT
#endif
```

[5] http://www.fftw.org/

Si la macro FFTW est définie alors le code nécessaire à FFTW est compilé, sinon c'est le code original qui est compilé et utilisé.

Dans l'exemple suivant, nous allons voir le code nécessaire à l'utilisation de FFTW :

Ici, *fftw_complex* correspond au type utilisé par la bibliothèque FFTW mais *TComplex* sera à chaque fois déclaré comme étant le type utilisé par la bibliothèque choisie.

```
typedef fftw_complex TComplex;
```

Les deux macros suivantes permettent d'accéder à la partie réelle et imaginaire d'un élément dans le tableau des données nécessaires au calcul de la FFT avec les données au format supporté par FFTW.

```
#define _access_tcomplex_r(i,a) (a[i][0])
#define _access_tcomplex_i(i,a) (a[i][1])
```

La classe en elle-même permet de réaliser l'allocation et la libération de la mémoire, et de retourner un pointeur vers les données nécessaires au calcul de la FFT par un thread.

```
class Complex_FFT : Array2D <fftw_complex *> {

unsigned int NbTh,Size_FFT;

public:
Complex_FFT() {                                    Initialisation et allocation mémoire
    this->NbTh=0;
    this->Size_FFT=0;
}

void Init(unsigned int size_fft,unsigned int NbTh) {
    this->NbTh=NbTh;
    this->Size_FFT=size_fft;
    this->Initialize(NbTh);

    for(unsigned int i=0;i<NbTh;i++)
    {
    this->data[i]=(fftw_complex *)
                  fftw_malloc(sizeof(fftw_complex)*size_fft);
    }
}

inline fftw_complex * index_fft_array(unsigned int T) {
    return(this->data[T]);
}                  Renvoi du pointeur vers les données à traiter par un thread
```

```
~Complex_FFT()                                          Libération de la mémoire
{
    for(unsigned int i=0;i<this->NbTh;i++)
    {
    fftw_free(this->data[i]);
    // data itself is deleted in Array2D's destructor
    }
}
};
```

Malheureusement, tout cela ne permet pas d'uniformiser complètement le code pour le calcul des FFT dans le source principal. Il est tout de même nécessaire d'initialiser quelques pointeurs et variables locales en fonction de la bibliothèque choisie et d'exécuter les fonctions différemment. Il s'agit en l'occurrence de deux lignes de code encastrées dans des macros #ifdef #else #endif. Malgré tout, l'ensemble de l'accès aux données pour les FFT est généralisé ce qui permet de limiter de manière drastique le nombre de lignes de codes dédoublées.

c. Parallélisation multi-cœur en OpenMP

OpenMP est un standard développé par un consortium industriel et académique qui définit un jeu de directives et de fonctions permettant de faire de la parallélisation sur une machine de type MIMD à mémoire partagée, c'est-à-dire avec plusieurs processeurs ou un processeur multi-cœur partageant la mémoire centrale [11]. Avant OpenMP, les constructeurs tels que CRAY, ou IBM ne proposaient que des interfaces propriétaires pour la parallélisation. Les programmes ainsi développés n'étaient pas portables et imposaient l'utilisation du compilateur fourni par le constructeur. Aujourd'hui, des implémentations d'OpenMP sont disponibles pour de nombreux compilateurs sur la majorité des architectures et systèmes d'exploitation.

Ses principaux atouts sont :

- la portabilité du code ;
- la simplicité de mise en œuvre pour paralléliser des boucles de calcul indépendantes;
- la possibilité de paralléliser le code graduellement.

La majorité de la parallélisation est effectuée grâce à des directives de compilation, mais il existe un certain nombre de fonctions permettant, entre autres, d'utiliser des verrous, d'intervenir sur le nombre de threads, etc. Ces fonctions sont essentiellement utilisées pour la parallélisation de calculs non indépendants et donc plus complexes à mettre en œuvre.

Les principales fonctions et directives utilisées pour la parallélisation de ce programme sont les suivantes :

```
int omp_get_num_procs()
```

Retourne le nombre de cœurs logiques dans la machine.

```
void omp_set_num_threads(int num_threads)
```

Permet de fixer le nombre de threads à créer.

```
int omp_get_thread_num();
```

Renvoie le numéro du thread en cours d'exécution.

```
#pragma omp parallel for private(idx,a,b,c)
```

Indique au compilateur que la boucle « for » directement après est parallèle et que les variables idx, a, b et c sont propres à chaque thread. Cela permet de gérer correctement l'index de la boucle et d'être sûr que les calculs intermédiaires d'un thread n'impactent pas les résultats d'un autre.

Schématiquement, nous pouvons représenter le programme original de cette manière :

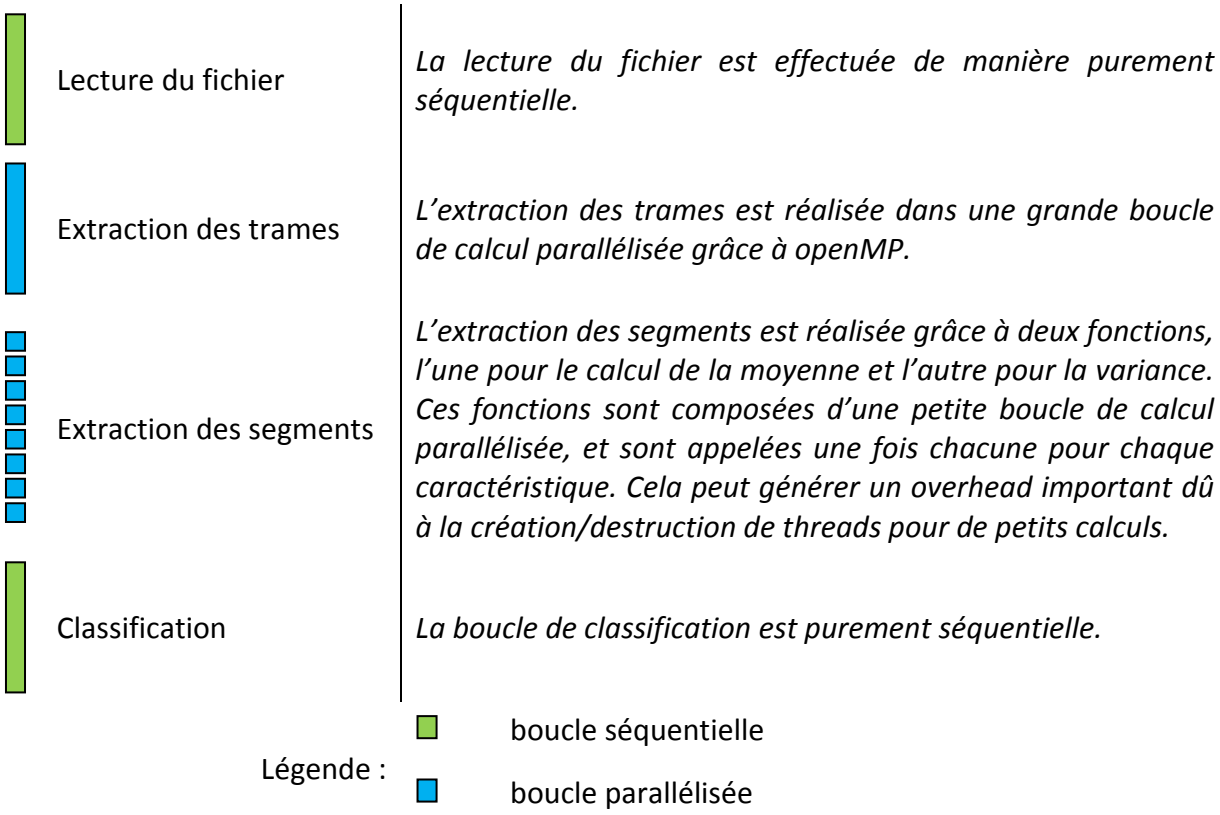


Figure 3.4 : représentation du programme original.

Une grande partie du programme original était déjà parallélisée avec OpenMP, mais l'étape de classification n'était exécutée que sur un cœur. Afin d'optimiser au maximum l'utilisation

des ressources, il était nécessaire de paralléliser cette boucle de calcul également. Cette démarche fut l'occasion de revoir intégralement la parallélisation implantée.

Schématiquement, la version optimisée peut être représentée de la manière suivante :

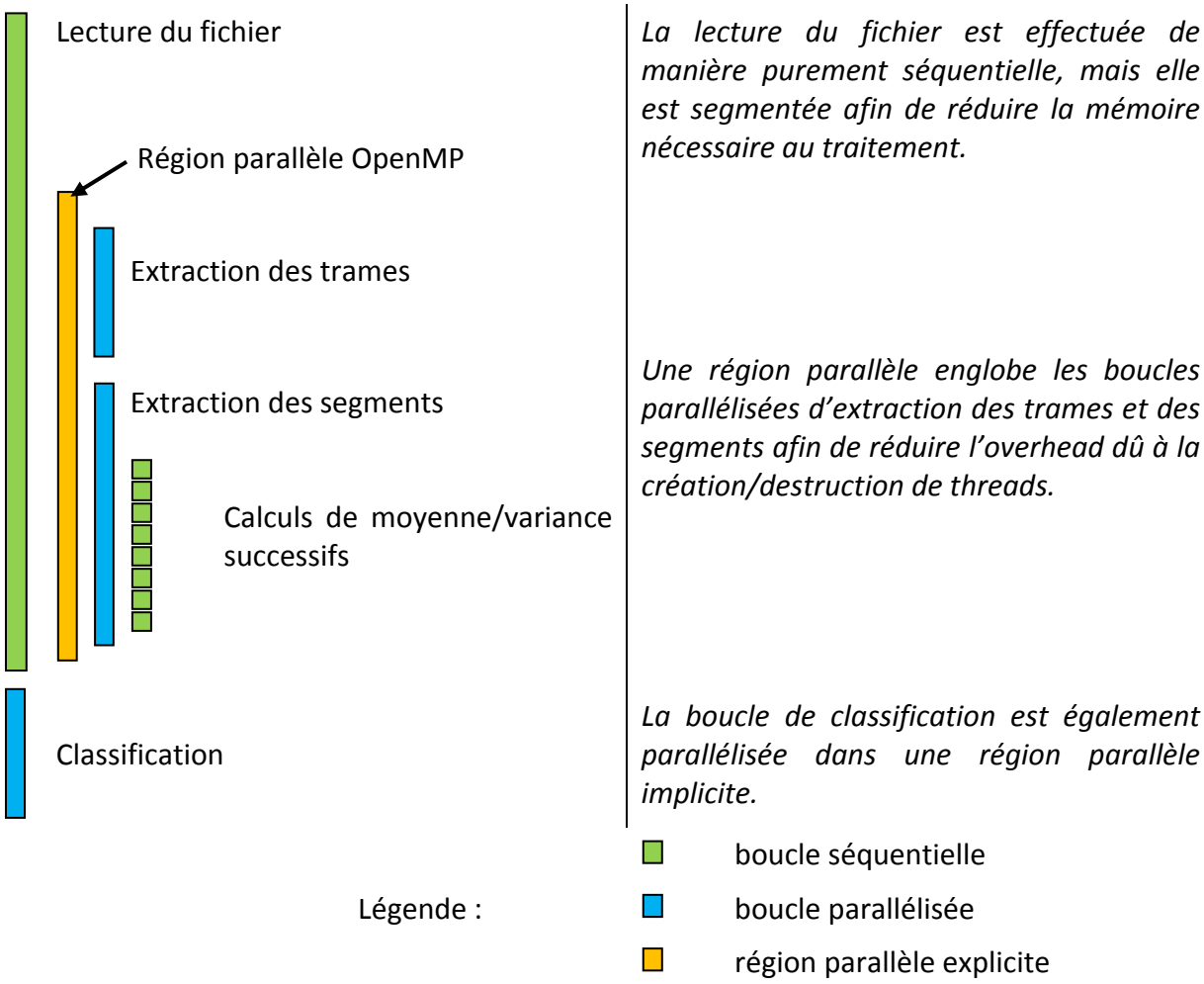


Figure 3.5 : représentation de la version optimisée du programme.

Il nous a été possible d'améliorer les performances en parallélisant l'intégralité de la boucle de calcul des segments et en rendant les petites boucles de calcul de la moyenne et de la variance séquentielles.

d. Optimisations diverses

D'autres optimisations s'avérèrent possibles en factorisant certains calculs afin d'éviter de calculer deux fois de suite les cosinus ou sinus d'une même valeur dans la même ligne. Cela permit de réduire la durée de ce calcul, réalisé un grand nombre de fois dans une boucle, d'environ 20%.

Compiler avec *fast-math* permet de réduire d'avantage encore le temps d'exécution du programme. Il s'agit malgré tout d'une option à utiliser avec précaution, car, même si pour

l'ensemble des tests effectués les résultats étaient identiques à ceux obtenus sans fast-math, la qualité des résultats obtenus n'est pas garantie.

En fait, *fast-math* active *unsafe-math-optimizations* qui permet des optimisations qui peuvent violer les standards IEEE et ANSI. *Unsafe-math-optimizations* active elle-même d'autres optimisations telles que *reciprocal-math* qui optimise les divisions dans certains cas, mais peut aussi réduire la précision du résultat obtenu [17]. Il s'agit d'une option intéressante permettant d'augmenter légèrement la rapidité du programme, mais il est nécessaire de garder ses éventuels inconvénients à l'esprit. Dans notre cas, des essais ont été réalisés sur plusieurs fichier audio et les résultats obtenus étaient exactement les mêmes avec ou sans *fast-math*. Le choix a donc été fait de compiler le programme avec cette option.

Chapitre 4

Conception d'une solution distribuée sur PCs multi-cœur

Distribuer les calculs sur un grand nombre de machines est l'étape logique suivante pour diminuer les temps de traitement. Plutôt que d'effectuer les calculs sur un seul ordinateur, la distribution permet de faire qu'un grand nombre de machines traitent simultanément une partie différente du fichier à analyser. Dans une version idéale de la distribution, le temps de traitement serait divisé par le nombre d'ordinateurs travaillant sur le problème.

1. Architecture et fonctionnement général

Il est nécessaire de concevoir la distribution en ayant à l'esprit les trois scénarios d'utilisation qui ont été identifiés pour les différents utilisateurs types : le développeur chargé de l'évolution, l'exploitant traitant de nouveaux fichiers audio, et l'expert en calcul parallèle soucieux d'identifier la configuration la plus efficace.

Le développeur doit pouvoir effectuer rapidement des essais sur quelques fichiers de test. Ceux-ci peuvent être copiés sur tous les nœuds du cluster pour éviter de charger le réseau lors de ces essais. L'exploitant, quant à lui, ne traitera chaque fichier qu'une seule fois. Il semble donc plus intéressant de s'affranchir de la phase de copie du fichier audio sur chaque nœud en traitant celui-ci à partir d'un espace de stockage centralisé. De plus, cette personne n'est pas censée connaitre en détail le mode de fonctionnement de l'application et s'en remet donc à l'outil pour obtenir une configuration donnant de bonnes performances. L'expert en calcul parallèle préférera ajuster manuellement tous les paramètres pour affiner la bonne configuration dans des conditions d'exécution particulières.

Les fonctionnalités répondant à ces différents scénarios d'utilisation sont donc :

- La lecture des fichiers audio à traiter directement à partir des disques durs locaux de chaque nœud de calcul (fichier dupliqué sur toutes les machines) ;
- La lecture des fichiers à traiter à partir d'un espace de stockage centralisé comme NFS par exemple ;
- La possibilité d'exécuter l'outil avec des connaissances minimes de son fonctionnement et obtenir malgré tout de bonnes performances ;
- La possibilité d'ajuster manuellement l'ensemble des paramètres de distribution.

Le programme doit fonctionner sur n'importe quel type de cluster de PCs, quelle que soit la configuration des machines (nombre de processeurs, nombre de cœurs, quantité de RAM, …). Il est également nécessaire, lors d'une lecture sur un espace de stockage centralisé, de limiter le nombre de nœuds accédant simultanément au fichier afin de ne pas surcharger le serveur qui risquerait de ne pas le supporter.

L'idée de spécialiser des processus pour faire uniquement des entrées/sorties lors de la phase d'extraction des caractéristiques est donc née. Le principe repose sur une notion de groupes, composés de nœuds exécutant des processus. Dans un groupe, sur un des nœuds, il y a un processus « *reader* » chargé de lire le fichier audio et d'attribuer des travaux d'extraction de caractéristiques aux autres processus composant le groupe. Les autres processus du groupe sont les « *worker* » dont l'objectif est de traiter les données reçues par leur *reader*. De nos jours, le faible coût des machines permet de faire évoluer ou de remplacer un cluster de PCs en une fois. Les clusters sont donc généralement homogènes. Cela nous permet de travailler naturellement avec des groupes homogènes qui sont plus faciles à gérer. Les groupes doivent donc être composés du même nombre de nœuds, et les processus répartis de la même manière. L'inconvénient de cette solution qui est relativement simple à mettre en œuvre est que le temps de traitement est lié aux performances du nœud le plus lent car tous ont la même quantité de données à traiter.

Schématiquement, le principe de fonctionnement peut être représenté de la manière suivante :

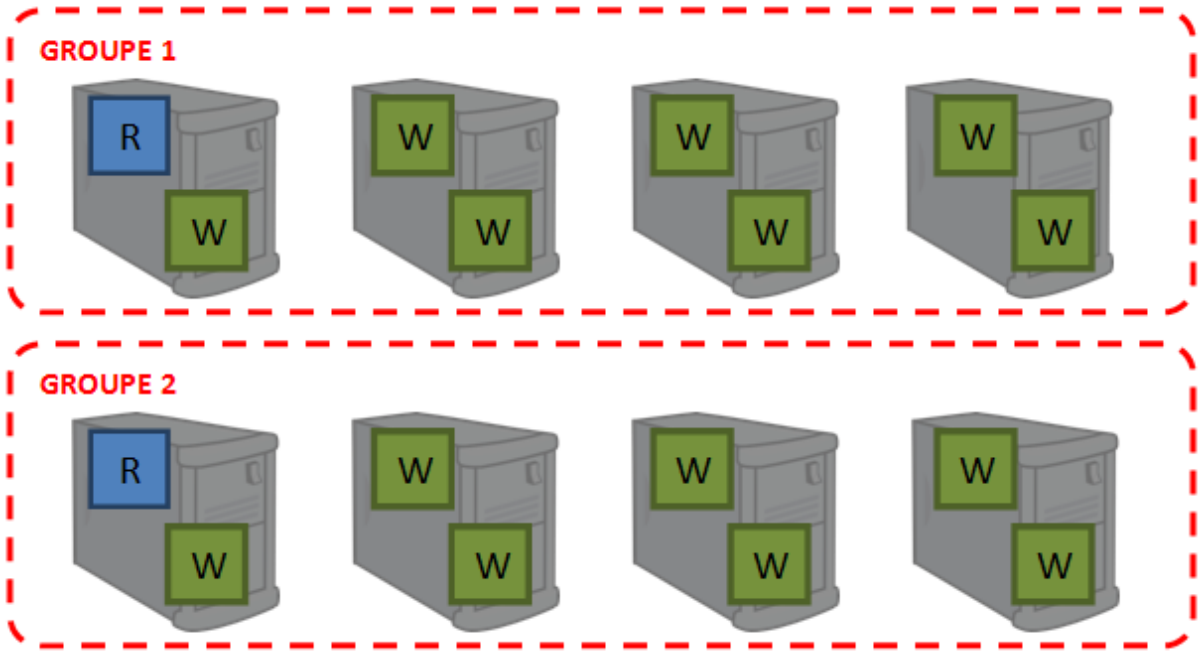


Figure 4.1 : représentation de l'exécution distribuée sur 8 nœuds avec 2 *reader*.

Dans cet exemple, nous disposons de huit nœuds qui supportent deux groupes. Chaque nœud exécute deux processus, soit sept processus *worker* par groupe, plus un *reader*.

Chaque processus *worker* exécute un ou plusieurs threads afin d'exploiter au mieux les capacités des machines.

Le processus *reader* de chaque groupe attribue et communique des morceaux du fichier à traiter à chacun de ses *worker*. Ceux-ci calculent dans un premier temps les trames, puis les segments à partir des trames précédemment calculées. Chaque *worker* renvoie ensuite l'ensemble des segments ainsi calculés au *reader* du premier groupe qui est chargé de faire la concentration et éventuellement la normalisation de tous les segments une fois l'ensemble du fichier traité. Celui-ci finit enfin par distribuer les segments à l'ensemble des processus *reader* et *worker* confondus afin de réaliser la dernière étape du traitement : la classification.

2. L'algorithme distribué

Pour la partie algorithmique, l'abstraction de la partie architecture est importante, car le programme doit fonctionner, plus ou moins rapidement, quelle que soit la distribution des processus sur les différents nœuds. Les groupes sont homogènes et seront donc représentés comme étant composés d'un processus *reader* et d'au moins un processus *worker*. L'algorithme parallèle est donc axé principalement sur des communications interprocessus et peut être représenté en une succession d'étapes identifiées par les communications nécessaires.

L'exemple ci-après montre les communications nécessaires au fonctionnement du programme pour deux groupes de lectures composés chacun de trois processus « *worker* ». Mais cela fonctionne de la même manière quel que soit le nombre de groupes, ou le nombre de processus *worker* par groupe.

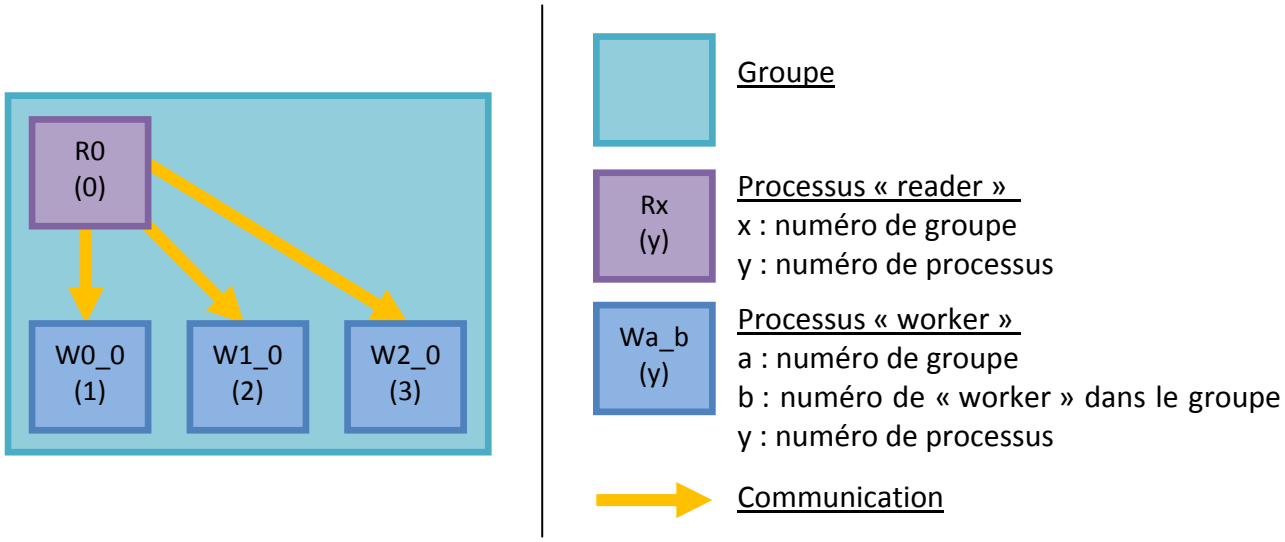


Figure 4.2 : légende pour la schématisation de la distribution.

Etape 1 : Lecture, assignation de travaux par les *reader*, calcul des trames par les *worker*

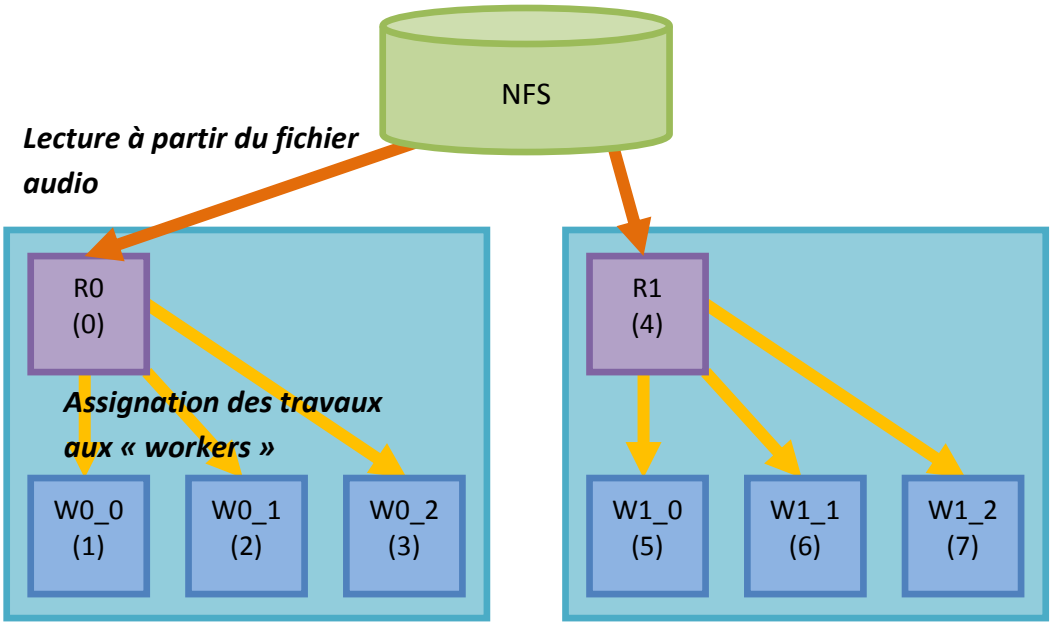


Figure 4.3 : étape 1 ➔ lecture, assignation de travaux de calcul des trames.

Les processus *reader* lisent assez d'échantillons pour permettre à leurs *worker* de calculer un nombre fixe de trames établi à l'initialisation du programme et dépendant de facteurs tels que le taux d'échantillonnage et la taille du fichier. Nous considérons dans cet exemple que chaque *worker* calcule 1000 trames par cycle :

R0 lit et envoie les échantillons nécessaires pour que W0_0 calcule les trames 1 à 1000 en même temps que R1 lit et envoie les échantillons pour que W1_0 traite les trames 1001 à 2000. On note que deux *reader* ne soumettent jamais deux fois les mêmes trames. Les données sont envoyées en utilisant les communications non bloquantes de MPI pour gagner du temps en permettant aux *worker* de recevoir les prochaines données à traiter en parallèle de leur tâche de calcul. Il semble toutefois que ces mécanismes non bloquants soient peu performant et ne permettent pas le recouvrement des calculs et des communications (voir page 44). La position sur laquelle doit se placer un *reader* dans le fichier est facile à calculer, malgré le chevauchement des trames et le décalage nécessaire au flux spectral (voir page 27). Il y a ainsi un petit nombre d'échantillons lus deux fois. Avec les valeurs que nous utilisons pour le chevauchement des trames et pour le décalage nécessaire au calcul du flux spectral, cela correspond environ au nombre d'échantillons nécessaires au calcul d'une trame (soit moins de 4ko).

Le schéma suivant montre la distribution des données identiques (la représentation n'est pas à l'échelle). Chaque *worker* se voit attribuer exactement les échantillons qui lui sont nécessaires pour calculer ses trames. En fonctionnement, aucun échantillon n'est en trop et aucune trame n'est calculée deux fois.

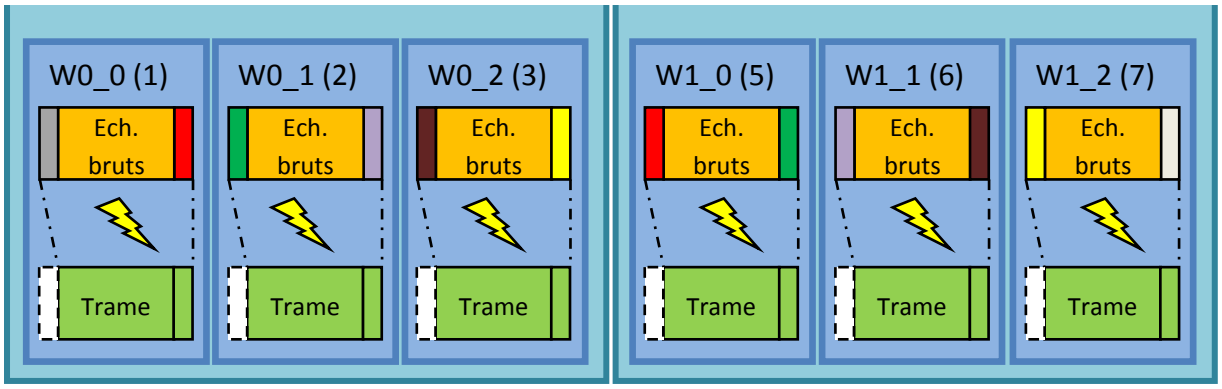


Figure 4.4 : représentation de la répartition des échantillons identiques.

La dernière trame calculée par un *worker* sera celle directement avant la première trame calculée par le *worker* de même rang appartenant au *reader* suivant (ou du *worker* de rang supérieur du *reader* 0 pour les *worker* du dernier groupe). Ainsi la dernière trame du *worker* 2 du groupe 0 est en réalité la trame qui précède la première trame calculée par le *worker* 2 du groupe 1.

Ce choix a été fait dans cette première version distribuée par souci de simplicité. Chaque *reader* progresse dans le fichier en suivant une carte (voir page 43) établie à l'initialisation pour tous les processus. La progression dans le fichier est réalisée de manière entrelacée. Les premières données sont traitées par le *worker* W0_0, les suivantes par W1_0, celles d'après par W0_1, etc. Le schéma suivant illustre la progression du traitement d'un fichier audio :

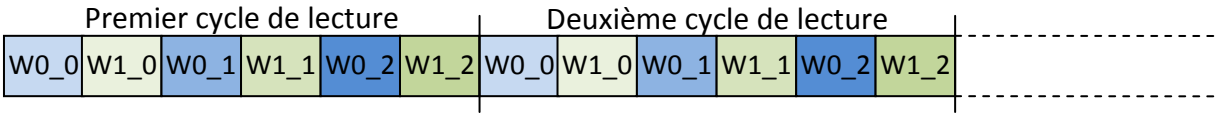


Figure 4.5 : progression entrelacée du traitement.

Nous avions le choix entre plusieurs solutions différentes, mais, faute de savoir à l'avance la solution la plus pertinente en termes de performance, il était nécessaire de faire un choix arbitraire. Cette solution semblait la plus simple à mettre en œuvre car il est possible d'identifier facilement les *worker* à qui il faut envoyer une partie des trames calculées et ceux dont on doit en recevoir. Elle permet de ce fait un fonctionnement avec lequel chaque trame n'est traitée qu'une seule fois.

Etape 2 : Partage des trames nécessaires au calcul des segments par les *worker*

Les segments se chevauchant également à hauteur de 75% (voir page 8). Chaque *worker* doit donc transmettre les trames nécessaires à celui chargé de traiter les données suivantes pour qu'il puisse calculer ses premiers segments.

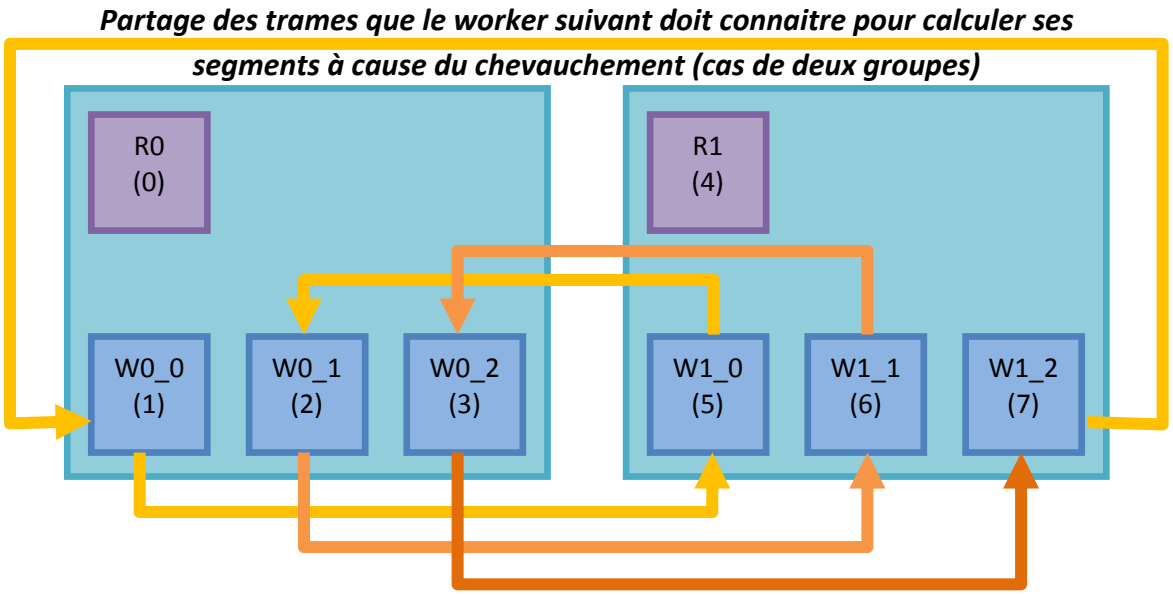


Figure 4.6 : étape 2 ➔ partage des trames avec le *worker* suivant.

Le dernier *worker* partage ses trames avec le *worker* 0 du groupe 0 mais cela ne lui sera utile que pour le cycle de lecture suivant.

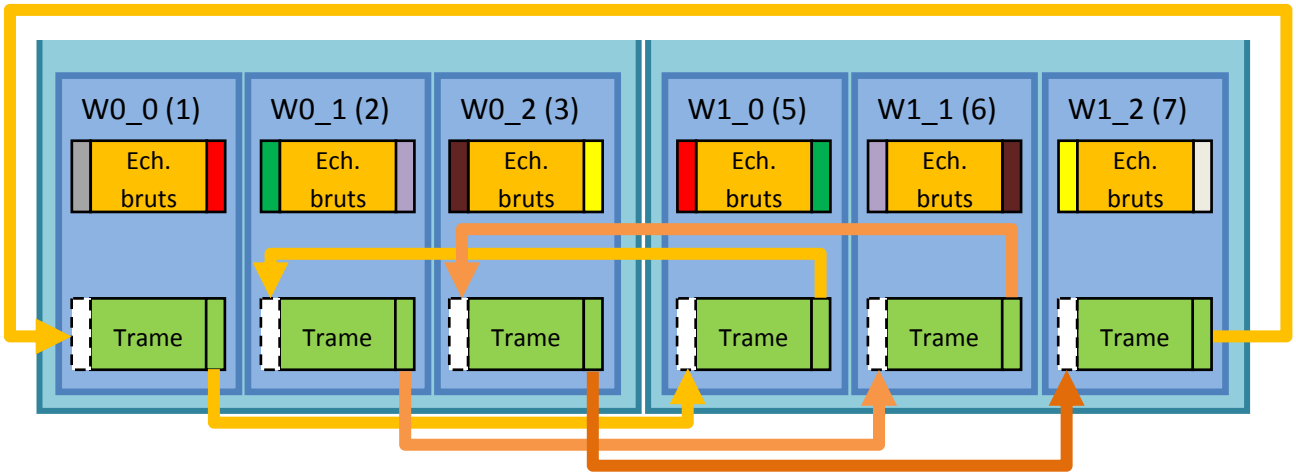


Figure 4.7 : les dernières trames calculées par un *worker* sont envoyées au suivant.

Chaque *worker* envoie au *worker* ayant les trames directement après les siennes, les données qui ne lui permettent pas de calculer un segment complet, ainsi que les données correspondant au chevauchement entre le dernier segment qu'il peut calculer et le segment suivant Toutes les trames envoyées lors de cette étape sont également utiles, les communications sont réduites au strict minimum.

Etape 3 : Centralisation des segments sur le *reader* 0

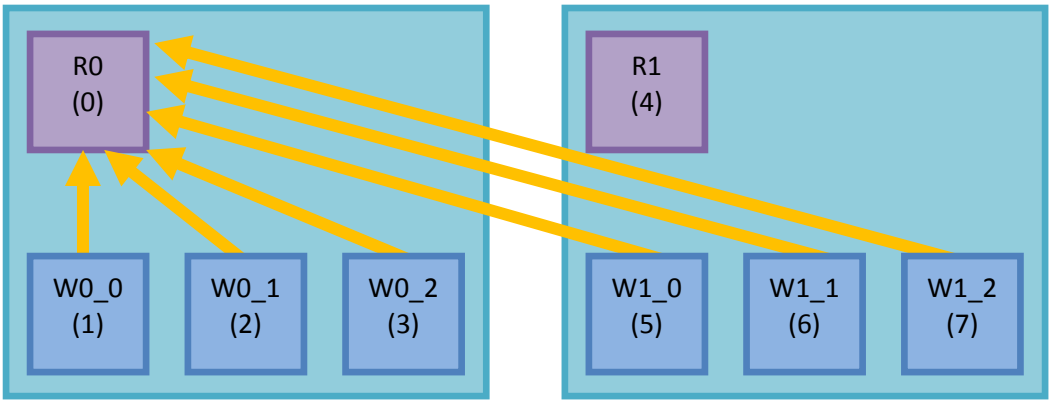


Figure 4.8 : étape 3 ➔ centralisation des segments.

Chaque *worker* renvoie au *reader* 0 l'intégralité des segments qu'il a pu calculer. Le volume des données à envoyer par les processus représente moins de 0.15% du volume des données qu'ils ont reçues lors de la première étape. Cette centralisation est souhaitable pour la mise en œuvre d'une éventuelle normalisation dynamique sur les résultats de segments afin de déterminer les bornes maximales et minimales. L'algorithme de normalisation n'est certes pas idéal d'un point de vue parallélisme mais il a l'avantage de rester très simple et d'être facilement modifiable s'il le faut. C'est donc la solution adoptée pour l'instant, bien qu'elle constitue une limite de passage à l'échelle du programme.

Bouclage sur les étapes 1, 2, et 3 :

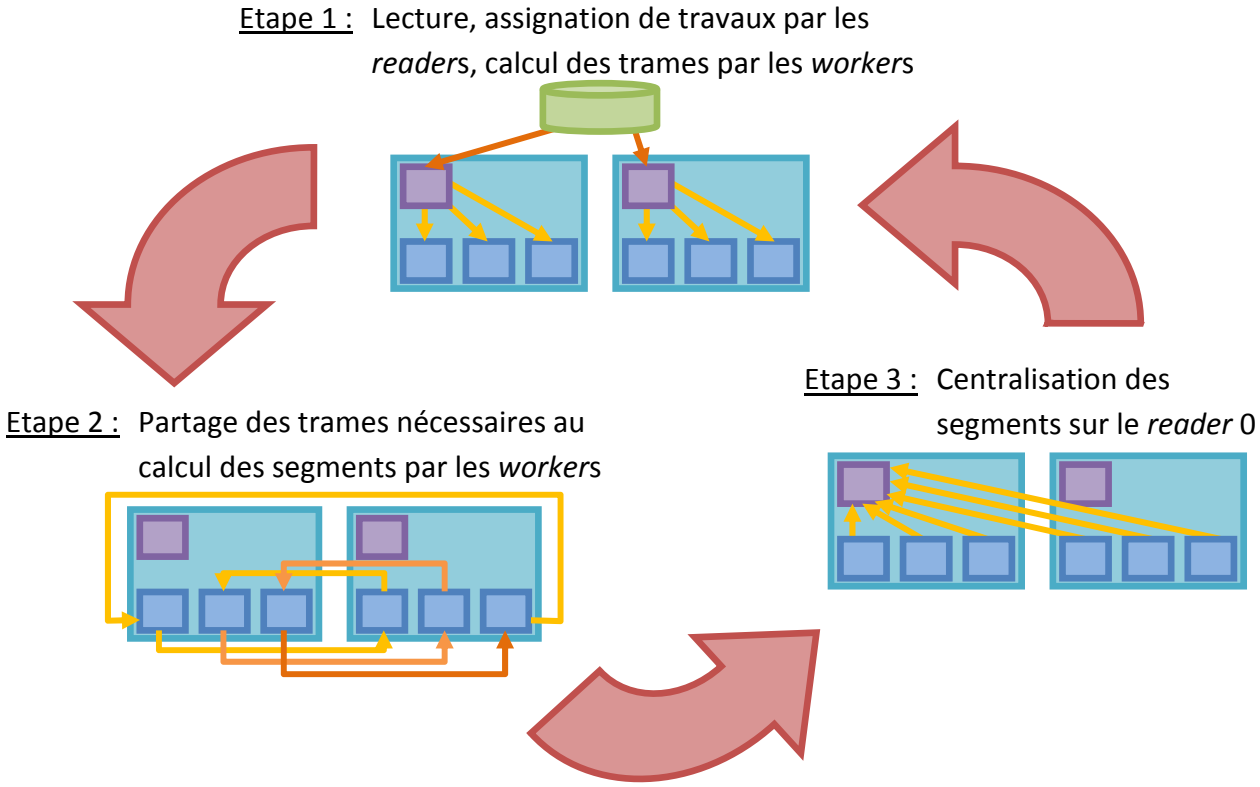


Figure 4.9 : bouclage sur les étapes 1, 2 et 3 jusqu'à atteindre la fin du fichier.

Les étapes 1, 2 et 3 s'enchainent tant que la fin du fichier n'est pas atteinte. Le *reader* 0 disposera à ce moment de l'intégralité des segments du fichier et pourra effectuer la normalisation si nécessaire.

Etape 4 : Assignation des calculs de classification

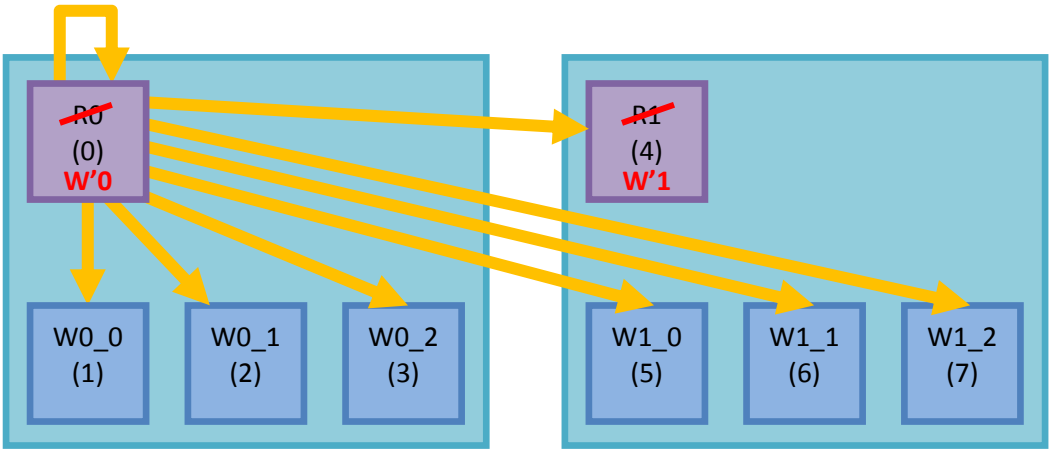


Figure 4.10 : étape 4 ➔ envoi des segments à classifier à l'ensemble des processus.

Les phases de lecture étant finies, les lecteurs abandonnent leur rôle pour participer également à la classification. Le processus *reader* 0, rebaptisé W'0 pour l'occasion, envoie à chaque processus une fraction de l'ensemble des segments à classifier.

Etape 5 : Récupération des résultats

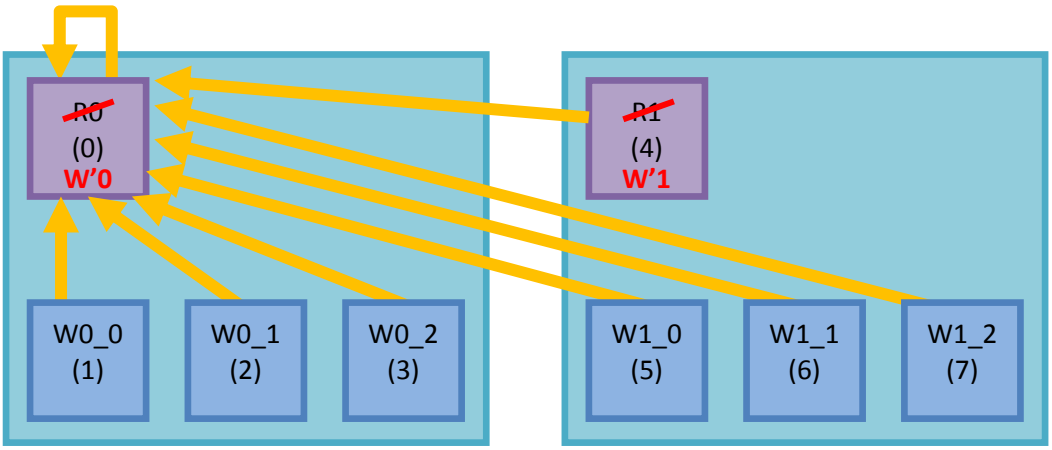


Figure 4.11 : étape 5 ➔ centralisation des résultats sur le *reader* 0.

Une fois la classification effectuée, chaque processus renvoie ses résultats au *reader* 0.

Etape 6 : Consolidation et sauvegarde du résultat final

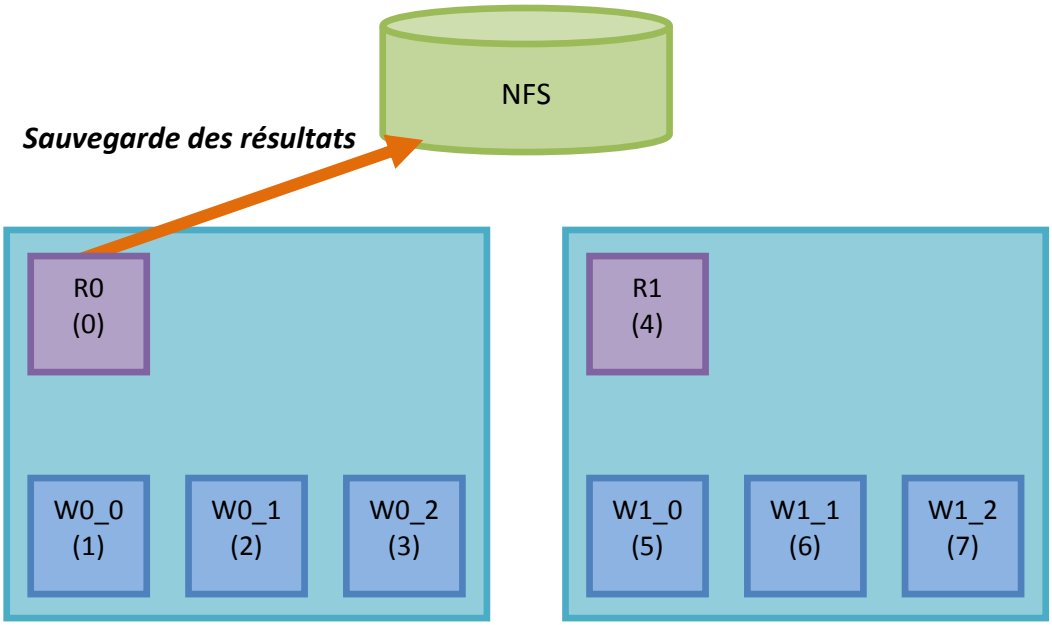


Figure 4.12 : sauvegarde des résultats dans un fichier.

Le *reader* 0 récupère l'ensemble des résultats afin d'en faire un fichier bien ordonné contenant le balisage du fichier analysé.

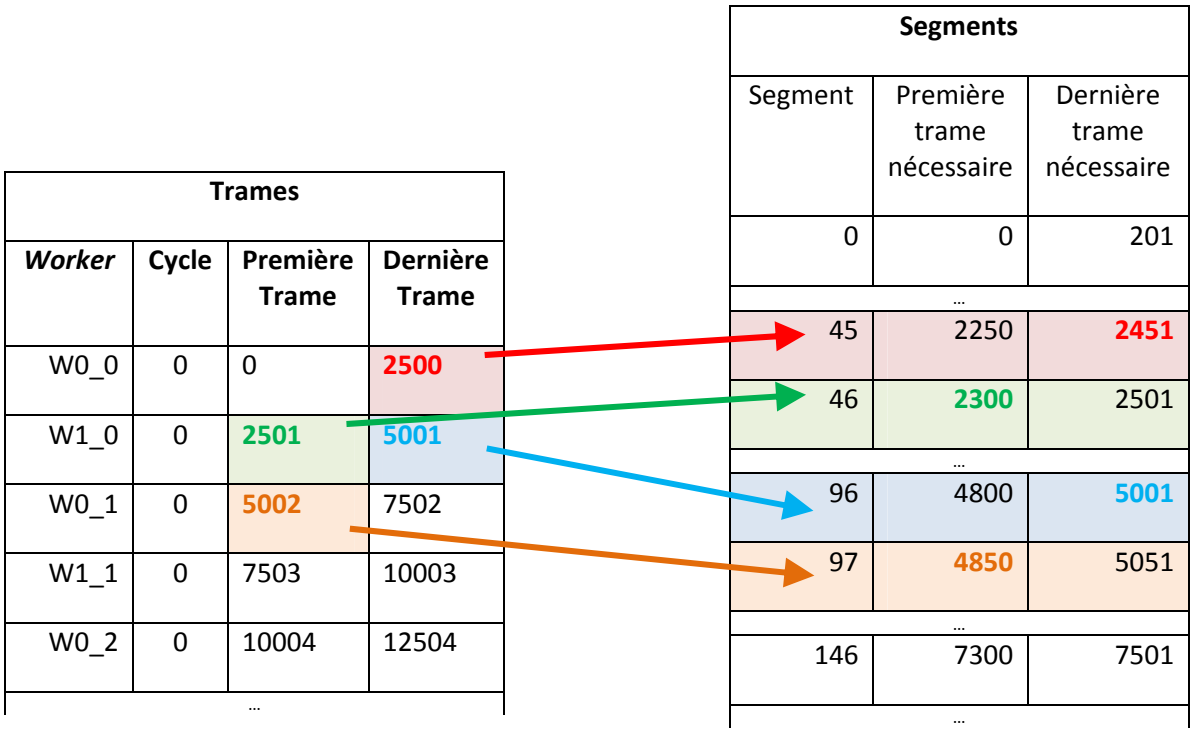
3. Gestion des données

Du fait des multiples chevauchements des trames et des segments, il est relativement compliqué de trouver un algorithme permettant à un *worker*, dans un cycle donné, de calculer quelles sont les trames qu'il va devoir partager et la quantité de données qu'il va recevoir. En revanche, il est plus simple de calculer les répartitions des trames et segments pour tout le fichier, avant même le début du traitement. Pour cette raison il a été choisi de calculer pour chaque processus, lors la phase d'initialisation, l'intégralité de la cartographie des données. Celle-ci est représentée sous la forme de deux tableaux :

- Le tableau des trames, indiquant pour chaque *worker*, la première et la dernière trame qu'il peut traiter à chaque cycle.
- Le tableau des segments, indiquant pour chaque segment la première et la dernière trame nécessaire à son calcul.

Ces tableaux d'indices sont calculés simultanément sur l'ensemble des processus, de manière très rapide et avant l'exécution des premiers traitements. Des boucles identiques à celles permettant d'effectuer la lecture sont utilisées afin de simuler un parcours complet du fichier.

L'exemple suivant montre les deux tableaux représentant la cartographie pour le traitement d'un petit fichier avec un système composé de deux *reader* et de 3 *worker* par *reader*.



Trames

Worker	Cycle	Première Trame	Dernière Trame
W0_0	0	0	2500
W1_0	0	2501	5001
W0_1	0	5002	7502
W1_1	0	7503	10003
W0_2	0	10004	12504
…			

Segments

Segment	Première trame nécessaire	Dernière trame nécessaire
0	0	201
…		
45	2250	2451
46	2300	2501
…		
96	4800	5001
97	4850	5051
…		
146	7300	7501
…		

Figure 4.13 : cartographie de la localisation des débuts de trames et de segments.

Au premier cycle le *worker* W0_0 dispose des trames 0 à 2500. Le dernier segment qu'il peut traiter avec ces données est le segment 45, dont la dernière trame nécessaire est la trame numéro 2451. Le *worker* ayant calculé les trames suivantes (W1_0) devra donc calculer à partir du segment 46. Il lui faut pour cela les trames de 2300 à 2500 que va lui envoyer W0_0. De la même manière, le dernier segment qu'il peut calculer en intégralité est le numéro 96. Il n'a d'ailleurs pas de trames « en trop », mais à cause des segments qui se chevauchent, il devra malgré tout envoyer les trames de 4850 à 5001 au *worker* opérant sur les données suivantes, et ainsi de suite.

4. Réalisation en MPI

MPI est un standard développé principalement par des constructeurs et utilisateurs de calculateurs et définissant les fonctions permettant de réaliser des échanges de messages entre processus. Le but de ce standard est d'offrir une interface identique quels que soient l'architecture des machines, le système d'exploitation et le compilateur utilisé. Certaines implémentations sont propriétaires, développées par des entreprises telles que IBM ou Cray pour optimiser les communications avec leurs calculateurs, alors que d'autres sont libres et disponibles pour un grand nombre d'architectures et systèmes d'exploitation différents [18].

Ses atouts sont :

- La portabilité du code ;
- l'utilisation sur des systèmes à mémoire distribuée ou partagée ;
- Le grand nombre d'applications possible.

MPI offre aux développeurs les outils nécessaires aux communications interprocessus et à la synchronisation, que ce soient des processus sur la même machine ou des machines différentes. En revanche toute la réflexion et la gestion des données à envoyer est à la charge du programmeur.

Les modes de communications sont variés permettant de faire des échanges bloquants ou non bloquants, synchrones ou asynchrones, bufférisés ou non bufférisés. MPI permet également de faire de la synchronisation en bloquant par exemple les processus à un endroit du code en attendant que tous arrivent à ce point avant de les laisser repartir tous en même temps.

Les communications dans ce travail ont été réalisées de manière non bloquante dès que cela avait un intérêt afin de permettre aux un recouvrement des calculs et des communications.

Les fonctions utilisées pour cela sont :

```
int MPI_Issend(void *buf, int count, MPI_Datatype datatype,
  int dest, int tag,MPI_Comm comm, MPI_Request *request);
```

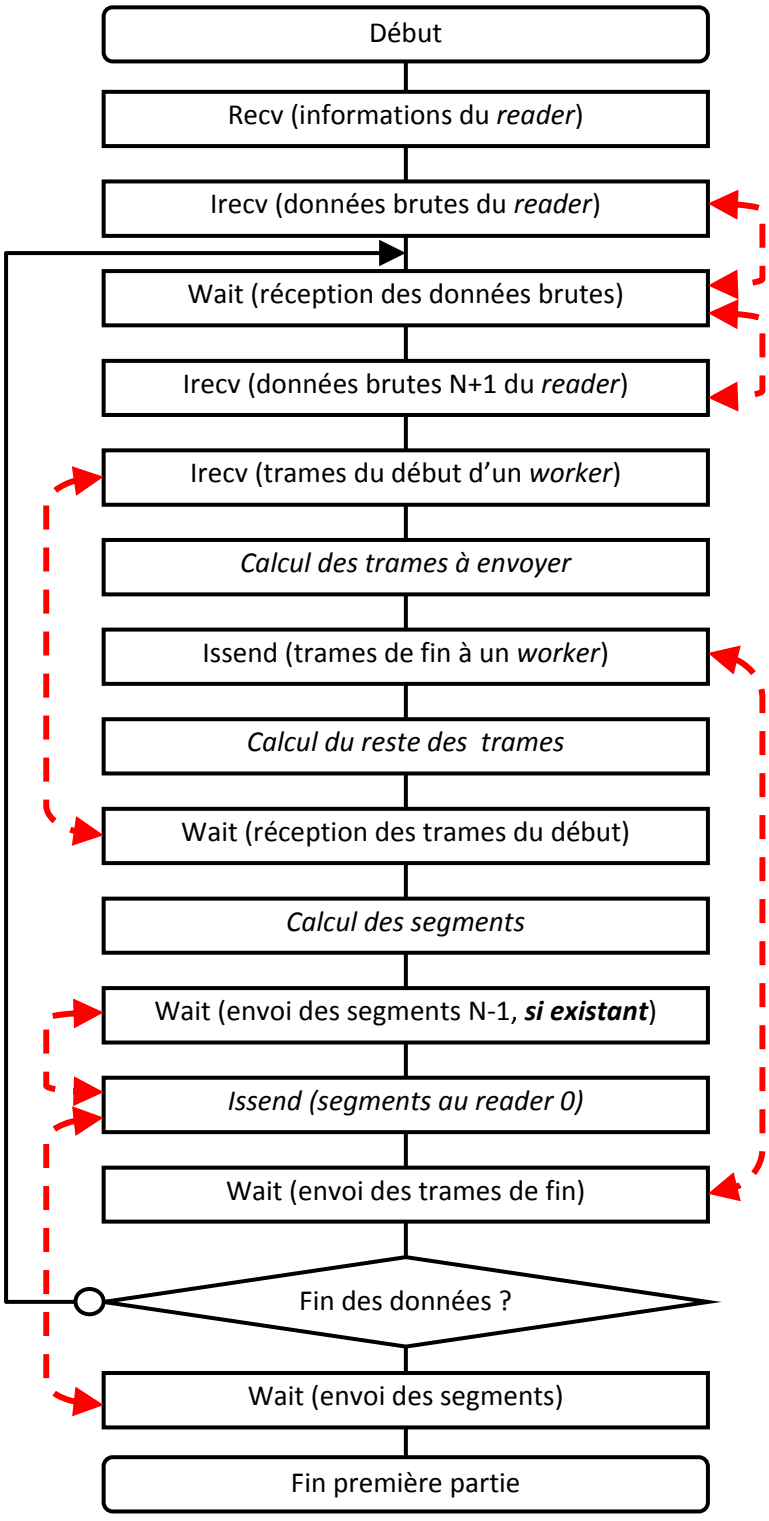
```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
  int source, int tag, MPI_Comm comm, MPI_Request *request);
```

Qui permettent de réaliser un échange synchrone non bloquant. Il est nécessaire de s'assurer avant de réécrire dans le buffer d'envoi, et de se servir des données du buffer de réception que la communication est bien terminée.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status);
```

Cette fonction permet d'assurer à un émetteur ou à un récepteur que la communication est terminée, ou de le bloquer en attendant.

a. Fonctionnement des *worker*



Dans un premier temps les worker reçoivent des informations sur le fichier, telles que sa taille et son taux d'échantillonnage afin de leur permettre de calculer la cartographie des segments et des trames.

Ils font ensuite la demande des premières données et peuvent faire très rapidement la demande des données suivantes pour le cas où le reader mette moins de temps à lire que les worker à traiter les données.

Les worker se mettent en attente des trames qui leur seront nécessaires pour traiter les segments et calculent en priorité les trames à envoyer au worker traitant les segments suivants.

Une fois les trames envoyées, ils calculent le reste des trames et enchainent par le traitement des segments s'ils ont déjà reçu les données nécessaires pour cela. Sinon, ils se mettent en attente le temps que les données arrivent.

Les segments ainsi calculés sont envoyés au reader 0, et, s'il reste des données du fichier à traiter, un nouveau cycle commence.

Figure 4.14 : fonctionnement des *worker* (1/2).

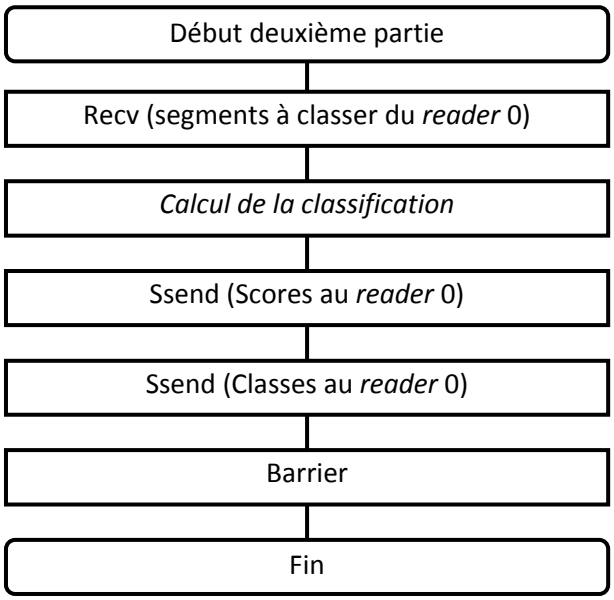
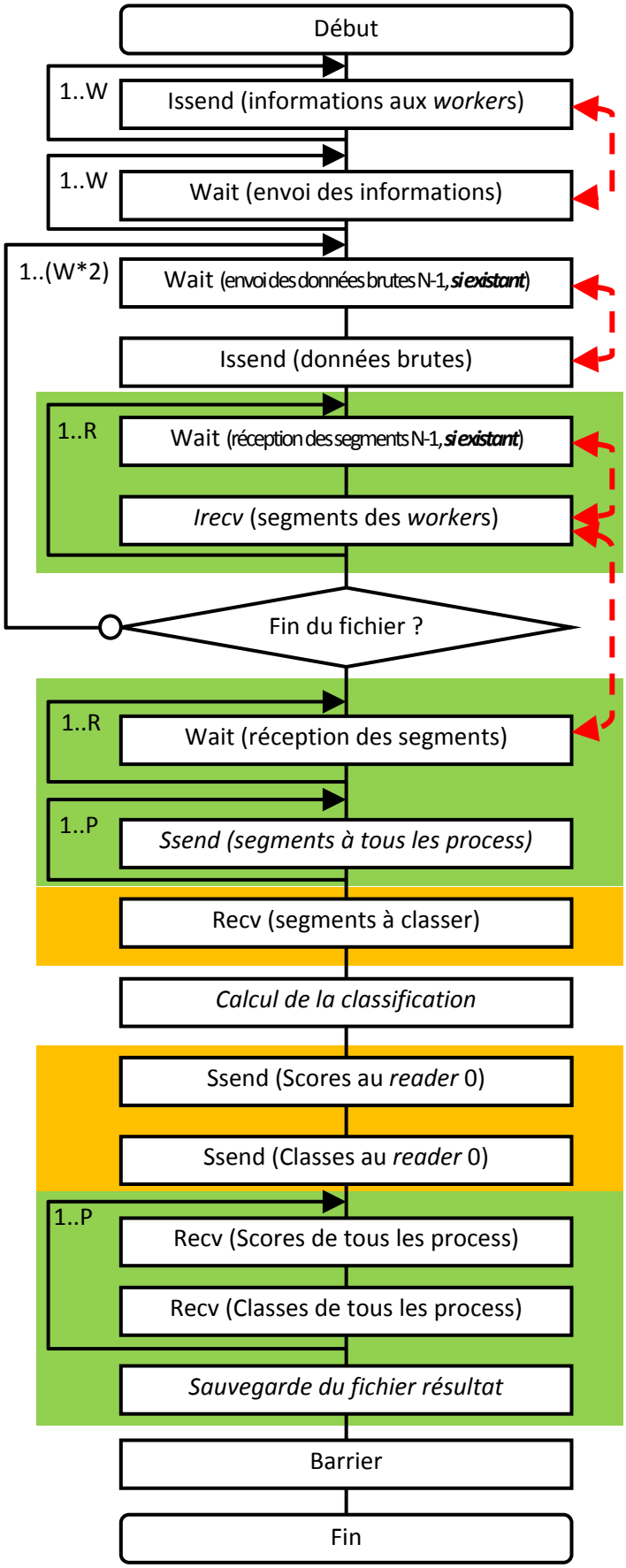


Figure 4.15 : fonctionnement des *worker* (2/2).

La suite est plus simple, les communications sont effectuées de manière bloquante, car les worker n'ont rien d'autre à faire s'ils n'ont pas les données. Ils attendent du reader 0 des segments à analyser. Une fois les traitements nécessaires à la classification effectués, ils envoient les scores et les classes calculés pour l'ensemble des segments au reader 0.

Ils attendent enfin que les autres processus aient fini leurs traitements pour terminer leur exécution simultanément.

b. Fonctionnement des *reader*



*Les étapes indiquées **en vert** sont exécutées uniquement par le **reader 0** et celles **en orange** uniquement par les **autres reader**.*

Les reader envoient les informations sur le fichier à leurs worker et attendent que tous les aient reçues.

Ils envoient les données brutes à leurs worker dès que possible, même s'ils sont encore en train de calculer grâce à l'utilisation de deux buffers par worker et l'utilisation de communications non bloquantes.

Le reader 0 demande les segments aux différents worker de même rang que celui à qui il vient d'envoyer les données. Une fois à la fin du fichier il envoie des segments à traiter à tous les processus, y compris aux autres reader, et se sert des données dont il dispose pour classifier lui-même une partie des segments.

Les autres reader reçoivent les segments à classer et renvoient les scores et classes calculées après traitement.

Le reader 0 reçoit les scores et classes des autres processus et sauvegarde les résultats dans un fichier.

Tous les reader attendent enfin tous les processus aient fini les traitements en cours pour terminer leur exécution en même temps.

Figure 4.16 : fonctionnement des *reader*.

Le maximum a été donc fait pour permettre le recouvrement des communications et des calculs grâce à l'utilisation des communications non bloquantes de MPI. Malheureusement les implémentations de MPI ne permettent pas cela dans le sens où *Issend()* ne semble pas se comporter conformément à la documentation initiale de MPI.

Ce problème a été étudié en détail par les équipes IDMaD et AlGorille seulement à la fin de ce travail de mémoire. Leur analyse confirme que, pour ne pas perturber les calculs parallèles à base de multithreading sur des processus multicoeurs, il n'y a plus de réellement de création de threads par MPI pour la réalisation des communications en parallèle des calculs. Pour parer à ce problème, une nouvelle version du programme devrait encapsuler les communications MPI à réaliser de manière non bloquante dans des threads explicites.

Chapitre 5

Mesures et analyses de performances

Les mesures présentées dans ce chapitre seront effectuées sur deux clusters différents :

- « Intercell » : 256 machines Intel Xeon 3075 @ 2.66GHz (deux cœurs) avec 4 Go de RAM ;
- « Skynet » : 16 machines Intel Core i7 920 @ 2.67GHz (4 cœurs physiques + Hyper-threading) avec 6 Go de RAM.

Pour permettre la création d'une heuristique optimale pour chacun des clusters, il faut balayer l'ensemble des configurations pertinentes possibles en ajustant :

- le nombre de processus lecteurs (nombre de « groupes ») ;
- le nombre de processus MPI par nœud abritant un processus lecteur ;
- le nombre de processus MPI par nœud pour tous les autres nœuds ;
- le nombre de threads OpenMP par processus MPI *worker* ;
- lecture en local ou via le réseau.

De plus, dans le cas de Skynet, les processeurs sont dotés de l'Hyper-threading, une technologie développée par Intel qui permet au système d'exploitation de voir deux fois plus de cœurs logiques qu'il n'y a de cœurs physiques dans le processeur. Chaque cœur logique dispose, en propre, des principaux registres, mais partage le reste des ressources telles que l'unité d'exécution avec un autre cœur logique [19]. Il a donc paru important de quantifier l'éventuel gain généré par l'utilisation de l'Hyper-threading avec notre application.

Un processus *reader* effectue peu de calculs, mais réalise beaucoup d'entrées/sorties, il est donc également intéressant de tester si le nœud abritant le processus lecteur peut être « surchargé » avec un processus *worker* supplémentaire.

Suite à des essais de diverses configurations, on décide, à l'exception des configurations induites par le choix précèdent, de nous limiter aux solutions dont le nombre de processus MPI multiplié par le nombre de threads OpenMP donne le nombre de cœurs physiques de la machine dans le cas des tests sans Hyper-threading, et le nombre de cœurs logiques dans le cas d'expérimentations avec Hyper-threading.

Les mesures ont été effectuées de 1 à 16 groupes pour une lecture sur réseau sur les deux clusters, 1 à 16 groupes pour une lecture en local sur Skynet, et 1 à 256 groupes pour une

lecture en local sur Intercell, en utilisant un nombre variable de nœuds. Des essais ont été réalisés le soir avec 32 *reader* sur Intercell pour une lecture via NFS, mais cela ne semble pas améliorer les performances.

Les performances sont donc mesurées sur les deux clusters. Pour chacun d'eux, de nombreuses mesures ont été faites, mais il est possible de les regrouper en quatre graphiques pour mettre facilement en avant les différences entre plusieurs modes de fonctionnement. Le fichier traité est identique dans tous les cas : il s'agit d'un fichier audio stéréo de 6 heures dont la taille est de 3.6 Go. Nous avons choisi un fichier de cette taille, car cela permet d'avoir, dans tous les cas, des mesures de temps de traitement pertinentes (entre environ 9 et 2500 secondes).

Les graphiques présentés seront donc les temps obtenus avec les modes suivants :

- A « froid » avec une lecture en réseau ;
- A « chaud » avec une lecture en réseau ;
- A « froid » avec une lecture en local ;
- A « chaud » avec une lecture en local.

A « froid » signifie que c'est la première fois que ce fichier a été traité depuis un moment, les données le concernant ne sont vraisemblablement pas en cache disque, c'est typiquement le cas rencontré en production lors du traitement de nouveaux fichiers.

A « chaud » signifie par contre que le fichier a été traité plusieurs fois d'affilée, on considère dans ce cas le temps moyen de la durée des exécutions successives sans compter le premier temps. Les caches peuvent être utilisés par les machines de manière transparente pour l'utilisateur. Il s'agit d'un mode de fonctionnement qui est rencontré en phase de débogage et de recherche de nouvelles caractéristiques, lorsque le programme est exécuté successivement sur les mêmes fichiers de test avec des paramètres différents.

1. Mesures réalisées sur Intercell

Les résultats présentés ici montrent les performances obtenues pour des exécutions sur 1 à 256 machines du cluster Intercell de Supélec en faisant varier les différents paramètres afin de trouver la configuration qui donne les meilleurs temps de traitement.

a. Interprétation des résultats

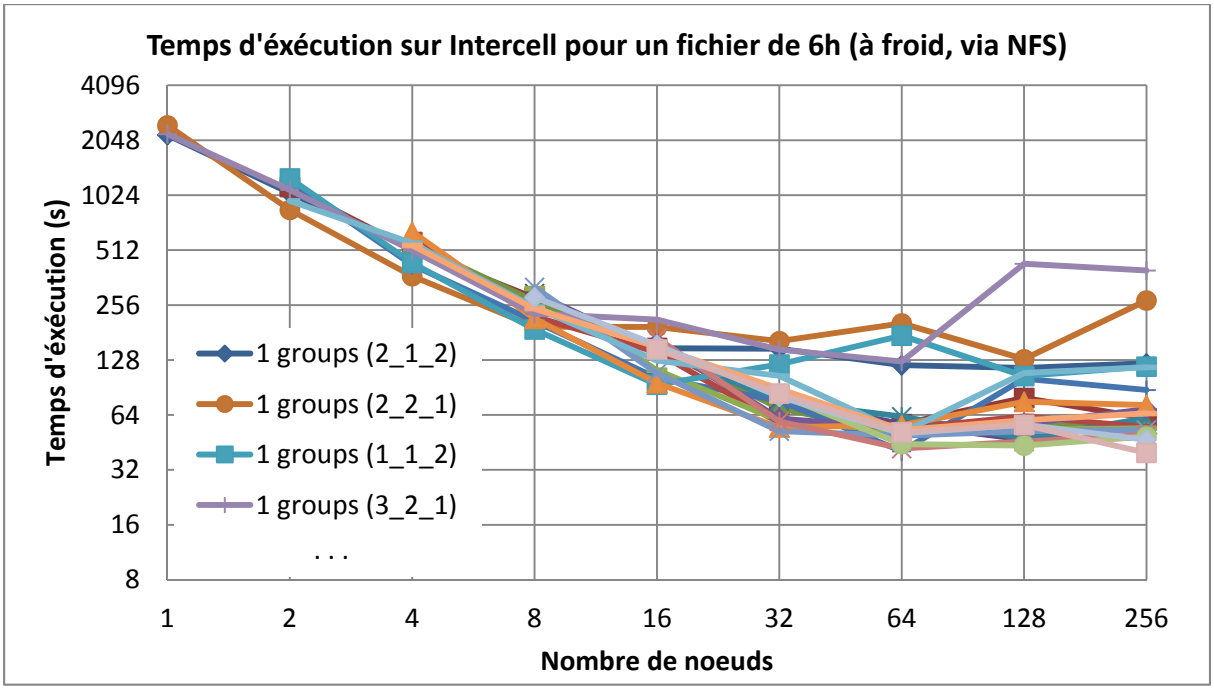


Figure 5.1 : benchmarks complets du temps d'exécution sur Intercell, à froid, en réseau.

Ce graphique est utilisé à titre d'exemple pour démontrer qu'il n'est pas forcément judicieux de disposer de l'intégralité des données car il est peu lisible à cause du nombre de courbes. Il y en aurait eu 36 avec la lecture en local car, dans ce cas, il est possible de monter le nombre de lecteurs à 256 en simultané, chose qui aurait été trop risquée en réseau.

Les nombres entre parenthèses dans la légende indiquent le nombre de processus et threads. Pour 3_2_1 par exemple, Il y a 3 processus MPI sur les nœuds contenant le *reader*, 2 processus sur les nœuds ne contenant que des *worker*, et 1 thread de calcul par processus *worker*.

Pour le point à **16** nœuds de la courbe « **4** groups (**3**_**2**_**1**) » cela signifie donc :

- **4** processus *reader* ;
- **16** nœuds ;
- **3** processus MPI sur les **4** nœuds abritant un *reader* (1 *reader* + 2*worker*) ;
- **2** processus MPI sur les 12 autres nœuds (2 *worker*) ;
- **1** thread de calcul par processus *worker*.

Au final, cette configuration aura nécessité 36 processus MPI (**4**x**3**+(**16**-**4**)x**2**).

Les prochaines courbes qui seront présentées ne contiendront que les configurations pertinentes, c'est-à-dire celles qui sont les meilleures en au moins un point de l'axe des abscisses.

b. Performances à froid, en réseau

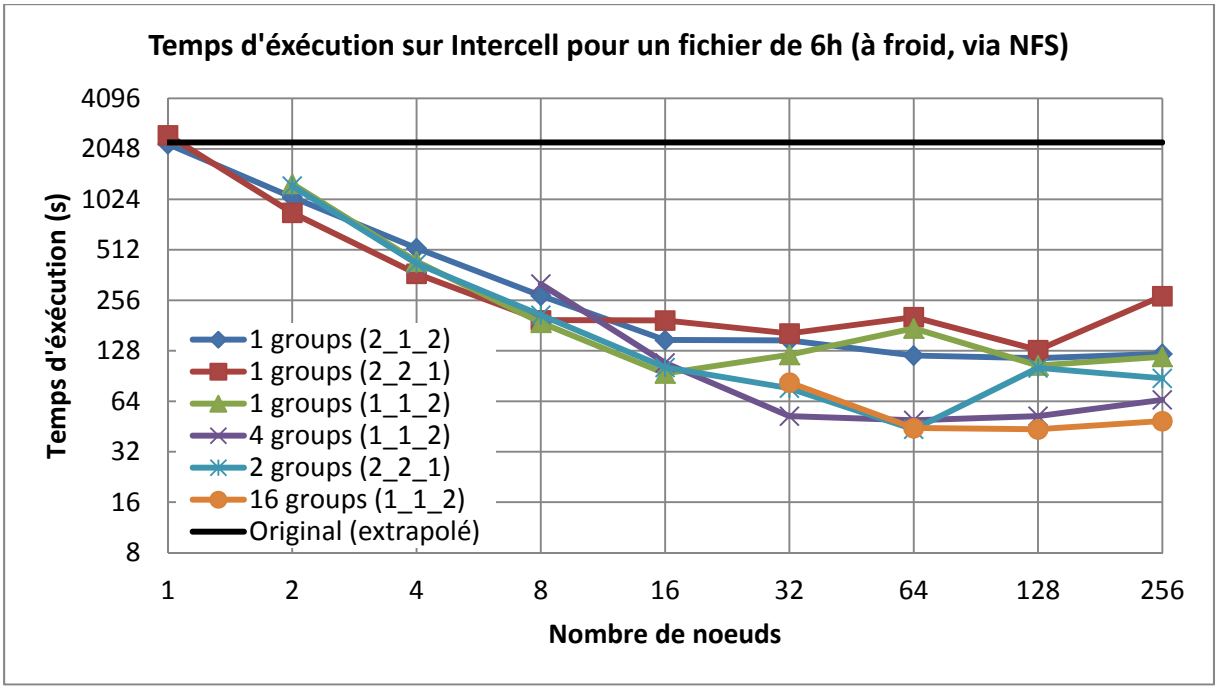


Figure 5.2 : benchmarks du temps d'exécution sur Intercell, à froid, en réseau.

Le temps d'exécution du programme original a été extrapolé à partir du temps mis pour traiter un fichier plus petit car il n'était pas capable de traiter un fichier aussi important.

On remarque que, pour un nœud, le temps d'exécution est à peine légèrement plus court que ce qu'il aurait été avec le programme original (2189 secondes contre 2237). On pourrait s'attendre à ce que le traitement soit beaucoup plus rapide grâce à toutes les optimisations réalisées précédemment (voir page 27), mais ce n'est malheureusement pas le cas. Cela s'explique par le fait que l'overhead lié à la gestion des *worker* par le *reader* est important lors de l'exécution sur un nœud et que les processus se ralentissent les uns les autres. On peut toutefois se consoler en se disant que, de toute manière, on ne réalise pas la distribution d'un programme complexe pour finalement ne l'exécuter que sur un seul nœud et que, malgré tout, nous sommes légèrement plus rapides.

L'intérêt de la distribution est flagrant dès l'utilisation sur deux nœuds, avec un temps d'exécution 2.63x plus rapide que l'optimal sur un nœud.

Pour juger de l'intérêt de la parallélisation, nous ne retiendrons que l'enveloppe inferieure du graphique précédent, c'est-à-dire le meilleurs temps obtenus en faisant varier les configurations pour un nombre de nœuds donné.

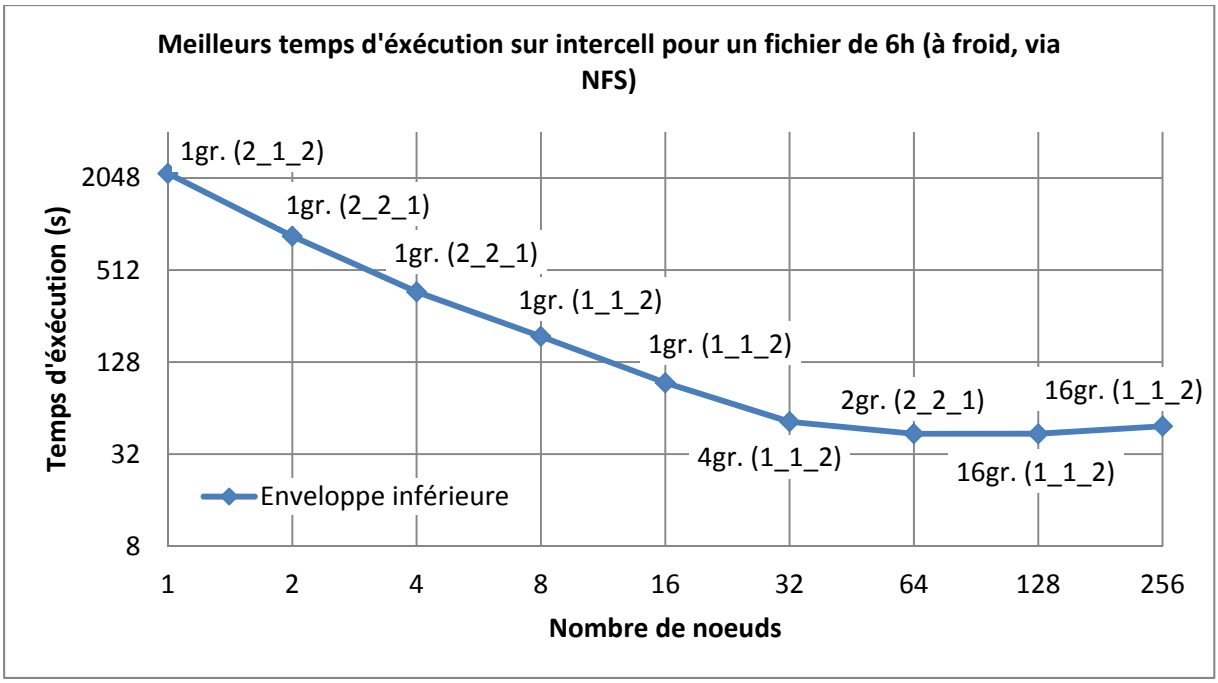


Figure 5.3 : courbe de l'enveloppe inférieure, à froid, en réseau.

Cette courbe ne sera pas tracée à chaque fois mais on considère ces données comme étant les performances obtenues grâces à la parallélisation de programme. De plus, ce sont ces valeurs qui servent au calcul des points de la courbe de *speedup* ci-après.

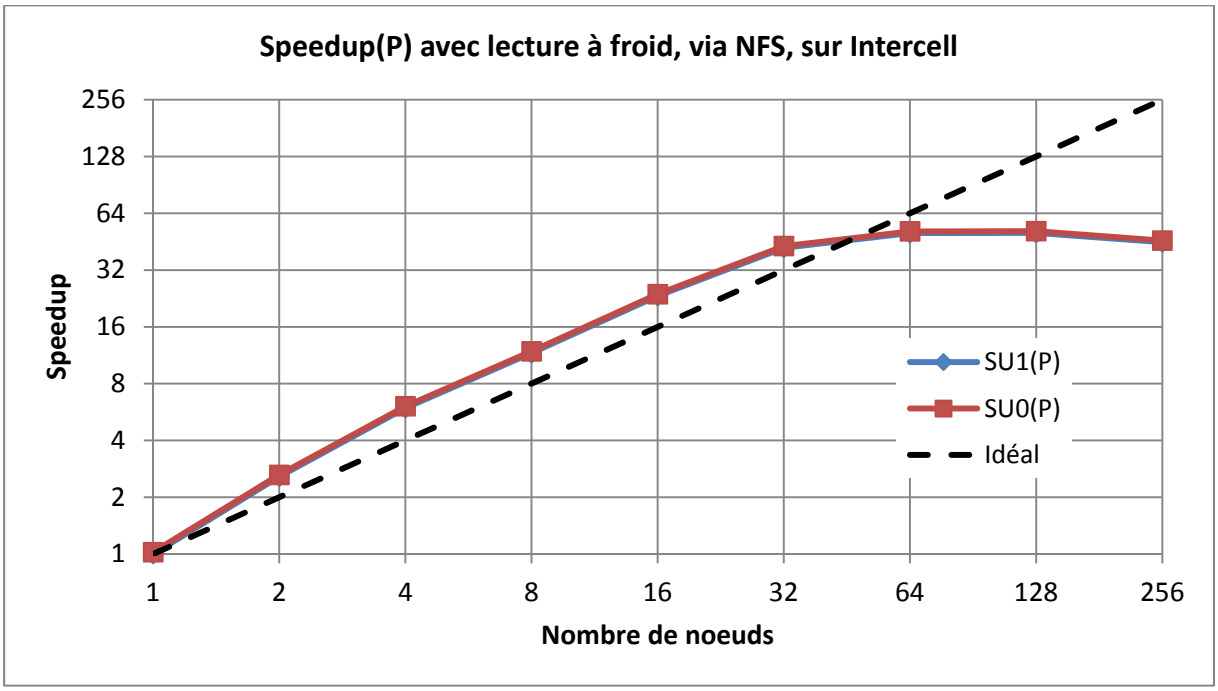


Figure 5.4 : speedup sur Intercell, à froid, en réseau.

La courbe du *speedup* peut choquer, car les performances obtenues sont souvent supérieures à l'idéal. Malheureusement cette « hyperaccélération » (*superlinear speedup*),

est simplement liée au fait que le système fonctionne de manière bien inférieure sur un seul nœud, et que c'est justement ce point qui sert de référence dans la courbe du *speedup*.

SU0(P) correspond au *speedup* par rapport au temps qu'aurait mis l'outil original, et SU1(P) correspond à celui par rapport au temps d'exécution de la version distribuée sur un seul nœud. Dans ce cas, les courbes sont superposées car les temps servant de références pour chacune d'elles sont très proches.

Les courbes sont très intéressantes, et quasiment linéaires jusqu'à 32 nœuds. Utiliser 64 nœuds a encore un intérêt dans le sens ou la courbe croît légèrement par rapport à 32 machines. Au-delà, par contre, le *speedup* chute.

c. Performances à chaud, en réseau

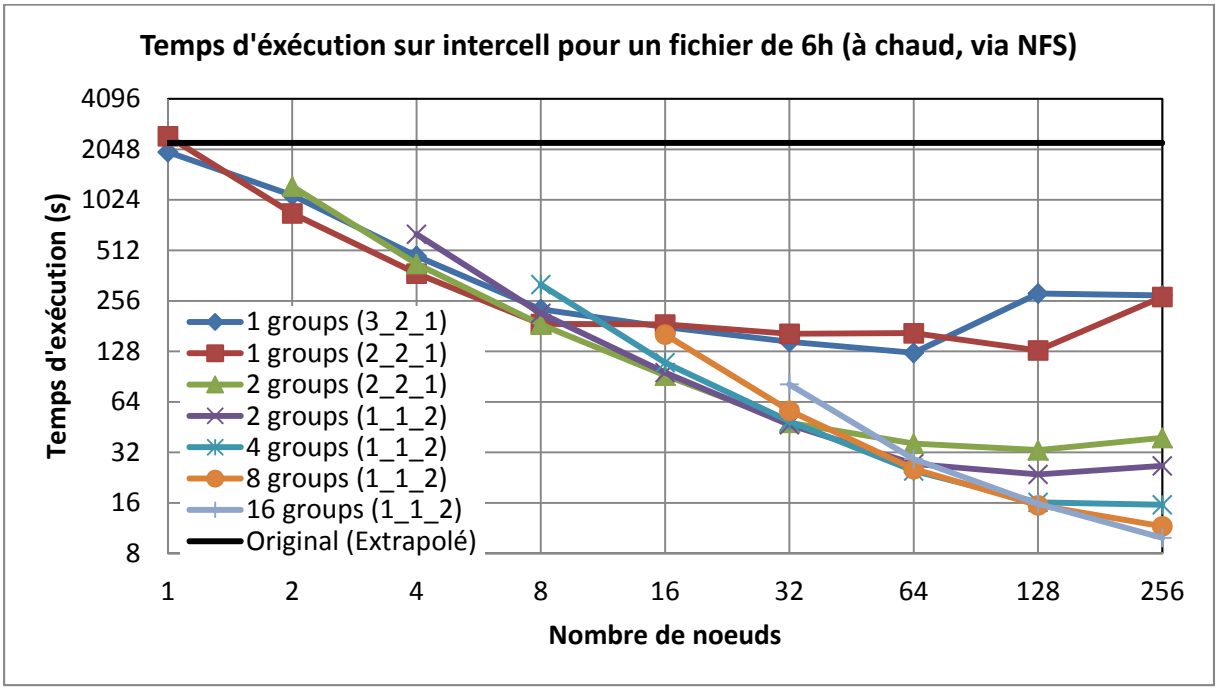


Figure 5.5 : benchmarks du temps d'exécution sur Intercell, à chaud, en réseau.

Pour un nœud le temps d'exécution est toujours légèrement inférieur au programme original, mais on remarque avant tout que le fait d'utiliser plusieurs nœuds donne des résultats intéressants. La durée d'exécution raccourcit au fur et à mesure que l'on rajoute des nœuds, même si on commence à remarquer un début « d'essoufflement » vers 256 machines.

Dans ce test « à chaud » on effectue un grand nombre de fois les mêmes traitements sur le même fichier. Cela n'a aucune utilité en production mais s'avère très intéressant dans le cas d'utilisation du chercheur en traitement de signal qui développe de nouvelles fonctionnalités dans l'application. Pour cette personne, il est intéressant de pouvoir exécuter successivement l'application avec les meilleures performances possible sur les mêmes

données en ajustant simplement quelques paramètres des nouvelles fonctionnalités sur lesquelles il travaille.

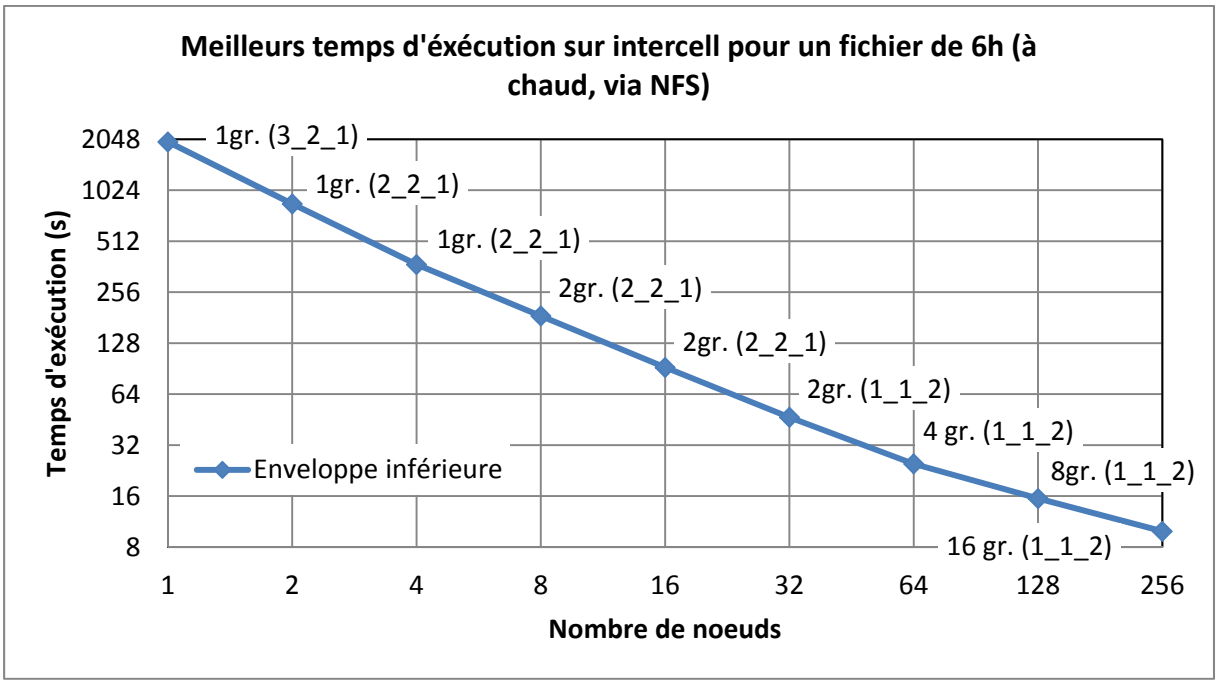


Figure 5.6 : courbe de l'enveloppe inférieure, à chaud, en réseau.

Toujours en considérant l'enveloppe inferieure, on se rend compte qu'il est plus intéressant de surcharger le nœud lorsqu'il est seul à faire le traitement. De 2 à 16 nœuds il est plus intéressant de fonctionner avec 2 processus mpi par nœud et 1 thread OpenMP par processus, alors que de 32 à 256 il vaut mieux avoir 2 threads avec seulement un processus.

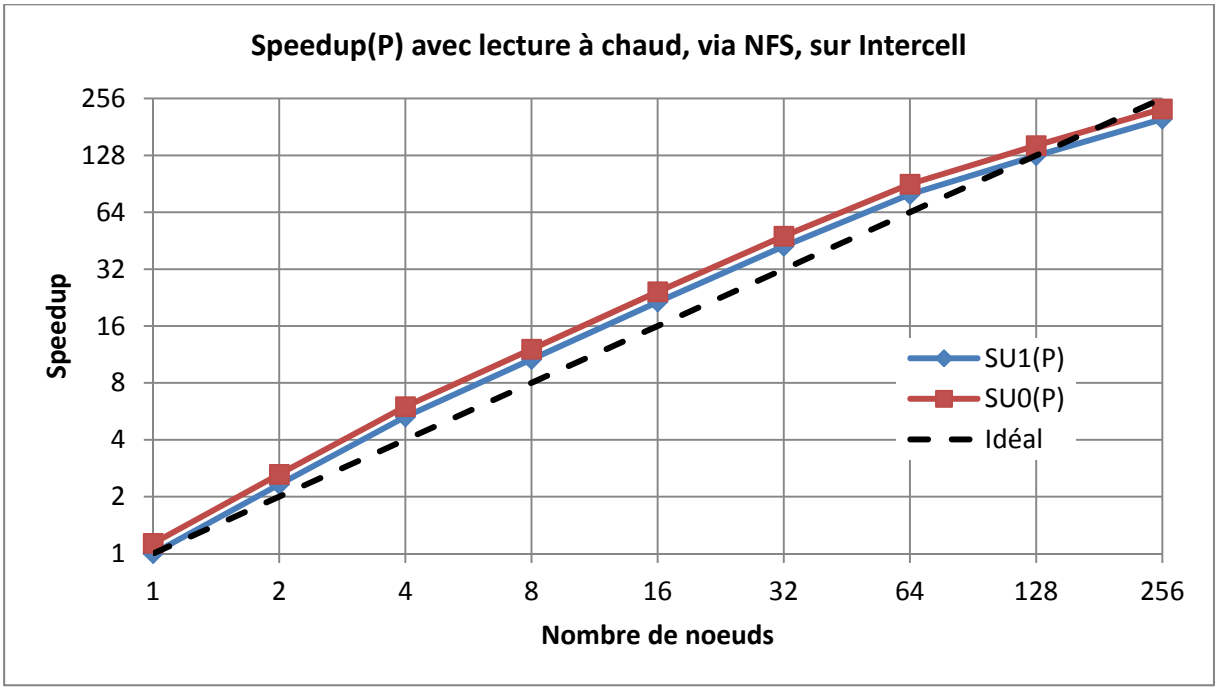


Figure 5.7 : speedup sur Intercell, à chaud, en réseau.

Les courbes SU0(P) et SU1(P) seront ici légèrement disjointes, du fait que le programme distribué sur un nœud a fonctionné « sensiblement » plus rapidement que le programme original contrairement à la courbe de *speedup* précédente.

Les courbes de *speedup* sont relativement linéaires jusqu'à 128 nœuds. On remarque également de manière plus flagrante l'essoufflement avec 256 machines, les courbes passant sous l'idéal. Malgré tout, il reste très intéressant d'utiliser les 256 machines dans ce cas.

d. Performances à froid, en local

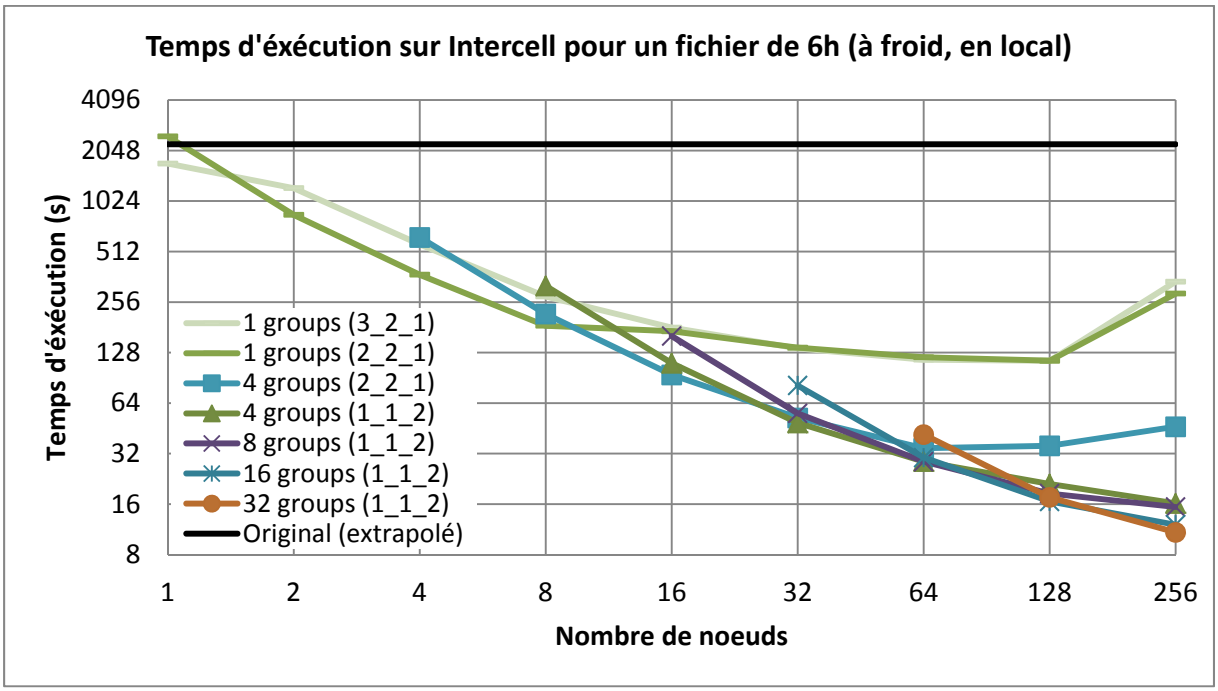


Figure 5.8 : benchmarks du temps d'exécution sur Intercell, à froid, en local.

On remarque ici une amélioration des performances sur un nœud. Globalement la courbe a la même allure que celle en réseau à chaud. Les résultats sont très bons, l'enveloppe inférieure étant plutôt linéaire, même pour un nombre important de nœuds.

La tendance identifiée précédemment semble se justifier. Pour un fonctionnement avec un seul nœud, il vaut mieux qu'il soit surchargé. De 2 à 16 nœuds, il est plus intéressant de fonctionner avec 2 processus MPI par nœud et 1 thread OpenMP par processus, alors que de 32 à 256 nœuds on obtient les meilleures performances avec 1 processus et deux threads.

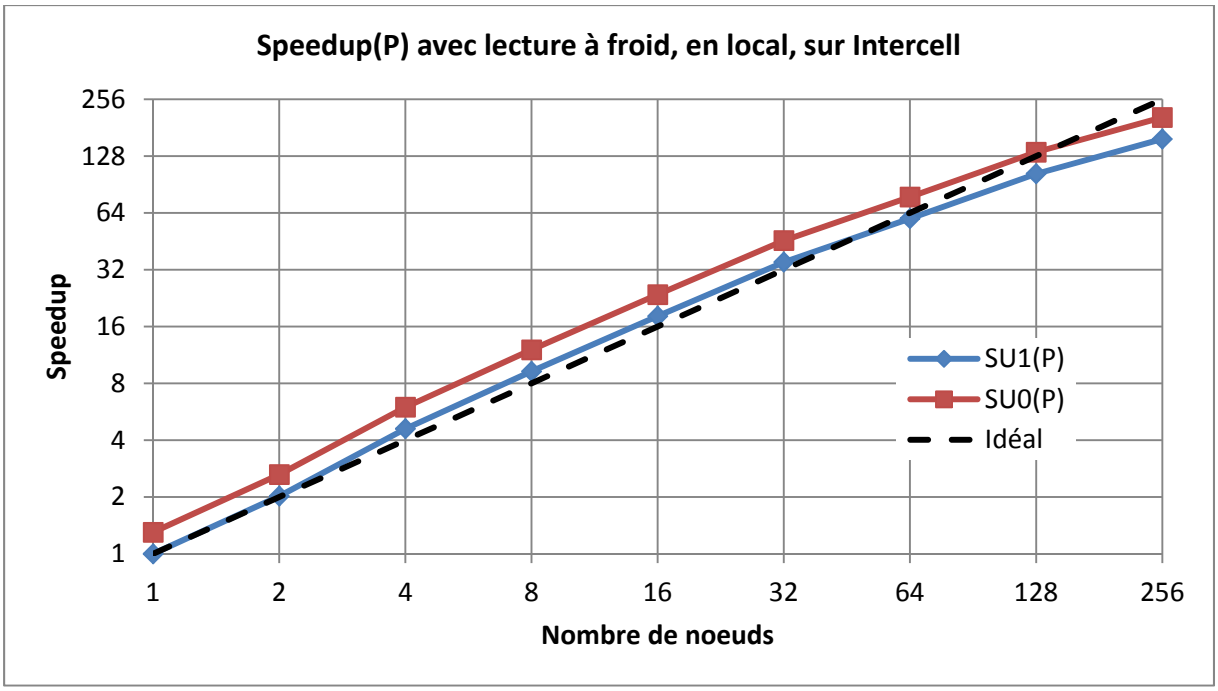


Figure 5.9 : speedup sur Intercell, à froid, en local.

Les courbes de *speedup* se tassent tôt que précédemment, mais la pente de 128 à 256 nœuds semble similaire. Cela laisse à penser que les performances ne seraient pas forcément très intéressantes en doublant encore le nombre de machines. Cela dit, jusqu'à 256 nœuds les résultats sont très intéressants.

e. Performances à chaud, en local

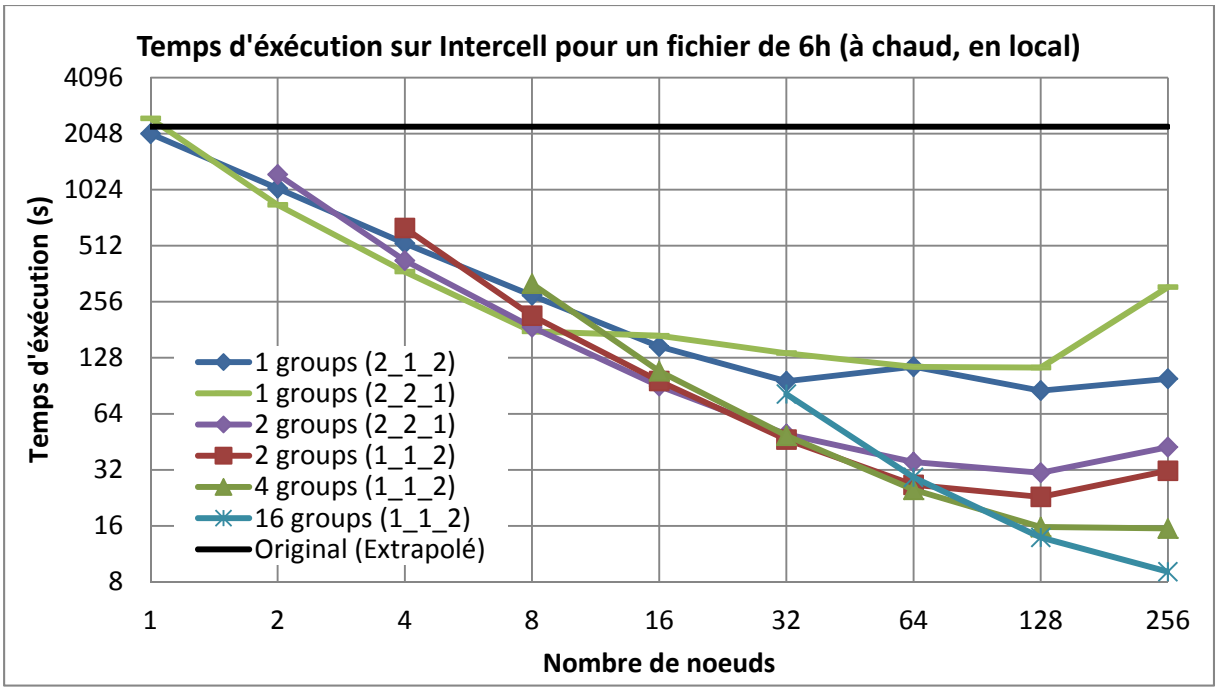


Figure 5.10 : benchmarks du temps d'exécution sur Intercell, à chaud, en local.

Les performances sont globalement semblables à celles à froid et en local, quoique légèrement meilleures. Cependant, les configurations utilisées pour atteindre ces résultats ne sont pas les mêmes, abandonnant par exemple le fonctionnement utilisant 32 groupes pour 256 machines au profit de 16 groupes seulement. La configuration donnant les meilleures performances avec un nœud est également différente. La machine est toujours surchargée, mais d'une autre manière en utilisant un processus *reader* et un processus *worker* ayant 2 threads OpenMP. Toutefois, comme précédemment, à partir de 32 nœuds il est plus intéressant de limiter le nombre de processus MPI au profit de threads OpenMP.

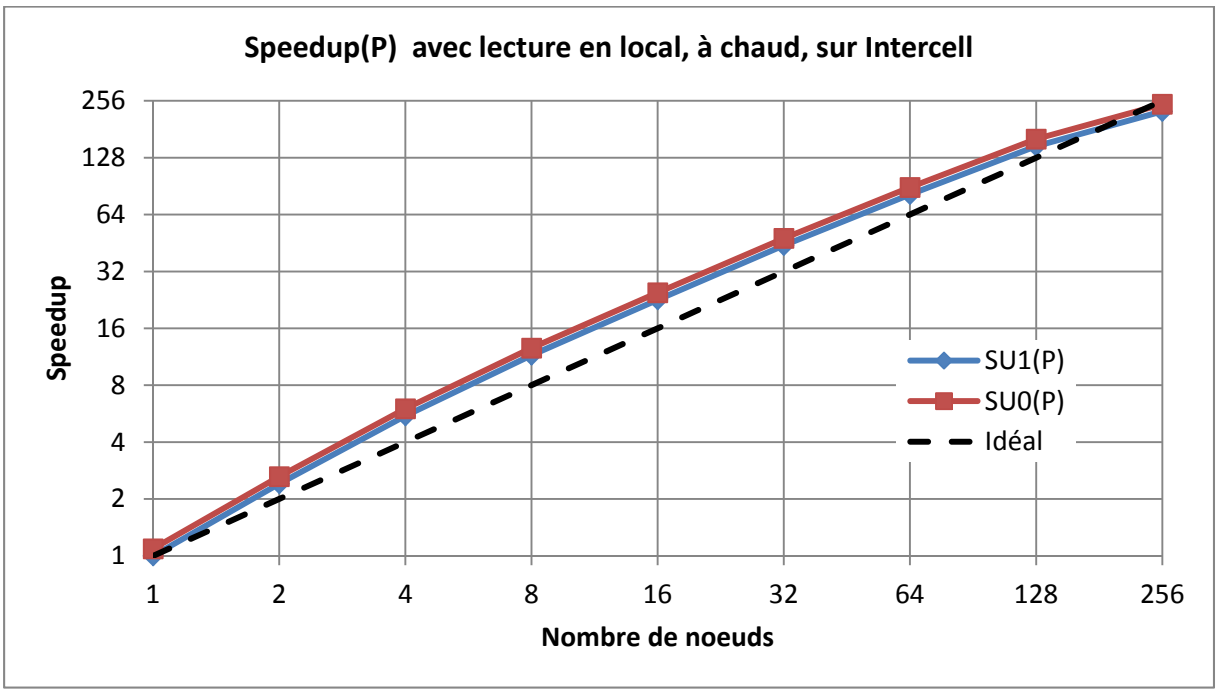


Figure 5.11 : speedup sur Intercell, à chaud, en local.

Le *speedup* est croissant jusqu'à 256 nœuds, mais on remarque aussi un début d'essoufflement vers la fin.

f. Synthèse pour Intercell

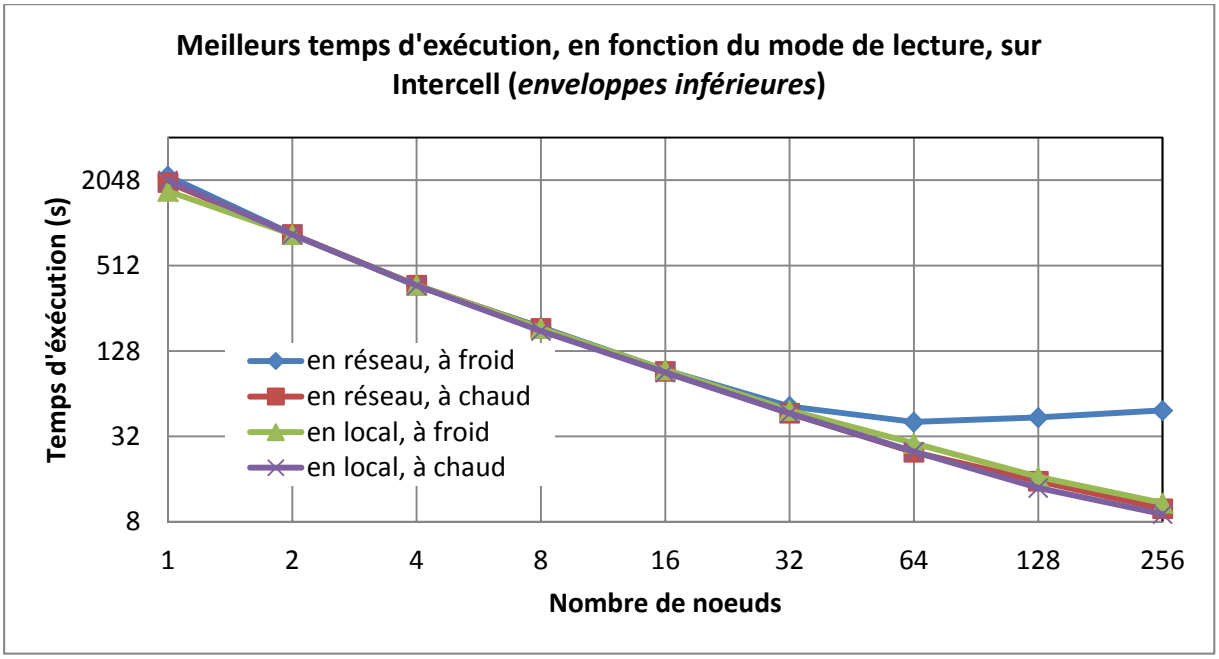


Figure 5.12 : meilleurs temps d'exécution sur Intercell

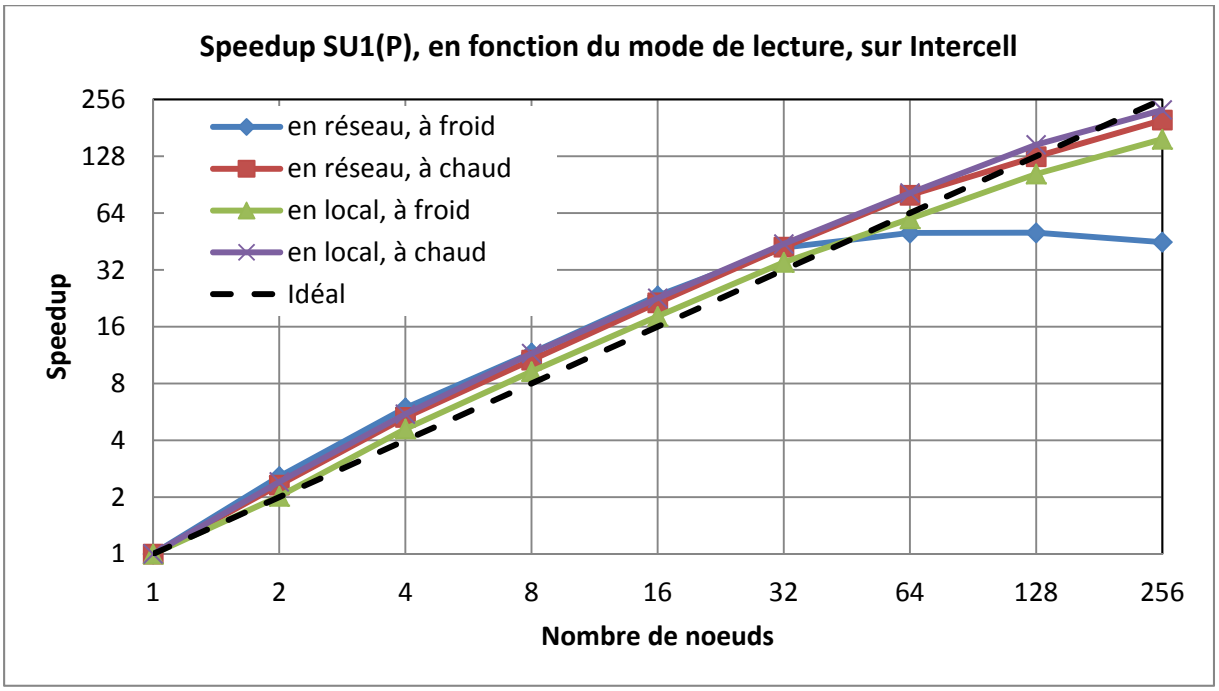


Figure 5.13 : courbes de speedup SU1(P) sur Intercell

On observe, dans ces courbes, une hyperaccélération liée au fait que les performances ne sont pas très bonnes sur un nœud. Le *speedup* est supérieur au *speedup* idéal dès l'utilisation d'une 2^{e} machine. Pour la lecture à froid et en réseau, il est clair qu'il existe une limite du nombre de nœuds après laquelle les performances ne s'améliorent plus, malgré

l'utilisation d'ordinateurs supplémentaires. On remarque une légère amélioration du temps d'exécution en passant de 32 à 64 machines, mais après cela, les performances stagnent, voire se dégradent. Pour les autres courbes, les performances semblent s'amoindrir vers 256 nœuds, laissant imaginer que le même genre de phénomène risque de se produire.

La formule de calcul de l'*efficacité de la parallélisation* est la suivante :

$$E(P) = \frac{SU(P)}{P}$$ avec P égal au nombre de nœuds.

Dans notre cas, elle est calculée en utilisant les *speedup* SU1(P), c'est-à-dire en utilisant comme référence le temps obtenu par le programme distribué sur un seul nœud. L'efficacité correspond en réalité au taux d'utilisation des ressources en prenant comme référence les performances obtenues sur un nœud.

Exemple avec 10 machines, et un *speedup* calculé de 8 :

$$E(10) = \frac{SU(10)}{10} = \frac{8}{10} = 80\%$$

On peut considérer dans ce cas que 80% du temps passé par les 10 machines est du temps utile, alors que les 20% de temps restants sont perdus en communications MPI et en synchronisations interprocessus.

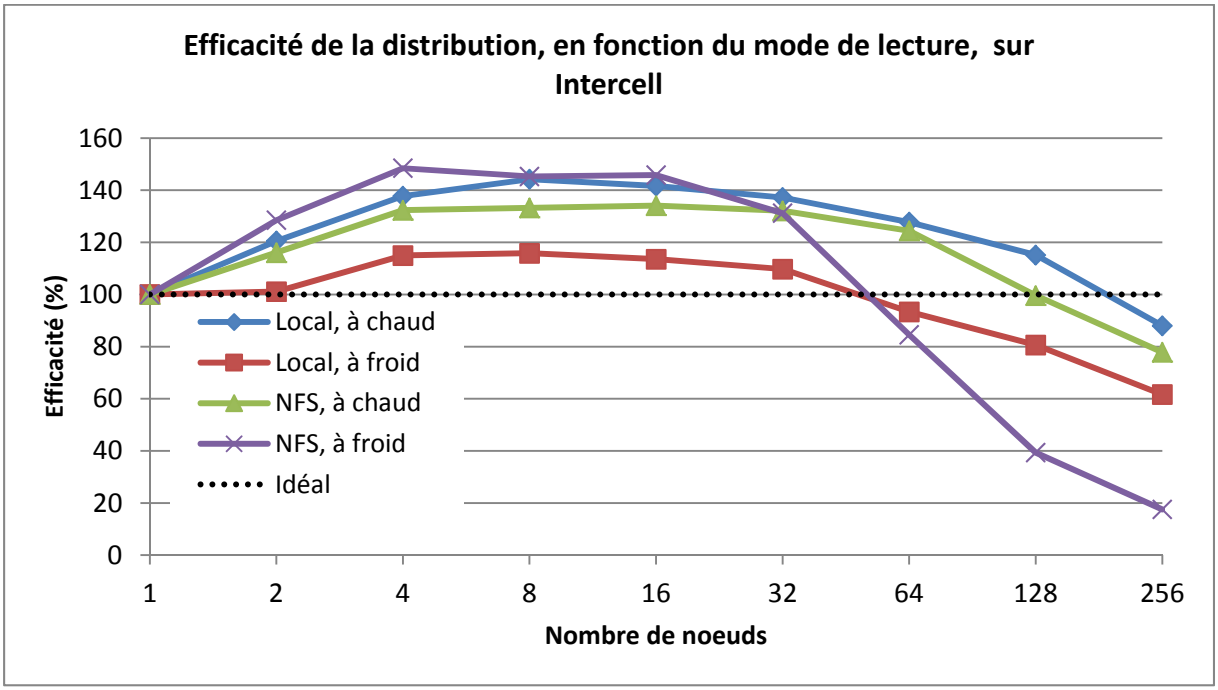


Figure 5.14 : efficacité de la distribution sur Intercell.

Cette courbe nous montre encore une fois qu'ajouter plus de machines est de moins en moins « rentable » au-delà de 32 nœuds.

2. Mesures réalisées sur Skynet

Les benchmarks présentés ici ont été réalisés sur le cluster Skynet de Supélec composé de 16 machines. Ces machines ont la particularité de disposer de cartes graphiques Nvidia offrant ainsi la possibilité de faire du calcul distribué sur GPU pour les applications le permettant.

Ce cluster est composé de 16 machines disposant d'un CPU Core i7 920 de génération « Nehalem » offrant quatre cœurs physiques + *hyper-threading* pour un total de huit cœurs logiques.

Avec huit cœurs logiques, les possibilités de configurations différentes sont démultipliées pour tester l'intérêt ou non d'utiliser l'*hyper-threading* avec ce programme.

a. Les performances à froid, en réseau

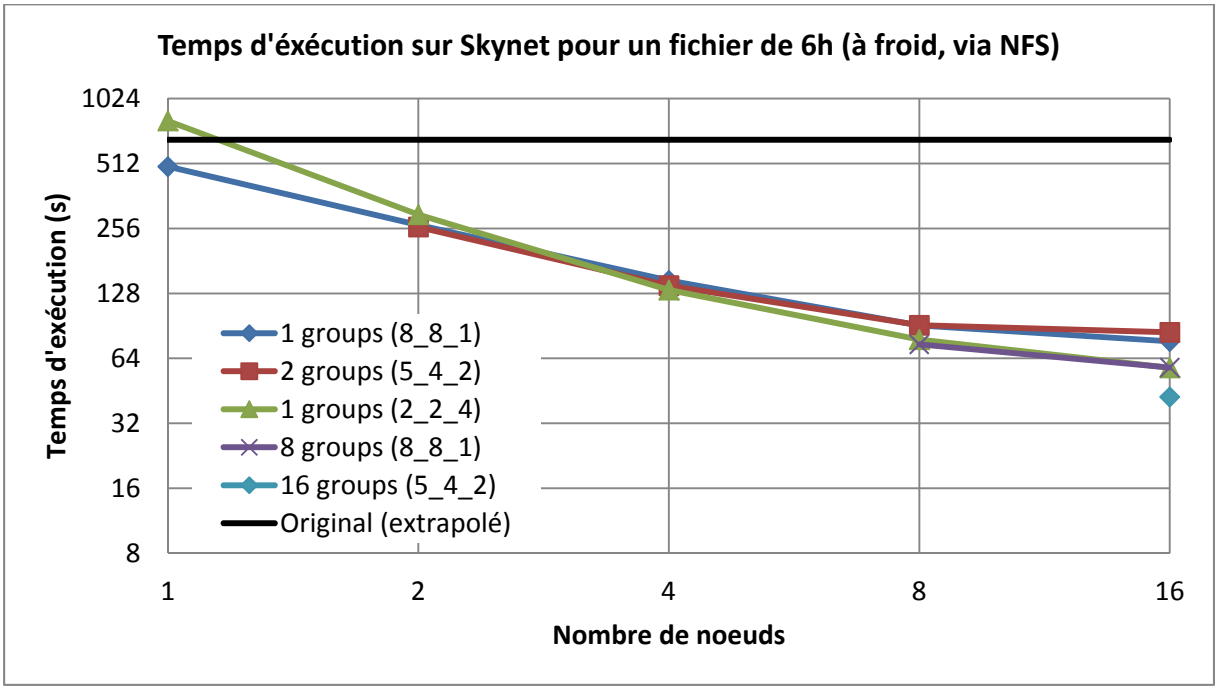


Figure 5.15 : benchmarks du temps d'exécution sur Skynet, à froid, en réseau.

La « courbe » pour 16 groupes est représentée par un point, car il n'est pas possible d'avoir plusieurs groupes sur un même nœud. L'*hyper-threading* est intéressant pour ce mode de fonctionnement, car les meilleurs temps ont été obtenus lorsqu'il était activé, c'est-à-dire en fonctionnant avec 8 threads ou plus par nœud. La configuration donnant les meilleures performances sur un nœud se montre sensiblement plus rapide que le temps d'exécution extrapolé du programme original sur une de ces machines. Dans ce cas, la machine n'est pas

surchargée comme c'était le cas avec Intercell, et le meilleur temps d'exécution est obtenu en exécutant 8 processus MPI sur ce nœud.

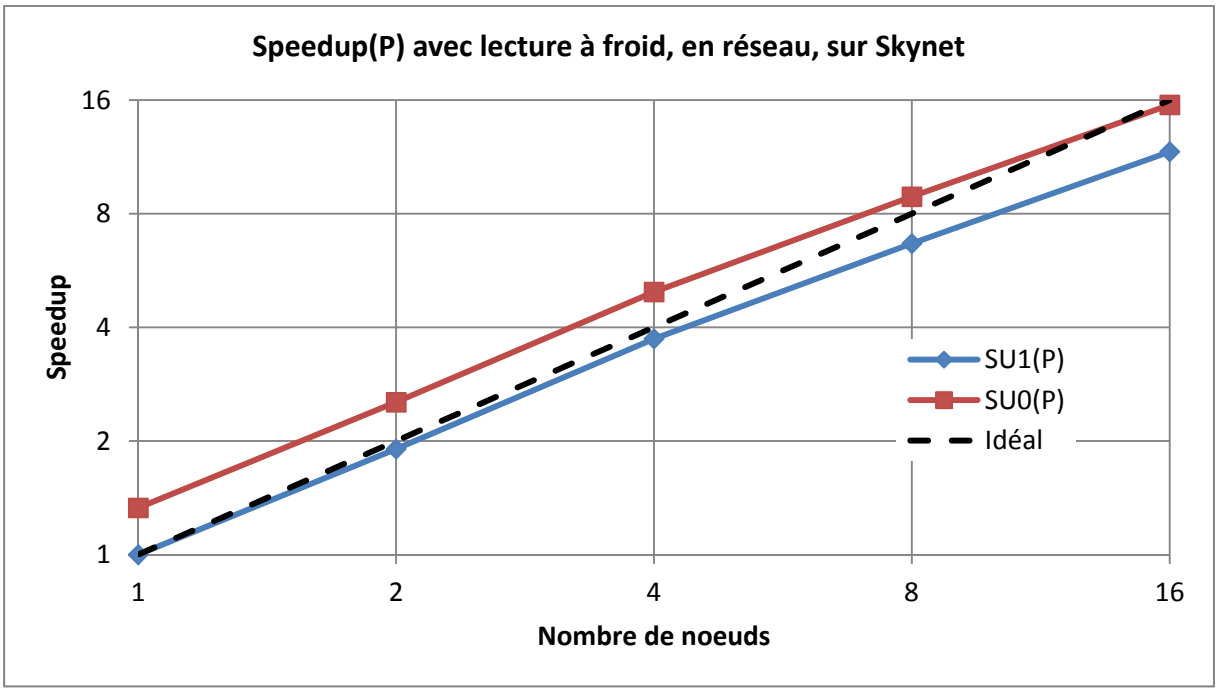


Figure 5.16 : speedup sur Skynet, à froid, en réseau.

La courbe SU1(P), c'est-à-dire le *speedup* utilisant comme référence le temps d'exécution du programme distribué sur un nœud se montre plus habituelle, restant proche de l'idéal, mais étant en permanence en dessous.

SU0(P) commence bien au-dessus de l'idéal, car la référence utilisée pour son calcul est le temps obtenu par le programme original. Cela s'explique par le fait que, sur Skynet, le programme distribué, même sur un nœud, fonctionne bien et donne un temps d'exécution inférieur au programme original.

Globalement, le *speedup* est bon, car la courbe n'a pas encore atteint son maximum avec seulement 16 nœuds et reste relativement proche de l'idéal.

b. Les performances à chaud, en réseau

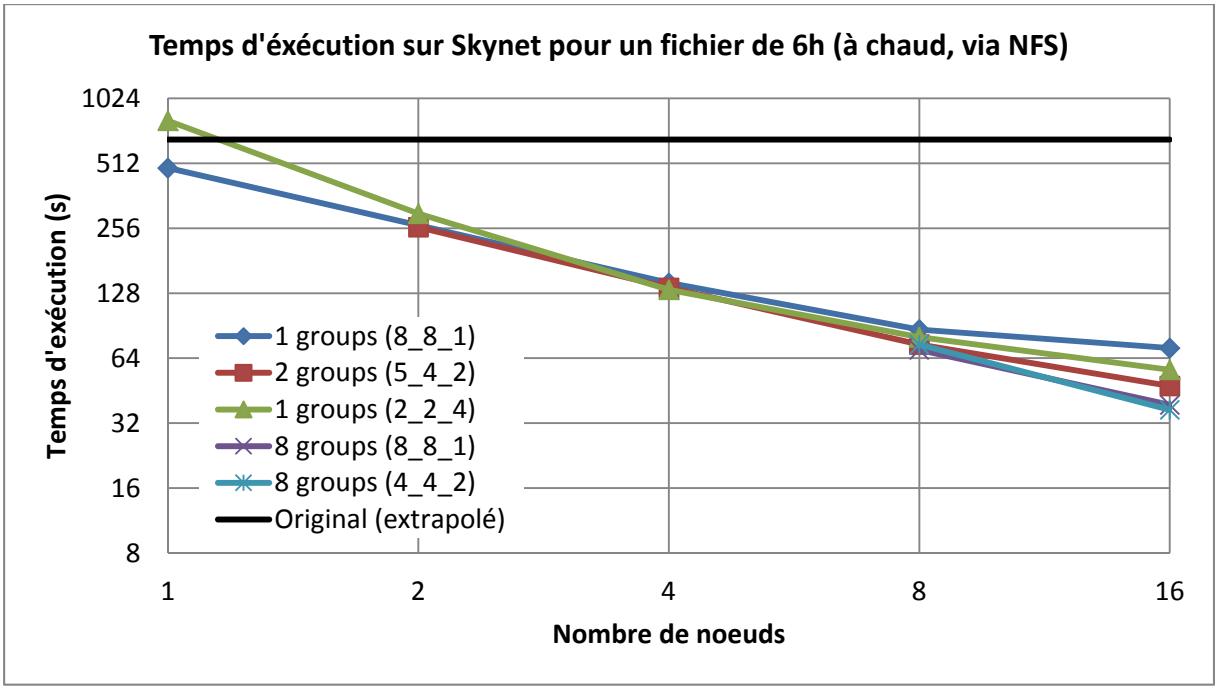


Figure 5.17 : benchmarks du temps d'exécution sur Skynet, à chaud, en réseau.

Les meilleurs temps obtenus ici ont également été réalisés par des configurations exploitant l'*hyper-threading*. Comme précédemment, sur un nœud, il est plus intéressant de fonctionner avec 8 processus MPI et de ne pas surcharger la machine. L'enveloppe inférieure est relativement linéaire, synonyme d'un apport intéressant de la distribution. On dit que la distribution est extensible ou « *scalable* » car la progression des performances est régulière.

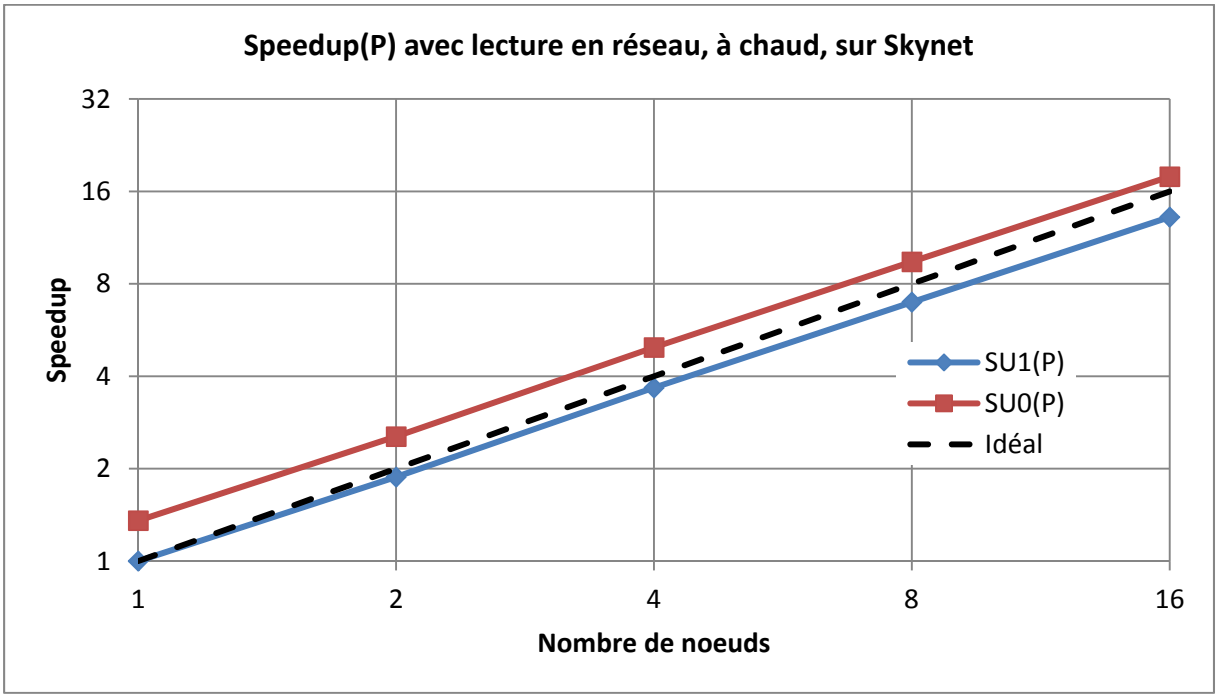


Figure 5.18 : speedup sur Skynet, à chaud, en réseau.

Le *speedup* est relativement linéaire, proche de l'idéal, ce qui confirme une bonne distribution de ce programme sur ce cluster, du moins jusqu'à 16 nœuds.

c. Les performances à froid, en local

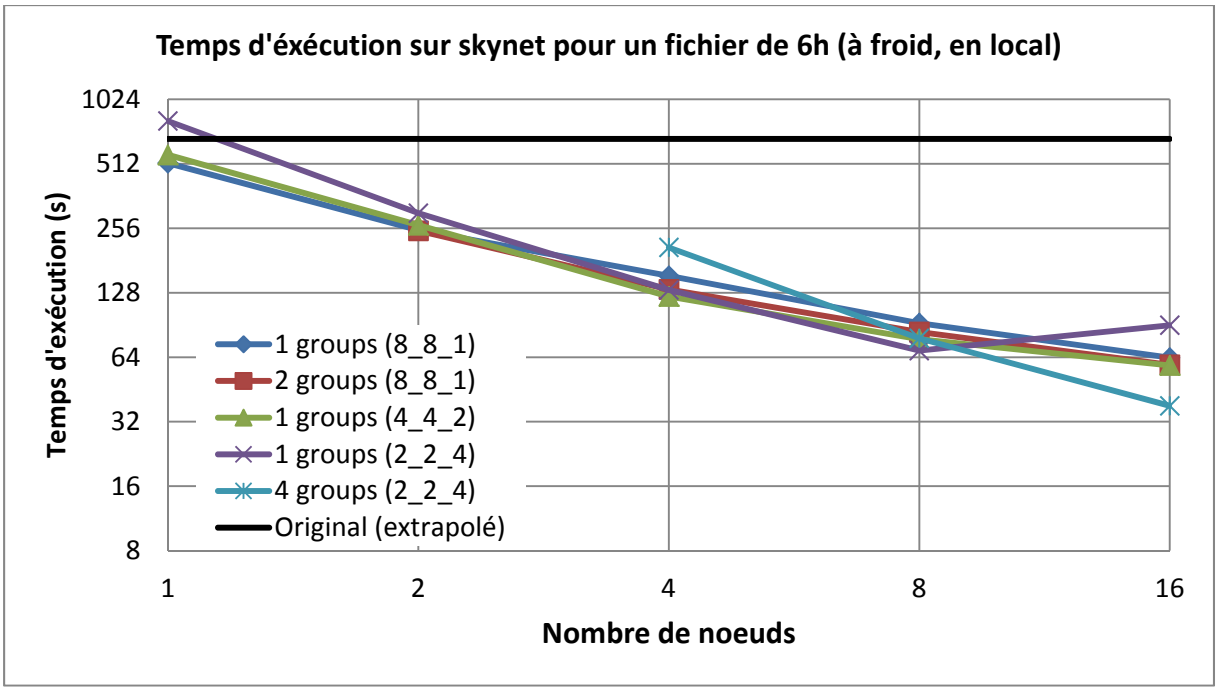


Figure 5.19 : benchmarks du temps d'exécution sur Skynet, à froid, en local.

Les performances ainsi que l'extensibilité obtenues avec ce mode de fonctionnement sont légèrement meilleures qu'à froid en réseau et ont également été réalisées grâce à l'*hyper-threading*. Encore une fois, les meilleures performances ont été obtenues sur un nœud avec l'utilisation de 8 processus MPI.

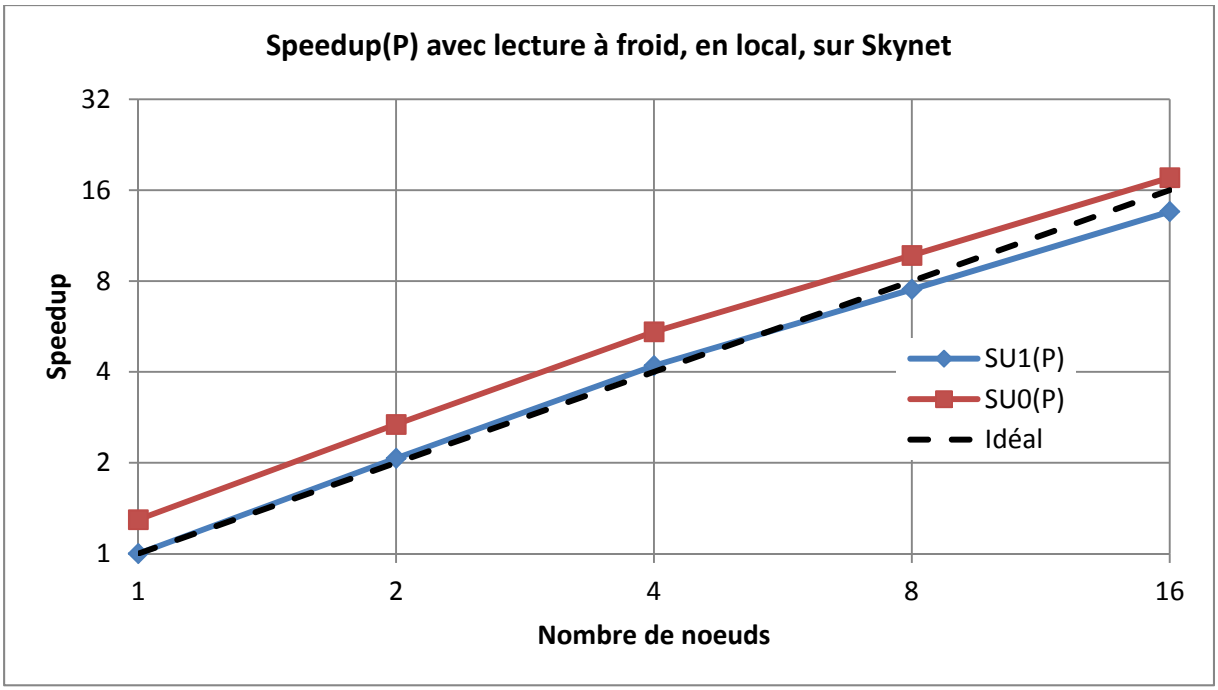


Figure 5.20 : speedup sur Skynet, à froid, en local.

Le *speedup* est un peu meilleur qu'avec des I/O en réseau, mais on observe un léger essoufflement dès l'utilisation de plus de 4 nœuds. Il reste malgré tout intéressant de doubler encore le nombre de machines si possible.

d. Les performances à chaud, en local

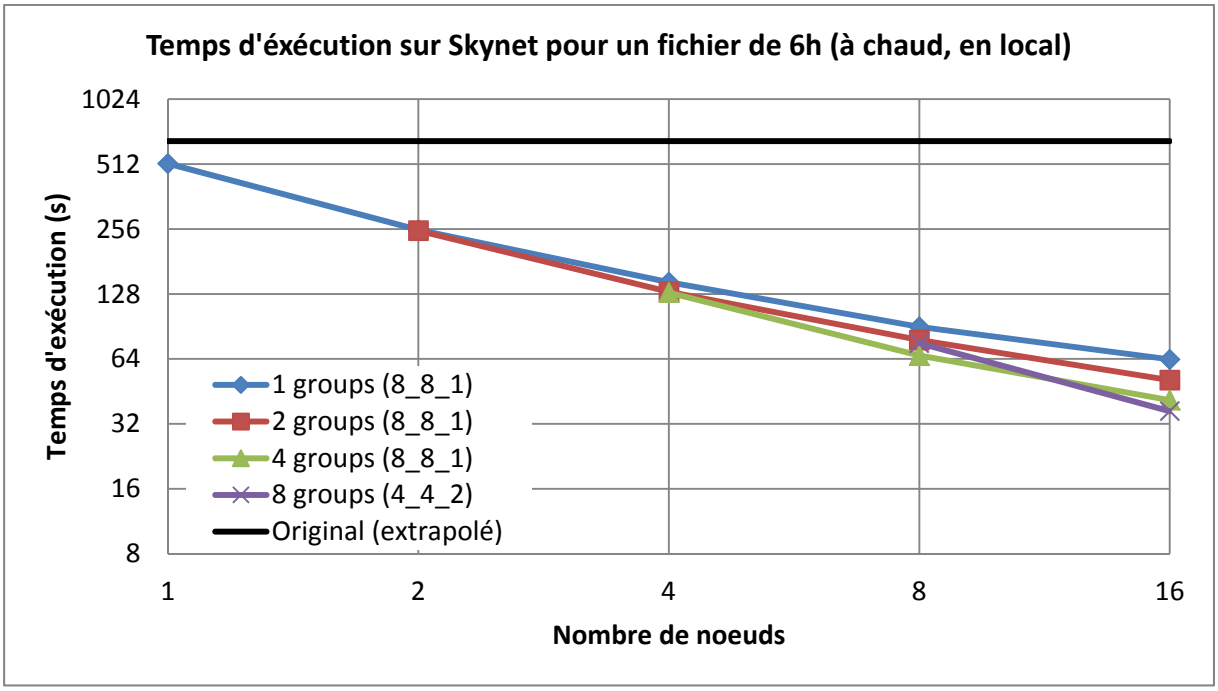


Figure 5.21 : benchmarks du temps d'exécution sur Skynet, à chaud, en local.

Encore une fois, l'*hyper-threading* se montre nécessaire à l'obtention des meilleures performances. Pour avoir le temps d'exécution le plus court possible sur un nœud, il est une fois de plus nécessaire de fonctionner avec 8 processus MPI. On constate également que le gain de la distribution est très intéressant, divisant par près de deux le temps de traitement au fur et à mesure que l'on double le nombre de nœuds.

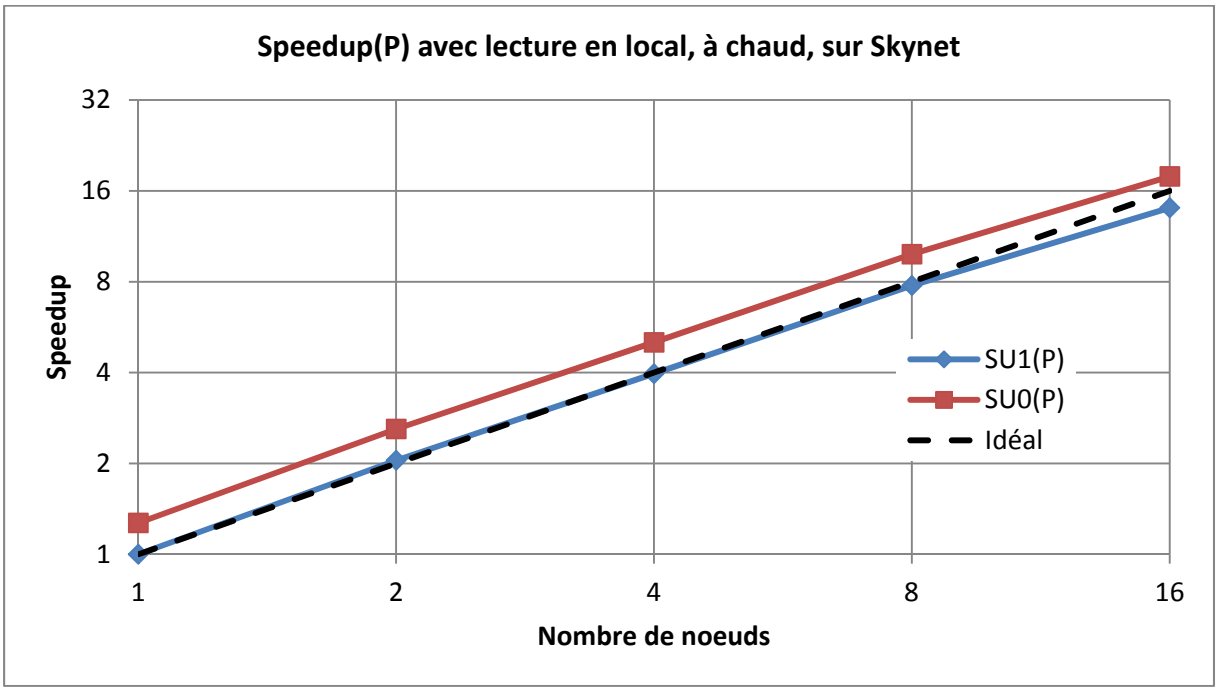


Figure 5.22 : speedup sur Skynet, à chaud, en local.

Le *speedup* confirme l'analyse précédente malgré une légère baisse à partir de 16 nœuds.

e. Synthèse pour Skynet

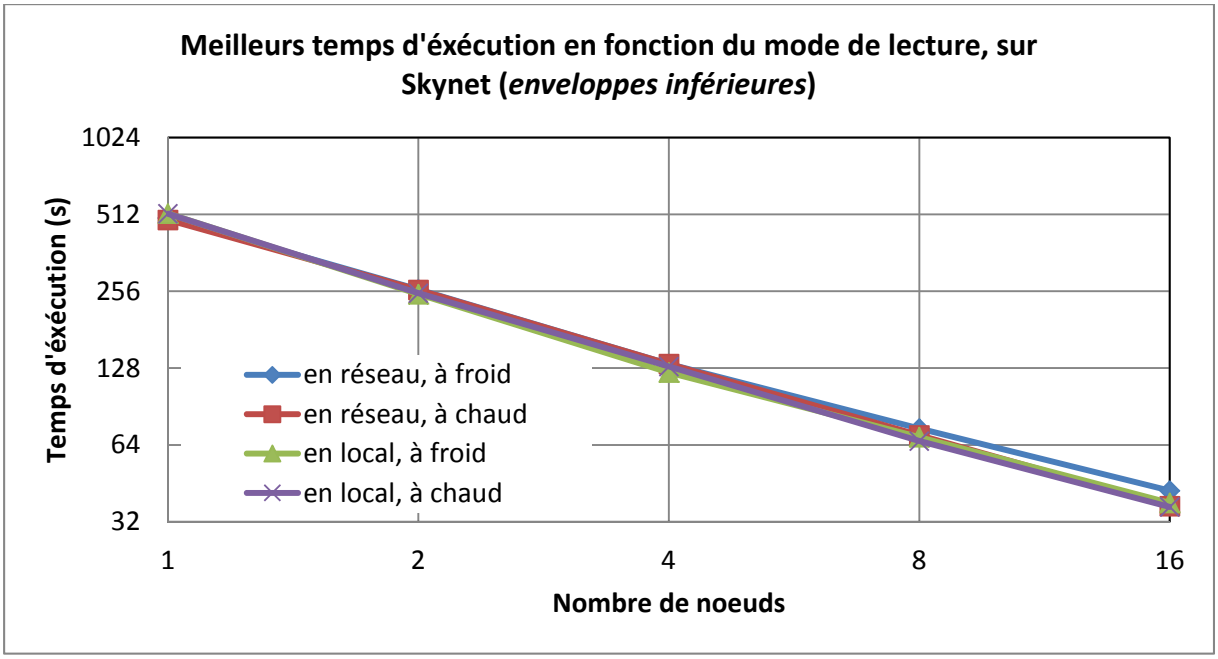


Figure 5.23 : meilleurs temps d'exécution sur Skynet.

On remarque avec ce cluster une nette amélioration lors du fonctionnement du programme distribué sur un seul nœud, même si le mode de fonctionnement n'est pas optimal dans ce cas.

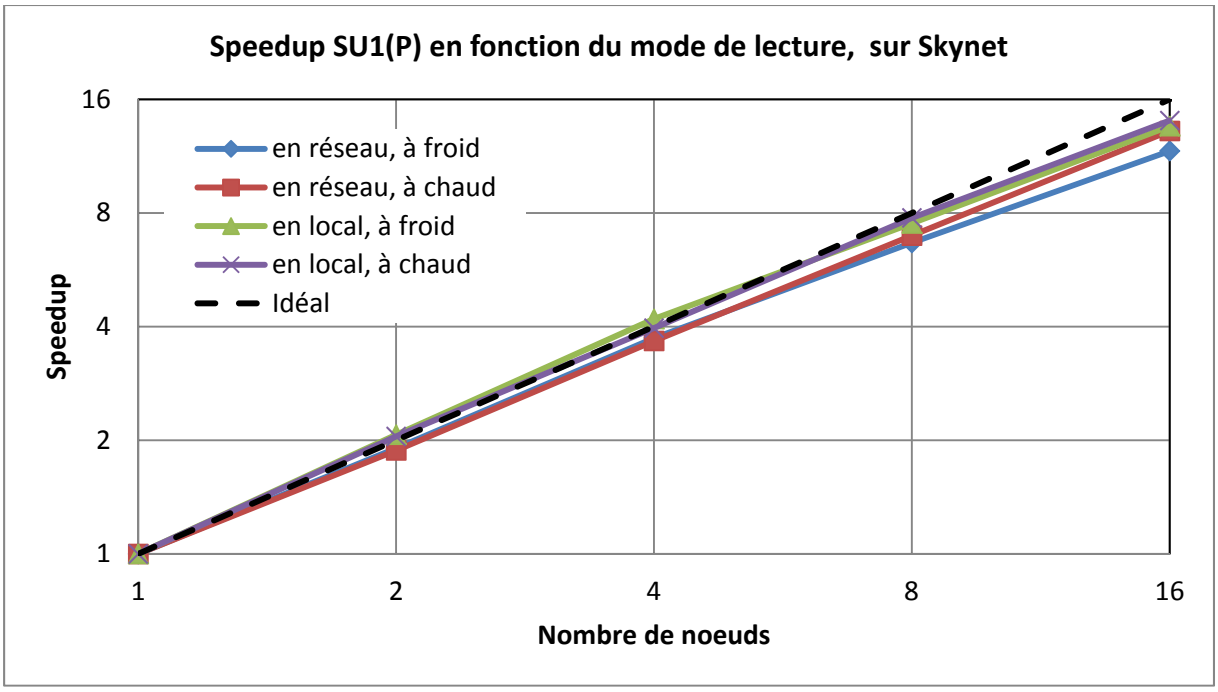


Figure 5.24 : courbes de speedup SU1(P) sur Skynet

On note ici un léger impact des caches disques avec une lecture en réseau sur 16 nœuds.

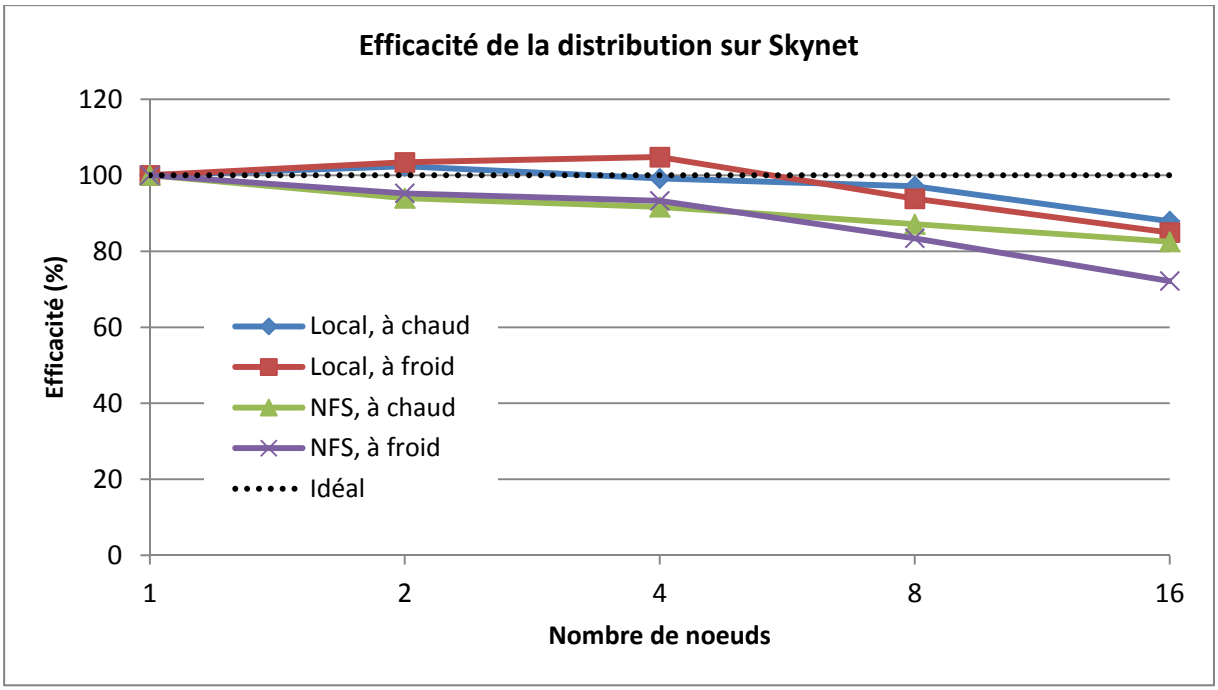


Figure 5.25 : efficacité de la distribution sur Skynet.

Au vu de l'ensemble des courbes, on peut s'attendre à une réduction intéressante du temps de traitement en doublant encore une fois le nombre de nœud, mais si la tendance se justifie, ce gain devrait être moins important que pour les points précédents. Il est dommage de n'avoir à disposition que 16 nœuds car, si l'*hyper-threading* se montre indispensable pour les tests que nous avons effectués jusqu'alors, cela n'est pas nécessairement le cas avec davantage de machines.

3. Synthèse globale

On peut remarquer sur Skynet une forte réduction du temps de traitement avec la version distribuée sur un seul nœud par rapport au temps réalisé par le programme original. Cela ne s'est pas produit sur Intercell car la version distribuée ne commençait à être intéressante qu'à partir deux nœuds. Le mode de fonctionnement sur un nœud est le même : un processus *reader* envoie les données à traiter aux processus *worker* qui sont exécutés sur la même machine. Les communications sont donc effectuées par MPI de manière locale. Une meilleure gestion des accès mémoires simultanés ou une meilleure coexistence des calculs et des entrées/sorties sur un même nœud avec une architecture « Nehalem » sont des hypothèses possible pour comprendre ce phénomène.

Il est possible de remplir un tableau permettant de classer les meilleurs temps obtenus sur les deux clusters en fonction du mode de lecture choisi :

	A froid		**A chaud**	
En local	Intercell	Skynet	**Intercell**	Skynet
	10.8s avec 256 nœuds	37.9s avec 16 nœuds	**9.1s avec 256 nœuds**	36.6s avec 16 nœuds
En réseau	Intercell	**Skynet**	Intercell	Skynet
	43.5s avec 64 nœuds	**42.4s avec 16 nœuds**	9.9s avec 256 nœuds	36.9s avec 16 nœuds

Ce tableau nous permet, en gardant à l'esprit les deux principaux cas d'utilisation, de conclure que:

- Le chercheur en traitement de signal travaillant sur de nouvelles fonctionnalités aura plus d'avantages à utiliser les 256 nœuds d'Intercell avec une lecture en local, en ayant au préalable déployé sa base de sons de tests sur toutes les machines. Cela lui permet de tester, rapidement et sans surcharger le réseau, des modifications dans le programme ou de nouveaux algorithmes (*accès à chaud*).

- L'exploitant, traitant des fichiers toujours différents, aura tout intérêt à utiliser Skynet en lisant les fichiers via le réseau car les performances à froid avec les 16 machines de ce cluster sont plus intéressantes que sur Intercell, quel que soit le nombre de nœuds. Cela est plus pertinent pour cet utilisateur, car déployer un son qui ne sera traité qu'une seule fois sur un grand nombre de machines est trop couteux en temps (*accès à froid*).

On constate également qu'en réseau avec une lecture à froid les performances sont quasiment similaires entre Skynet et Intercell. Dans les deux cas ces résultats ont été obtenus avec l'utilisation de 128 cœurs, 64×2 dans le cas d'Intercell, et 16×8 dans le cas de Skynet.

Chapitre 6

Conception d'un outil de déploiement

L'ensemble des mesures effectuées montrent qu'il n'existe pas une seule configuration qui fonctionne mieux que toutes les autres dans tous les cas. La configuration à adopter dépend de plusieurs facteurs :

- l'architecture du cluster ;
- le nombre de nœuds utilisés pour le traitement ;
- le mode de lecture (local ou réseau, à chaud ou à froid).

L'idée derrière la conception d'un « outil de déploiement » est de permettre à la personne utilisant le programme de l'exécuter facilement avec la meilleure configuration possible en fonction de ces facteurs.

1. Utilisation d'un cluster

Afin d'éviter l'anarchie lors de l'utilisation de fractions du cluster par différentes personnes, il est nécessaire d'utiliser un allocateur-ordonnanceur. Celui-ci permet aux utilisateurs de faire une demande de nœuds et d'être assuré que personne d'autre ne travaillera sur ceux-ci pendant la période où ils sont réservés. Pour ses clusters, Supélec utilise OAR qui a été créé par des chercheurs et ingénieurs de la communauté GRID'5000, notamment par des membres du laboratoire appelé aujourd'hui « Laboratoire Informatique de Grenoble », et est activement maintenu depuis par divers établissements publics tels que le CNRS et l'INRIA. OAR permet aux utilisateurs d'obtenir des nœuds en interactif, pour faire des tests et du débogage ou en *batch* afin de lancer de gros travaux, la nuit ou le weekend par exemple.

a. Réservation de nœuds

Pour Intercell, il est nécessaire de se connecter à un des serveurs frontaux disposant d'OAR, comme « vera.ic ». Il est alors possible d'entrer des commandes OAR telles que :

```
oarsub -l nodes=200,walltime=10:00:00 –I
```

Cette commande permet l'allocation de 100 machines pour 10 heures en mode interactif. Sur Intercell, pour les besoins d'un projet passé, *oarsub* attribue des cœurs de calcul plutôt que des nœuds, ce qui explique qu'il faille demander 200 « *nodes* » pour se voir attribuer en

réalité 100 machines bi-cœur. La configuration d'OAR devrait être revue à l'occasion d'un upgrade de ce cluster.

Si tout se passe bien, la commande nous connecte sur un des nœuds alloués et s'il n'y a pas assez de ressources disponibles nous sommes mis en attente.

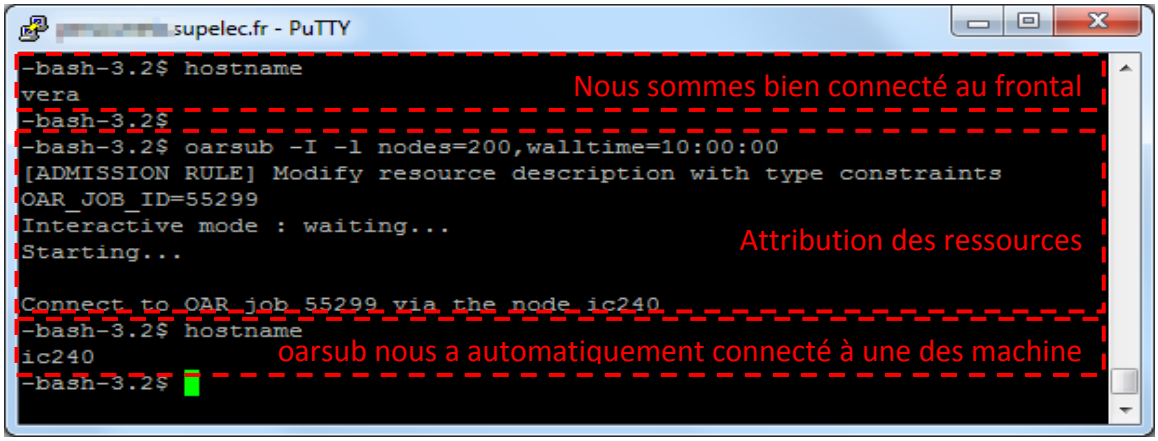


Figure 6.1 : exemple d'attribution de 100 noeuds sur Intercell en interactif.

Pour Skynet la commande est presque la même mais le serveur frontal est « gserver.grid » et il faut bien demander le nombre de nœuds exact que l'on souhaite obtenir.

```
oarsub -p "cluster='Skynet'" -l nodes=16,walltime=8:00:00 –I
```

Il est important ici de spécifier le cluster sur lequel on veut se connecter car gserver se charge de l'ordonnancement de plusieurs petits clusters. La commande précédente permet donc d'attribuer 16 nœuds de Skynet pour 8 heures en interactif.

Les nœuds alloués figurent dans un fichier qui est pointé par la variable d'environnement *$OAR_NODEFILE*. Le nom de chaque machine y apparait autant de fois qu'elle a de cœurs logiques. Pour Intercell, chaque machine y sera indiquée à deux reprises, alors que pour Skynet, chacune sera inscrite huit fois.

Pour lancer un batch il suffit de remplacer « -I » par le nom du script à exécuter :

```
oarsub -l nodes=100,walltime=10:00:00 ./mpi_bench_Intercell.pl
```

La commande « oarsub » rend immédiatement la main. Le job sera exécuté jusqu'à la fin du traitement sauf si celui-ci dure plus de 10 heures, auquel cas il sera tué brutalement.

b. Suivi et suppression de jobs

Il est possible à tout moment d'avoir des informations sur l'utilisation du cluster. La commande « oarstat » exécutée sur le frontal permet d'afficher ces informations :

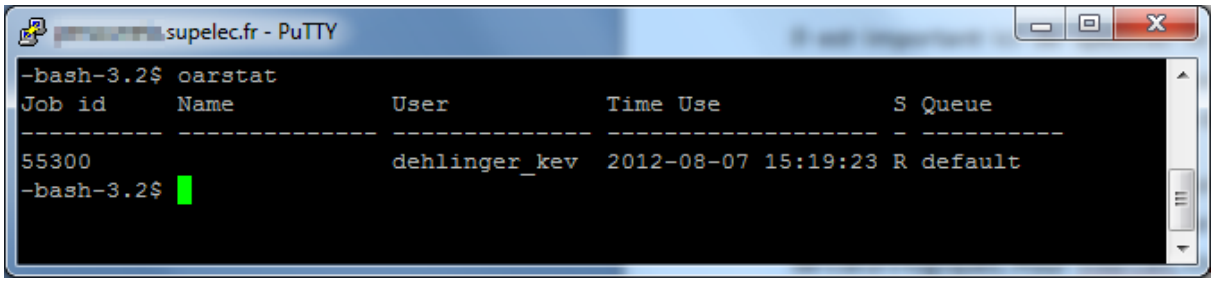


Figure 6.2 : résultat d'exécution de la commande « oarstat ».

Cela permet de voir le nombre de sessions, les utilisateurs, la date de début, l'état, etc. Malheureusement cette vue synthétique ne permet pas de connaitre le nombre de nœuds alloués pour chaque job. Il faut alors recourir à la commande « oarstat –f » qui est bien plus verbeuse, allant même jusqu'à lister les ressources allouées à chaque travaux.
La commande « oardel 55300 » permet d'arrêter et supprimer le job 55300 en cas de problème. A noter que cette commande ne permet de supprimer que les jobs dont on est propriétaire, à moins d'être root.

2. Fonctionnement sans outil de déploiement

L'exécution de l'application distribuée de classification de sources sonores sur les nœuds souhaités est réalisée grâce à la commande « mpirun » à laquelle on indique au lancement, le nombre de processus à exécuter grâce à l'argument `-np <nombre de machines>`. Cette commande s'appuie ensuite sur un fichier contenant la liste des machines sur lesquelles il faut exécuter les processus. Un mécanisme de *round-robin* est appliqué en parcourant ce fichier autant de fois qu'il est nécessaire pour trouver une machine hôte à tous les processus. Le nom d'un nœud y peut figurer plusieurs fois, si l'on souhaite qu'il reçoive plusieurs processus dans la même « passe » du fichier. Il est également possible réaliser complètement le *mapping* en y spécifiant exactement autant de nom de machines qu'il y a de processus à exécuter.

Il est toutefois absolument nécessaire que ce fichier ne comporte que des noms de machines qui nous ont été attribuées par OAR. La génération du fichier « machines »

Dans le cas de cette application, la génération manuelle du fichier est fastidieuse, car le nombre de processus sur les nœuds lecteurs et sur des nœuds non lecteurs sont des éléments clés des différentes configurations.

Sur Skynet, par exemple, lors de la phase de benchmarks les différentes configurations testées étaient :

Configuration	Nombre de processus sur nœuds lecteur	Nombre de processus sur nœuds non lecteurs
A	2	2
B	3	2
C	4	4
D	5	4
E	8	8
F	9	8

Dans le cas de la configuration B (3 processus sur les nœuds lecteurs et 2 sur les nœuds non lecteurs) pour 4 nœuds au total avec 2 lecteurs, le fichier serait :

```
sh01
sh01
sh01
sh02
sh02
sh03
sh03
sh03
sh04
sh04
```

Trois processus sur les nœuds lecteurs (un *reader* et deux *worker*)

Deux processus *worker* sur les nœuds non lecteurs

Dans certains cas, on peut également être amené à faire du partage de nœud entre deux groupes pour optimiser l'utilisation des machines. Utiliser 3 nœuds et deux groupes sur un cluster de machines *dual-core* donnerait le fichier machine suivant :

```
ic01
ic01
ic02
ic03
ic03
ic02
```

Groupe 1

Groupe 2

Dans cet exemple relativement simple la machine « ic02 » est partagée car elle exécutera un processus *worker* pour chacun des deux groupes.

L'ordre des nœuds dans le fichier détermine le numéro de processus MPI qui est utilisé pour la formation des groupes. Un ordre cohérent est une chose importante pour les

performances. Expérimentalement il est apparu que pour s'assurer d'avoir le temps de traitement le plus rapide il faut :

- ne pas partager de nœuds entre plusieurs groupes si ce n'est pas nécessaire ;
- veiller à ne pas avoir de *worker* et *reader* de groupes différents sur le même nœud ;
- veiller à ne pas avoir plusieurs *reader* sur le même nœud ;
- veiller à ce que le fichier contienne une ligne par processus que l'on souhaite créer, sinon cela enclenche le mécanisme de *round-robin* pour trouver un hôte à tous les processus. Dans notre cas cela peut s'avérer pénalisant car il n'est alors pas possible de s'assurer que les trois règles précédentes soient appliquées.

a. L'exécution du traitement

En supposant que nous disposions de quatre machines *quad-core* avec *hyper-threading* du cluster Skynet, et que la configuration d'exécution désirée est la suivante :

- deux groupes de lecture ;
- trois processus MPI sur les nœuds lecteurs (un *reader*, et deux *worker*) ;
- deux processus MPI sur les autres nœuds ;
- quatre threads OpenMP.

La commande permettant d'exécuter le traitement avec la configuration voulue est :

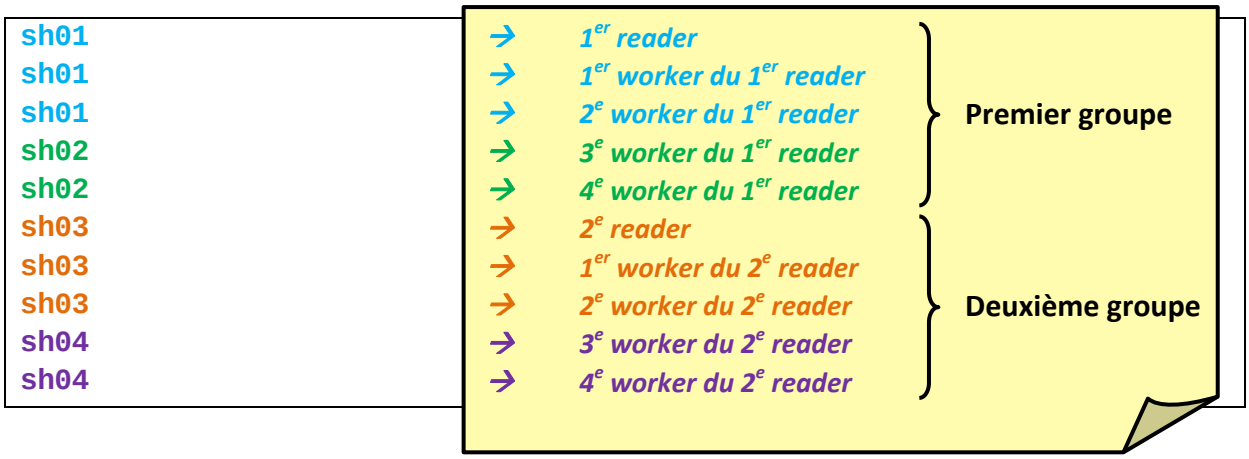
```
mpirun -np 10 -machinefile mac.txt sources -L2 -W4 -p4 -S3h.wav
```

mpirun	La commande *mpirun* permet d'exécuter les processus sur le cluster
-np 10	Avec 10 processus (2 groupes x(3 process sur nœud *reader* + 2 process))
-machinefile mac.txt	Le fichier machine est spécifié grâce à l'argument *–machinefile*
sources	sources est le nom de notre exécutable de classification de sources sonores.
-L2	Deux groupes
-W4	Quatre processus *worker* par groupe
-p4	Quatre threads OpenMP par processus MPI
-S3h.wav	3h.wav est le fichier audio à traiter

Les éléments en rouges correspondent aux paramètres passés à la commande mpirun, alors que ceux en vert correspondent à ceux passés à l'exécutable « sources » à chaque exécution.

L'indication que les nœuds exécutant un *reader* doivent lancer trois processus n'est pas une information que l'on peut passer en ligne de commande, mais dépend bien de la composition du fichier machine que l'on passera à *mpirun*.

Voici le fichier machine correspondant à notre exemple :



3. Besoin d'un déployeur automatique

En gardant à l'esprit les besoins des différents utilisateurs types, il n'est pas imaginable de ne pas simplifier ces longues étapes. La création du fichier machines est une étape réellement fastidieuse pour laquelle la moindre erreur peut impacter les performances.

Les besoins que l'on voit apparaitre pour cet outil sont :

- la simplicité d'utilisation ;
- les possibilités de paramétrage étendues ;
- la génération du fichier machines ;
- la génération de la ligne de commande adéquate et l'exécution.

Pour répondre à ces besoins l'outil de déploiement devra proposer un mode heuristique permettant à une personne ne connaissant pas l'architecture du cluster ou le programme de taper une ligne de commande générique tout en pouvant s'attendre à ce que la configuration utilisée permette d'obtenir le meilleur temps de traitement possible. L'utilisateur doit également avoir la possibilité d'intervenir sur certains paramètres tout en laissant certaines décisions au déployeur pour finaliser la configuration. Un utilisateur avancé peut souhaiter ne laisser aucun choix à l'outil en spécifiant tous les arguments nécessaire et simplement s'en servir pour générer le fichier. Le dernier cas à prendre en compte est la possibilité de spécifier complètement la formation des groupes, en définissant le nombre de processus par nœud, machine par machine afin de faire des tests ou des benchmarks.

L'outil doit également offrir la possibilité de lancer plusieurs fois le même traitement d'affilée, et proposer un fonctionnement de type « bench » formatant les données de sortie de manière à être facilement importées dans un tableur.

a. Choix du langage de programmation

Le langage utilisé pour développer le déployeur est Perl. Celui-ci est activement maintenu et livré en standard dans la majorité des distributions linux. Perl d'ailleurs est le langage qui est utilisé pour OAR.

Ses grandes forces :

- l'intégration à l'environnement ;
- un moteur d'expressions régulières natif très puissant ;
- une grande communauté ;
- des modules livrés par défaut très puissants ;
- un code compact et rapide à développer.

Le déployeur doit être capable d'analyser le fichier des ressources fourni par OAR et les fichiers systèmes linux pour permettre de connaitre le nombre de cœurs dans la machine et si l'*hyper-threading* est activé. Il doit pour cela accéder facilement aux variables d'environnement. Ces raisons, plus le fait que Perl est installé par défaut sur l'ensemble des machines du cluster en font le langage de script parfait pour le développement du déployeur.

b. Mode d'emploi

Le déployeur doit être le même dans tous les cas d'utilisations possibles et répondre aux attentes des différents utilisateurs. Pour cela des arguments différents sont à la disposition des utilisateurs :

-heu <type>	Indique le type « d'heuristique » à utiliser. Pour l'instant seul le type « 0 » est implémenté.
-hf <fichier de paramétrage d'heuristique>	Indique quel fichier de paramétrage d'heuristique utiliser car la meilleure configuration dépend de l'architecture du cluster et du mode de fonctionnement.
-ctr <type>	Indique le type de « mode contrôlé » à utiliser.
-gf <formation des groupes>	Permet de forcer la formation des groupes.

-m <nombre de process par nœud>	Indique le nombre de processus MPI par nœud.
-mrn <nombre de process par nœud lecteur>	Indique le nombre de processus MPI par nœud lecteur.
-o <nombre de thread >	Indique le nombre de thread OpenMP à utiliser dans chaque processus *worker*.
-npg <nombre de nœuds par groupe>	Spécifie la formation d'un groupe en nombre de nœuds.
-g <nombre de groupes>	Permet d'indiquer le nombre de groupes.
-w <nombre de *worker* par groupe>	Indique le nombre de processus *worker* dans un groupe.
-n <nombre de nœuds max>	Permet de limiter le nombre de nœuds à utiliser. Par exemple si 200 machines sont allouées dans la session OAR et que l'on souhaite tester sur 100 nœuds seulement.
-p <programme>	Spécifie le programme à exécuter sur le cluster.
-s <fichier à analyser>	Indique le fichier audio à analyser.
-b	Exécute en mode bench.
-printonly <niveau d'information>	Permet d'afficher des informations sur la configuration sans exécuter le programme.
-loop <nombre de boucles>	Exécute le traitement plusieurs fois d'affilée.

Tous ces arguments ne sont pas compatibles entre eux. Ils dépendent du mode de fonctionnement choisi : « heuristique » ou « contrôlé » et des autres arguments déjà positionnés.

Le mode « heuristique 0 » :

Fonctionnement dynamique : Le nombre de processus et de threads sont déduits à partir de la configuration de la machine s'ils ne sont pas spécifiés. Dans ce cas, on fonctionnera avec un processus par cœur physique et on utilisera en plus deux threads par processus si l'*hyper-threading* est activé sur les machines. L'outil choisira de « tasser », c'est-à-dire de remplir au maximum les ressources disponibles en plaçant, si nécessaire, plusieurs *worker* de groupes différents sur le même nœud. Le nombre de nœuds par groupe et le nombre de *worker* par groupe sont calculés s'ils ne sont pas spécifiés

Syntaxe possible du déploiement avec le mode heuristique 0 :

```
./launcher5.pl -p <programme> -s <fichier à traiter> -heu 0
-g <nombre de groupes>
[-m <processus MPI par nœud>
     [-o <nombre de thread OpenMP>]]
[-mrn <processus MPI par nœud lecteur>]
[-npg <nombre de nœuds par groupe>]
[-w <nombre de worker par groupe>]
[-b]
[-printonly <niveau d'information>]
[-loop <nombre de boucles>]
[-n <nombre de nœuds max>]
```

Au plus un des deux

Fonctionnement statique : Le nombre de groupes, de processus et de threads sont extraits du fichier heuristique s'ils ne sont pas spécifiés. S'il n'y a pas de configuration utilisant exactement le même nombre de nœuds dans le fichier de configuration, celle avec le nombre de nœuds directement inferieur sera utilisée. Le nombre de nœuds par groupe et le nombre de *worker* par groupe sont calculés s'ils ne sont pas spécifiés.

Autre syntaxe possible du déploiement en heuristique 0 :

```
./launcher5.pl -p <programme> -s <fichier à traiter> -heu 0
-hf <fichier de paramétrage d'heuristique>
[-g <nombre de groupes>
     [-m <processus MPI par nœud>
          [-o <nombre de thread OpenMP>]]
     [-mrn <processus MPI par nœud lecteur>]
     [-npg <nombre de nœuds par groupe>]
     [-w <nombre de worker par groupe>]
     ]
[-b]
[-printonly <niveau d'information>]
[-loop <nombre de boucles>]
[-n <nombre de nœuds max>]
```

Au plus un des deux pour pouvoir utiliser -m ou -mrn

A l'heure actuelle, il y a quatre fichiers de paramétrage d'heuristique par cluster, correspondant à des fonctionnements en :

- lecture en local à froid ;
- lecture en local à chaud ;
- lecture en réseau à froid ;
- lecture en réseau à chaud.

Ces fichiers contiennent les meilleures configurations obtenues après benchmarks d'un maximum de solutions possibles en fonction du nombre de nœuds disponibles. La granularité du nombre de nœuds dans ces fichiers est en 2^n, mais il est toutefois possible de l'affiner en réalisant les benchmarks nécessaires aux configurations intermédiaires.

Si on veut exécuter le traitement une fois sur 50 nœuds à partir d'un fichier en réseau, il suffit de lancer la commande suivante :

```
./launcher5.pl -p sources -s fichier.wav -n 50 -heu 0
     -hf Intercell_nfs_coldstart.csv
```

Concrètement, la configuration choisie est la meilleure pour 32 nœuds, car c'est celle figurant dans le fichier de configuration d'heuristique avec le nombre de nœuds directement inferieur :

- quatre groupes ;
- un processus MPI par nœud lecteur (dans ce cas, 1 *reader* et 0 *worker*);
- un processus MPI par autre nœud ;
- deux threads OpenMP par processus.

Cette configuration ne permet pas une exécution sur 50 nœuds car les groupes ne seraient pas homogènes, seul 48 nœuds seront donc utilisés.

Les arguments passés au déployeur en mode « heuristique 0 » doivent être considérés comme des suggestions de l'utilisateur. S'il est possible de trouver une configuration cohérente ce sera fait, sinon de légères adaptations sont effectuées automatiquement.

Pour être sûr de pouvoir exécuter le traitement avec la configuration exacte que l'on souhaite, il est nécessaire d'utiliser un des modes contrôlés.

Format des fichiers d'heuristique :

Le fichier pour un fonctionnement en local à froid sur le cluster Intercell est le suivant :

```
#nodes;#groups;#procs/readernode;#procs/othernode;#thread/process
#1 node(s) => 1713.194045 sec
1;1;3;2;1
#2 node(s) => 847.397472 sec
2;1;3;2;1
#4 node(s) => 372.603366 sec
4;1;2;2;1
#8 node(s) => 184.914247 sec
8;1;2;2;1
#16 node(s) => 94.306949 sec
16;4;2;2;1
```

```
#32 node(s) => 48.789998 sec
32;4;1;1;2
#64 node(s) => 28.681404 sec
64;8;1;1;2
#128 node(s) => 16.597963 sec
128;16;1;1;2
#256 node(s) => 10.861733 sec
256;32;1;1;2
```

Les lignes commençant par des # sont considérées comme des commentaires et sont ignorées. Elles permettent toutefois à l'utilisateur de mieux comprendre le format et éventuellement de créer ses propres fichiers ou de compléter une heuristique existante avec ses propres retours d'expérience.

Les données utiles présentes dans ce fichier sont séparées par des points-virgules et représentent :

1. le nombre de nœuds pour la configuration
2. le nombre de groupes
3. le nombre de processus sur un nœud ayant un processus lecteur
4. le nombre de processus sur les autres nœuds
5. le nombre de thread OpenMP par processus

Ainsi, dans l'exemple précédent, avec 128 nœuds la configuration adoptée serait 16 groupes, avec un processus par nœud ayant un lecteur, idem pour les autres machines, et deux threads par processus.

Le mode « contrôlé 0 » :

Il permet à l'utilisateur de spécifier exactement la configuration à adopter.

Syntaxe possible du déploiement avec le mode contrôlé 0 :

```
./launcher5.pl -p <programme> -s <fichier à traiter> -ctr 0
-npg <nombre de nœuds par groupe>
-m <processus MPI par nœud>
-o <nombre de thread OpenMP>
[-mrn <processus MPI par nœud lecteur>]
[-b]
[-printonly <niveau d'information>]
[-loop <nombre de boucles>]
[-n <nombre de nœuds max>]
```

Il s'agit du mode qui fut utilisé pour les benchmarks car il permet de s'assurer que la configuration souhaitée est effectivement celle qui sera adoptée par le déployeur.

Exemple d'utilisation du mode contrôlé 0 :

```
./launcher5.pl -p sources -s fichier.wav -ctr 0 -npg 4 -m 2 –mrn 1 -o 1
```

Ici, un groupe est composé de quatre nœuds et formé de la façon suivante :

- un processus sur le nœud lecteur ;
- deux processus sur les autres nœuds ;
- chaque processus *worker* est composé de deux threads OpenMP.

Le mode « contrôlé 1 » :

Ce mode offre les mêmes possibilités que le mode contrôlé 0 mais permet en plus de spécifier la formation exacte des groupes en indiquant le nombre de processus MPI à exécuter sur chacun des nœuds du groupe. On peut alors spécifier une configuration différente sur chaque machine.

Syntaxe possible du déploiement avec le mode contrôlé 1 :

```
./launcher5.pl -p <programme> -s <fichier à traiter> -ctr 1
-gf <formation d'un groupe>
-o <nombre de thread OpenMP>
[-b]
[-printonly <niveau d'information>]
[-loop <nombre de boucles>]
[-n <nombre de nœuds max>]
```

Ce mode permet de forcer la composition des groupes. On peut dire que le premier nœud d'un groupe exécute cinq processus, alors que le suivant seulement deux, celui d'après quatre, etc.

Exemple d'utilisation du mode contrôlé 1 :

```
./launcher5.pl -p sources -s fichier.wav -ctr 1 -o 2 -gf 1:2:3:4:5
```

Ici, un groupe est composé de cinq nœuds et formé de la façon suivante :

- un processus sur le premier nœud (le nœud lecteur) ;
- deux processus sur le deuxième nœud ;
- trois processus sur le troisième nœud ;
- quatre processus sur le quatrième nœud ;
- cinq processus sur le cinquième nœud ;
- chaque processus *worker* est composé de deux threads OpenMP.

Comme « –n » n'est pas spécifié, le déployeur va faire le plus de groupes de cinq nœuds possible avec toutes les machines allouées par OAR.

c. Problèmes de déploiement et solutions

Gestion des nœuds lents ou à problèmes :

Il est possible, lors d'un déploiement sur un grand nombre de machines qu'un des nœuds ait un problème et ralentisse les autres à cause d'étapes de synchronisation nécessaires au bon fonctionnement de l'application. Malheureusement, celle-ci ne permet pas de détecter le ou les nœuds fautifs. Une fois identifiés il est possible de les bannir de l'exécution grâce au script de déploiement. Pour cela il faut modifier le code source Perl avec un éditeur de texte et ajouter le nom des machines dans le tableau des nœuds inutilisables.

```
my @unusable_computers = ("ic169", "etc", "etc");
```

Ces nœuds, s'ils ont été alloués par OAR, seront automatiquement évités par le déployeur. Pour celui-ci, le nombre de nœuds disponible est égal au nombre de nœuds alloués par OAR en retirant ceux qui sont inutilisables.

Attribution de demi-nœuds par OAR :

Pour s'attribuer des machines sur Intercell, il est nécessaire de demander le double avec la commande « oarsub », car elle alloue sur ce cluster des cœurs de traitement plutôt que des machines (voir page 71). Une erreur étant vite arrivée, il est possible qu'un utilisateur du cluster ait fait une demande d'un nombre impair de ressources, laissant un cœur d'un des nœuds qui lui ont été attribués comme disponible pour OAR. L'utilisateur suivant, souhaitant travailler sur cinq nœuds par exemple demandera 10 ressources et se retrouvera concrètement avec un fichier machine renvoyé par OAR contenant 6 noms différents.

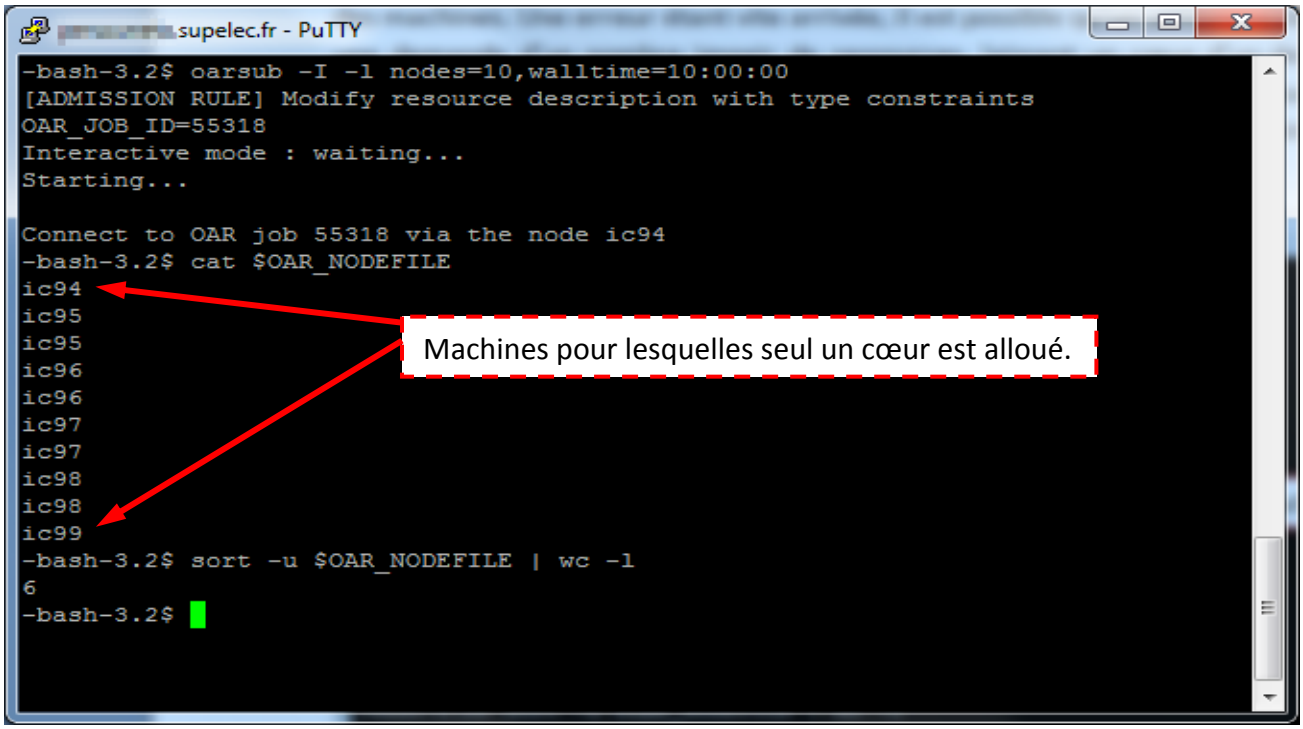


Figure 6.3 : problème d'allocation de demi-nœuds.

La commande suivante permet de connaitre le nombre de noms de machines différentes allouées par OAR :

```
sort -u $OAR_NODEFILE | wc -l
```

Dans cet exemple elle retourne « 6 » alors que nous souhaitions cinq nœuds seulement.

La solution dans ce cas consiste à rendre les nœuds puis demander une seule ressource que l'on n'utilisera pas pour permettre à OAR de refaire des attributions de nœuds complets.

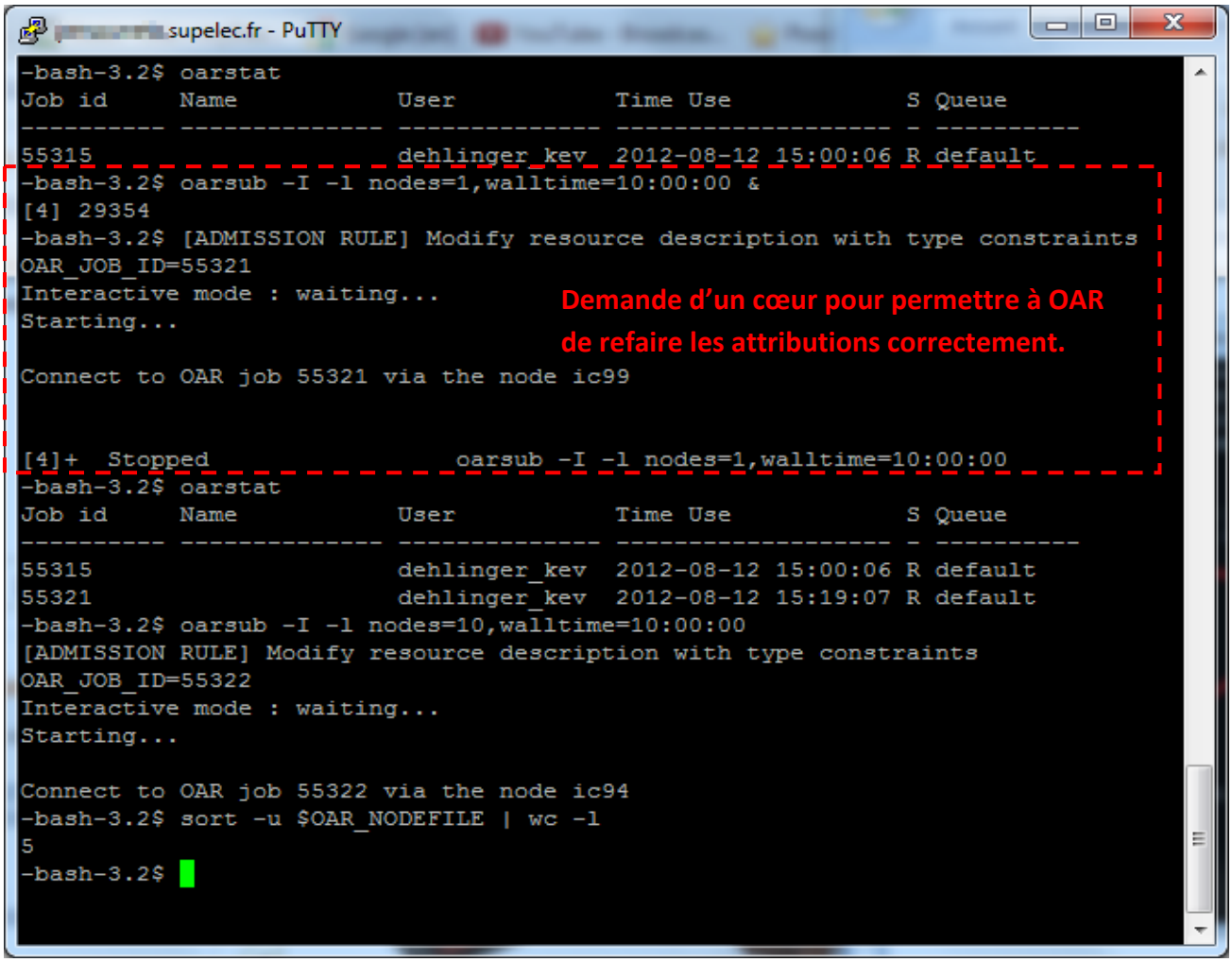


Figure 6.4 : solution en cas d'allocation de demi-nœuds.

Dans cet exemple, l'utilisateur avait demandé pour le job 55315, l'attribution d'un nombre impair de ressources. Le job 55321 pour lequel on a volontairement attribué un seul cœur de traitement permet à OAR d'attribuer les nouveaux jobs sur des nœuds complets. Lorsque l'on a souhaité réserver 5 nœuds pour le job 55322 il n'y a pas eu de problème.

Chapitre 7

Conclusions et perspectives

1. Rappel des objectifs

Le travail réalisé pour ce mémoire visait à améliorer un programme de classification de sources sonores développé par un enseignant-chercheur de Supélec. Les résultats obtenus par ce programme étaient bons, mais son utilisation n'en restait pas moins limitée, notamment par la taille maximale des fichiers qu'il pouvait analyser, mais également par le temps qu'il mettait à effectuer le traitement. De plus, dans le code source original, les algorithmes de calcul des caractéristiques étaient tous imbriqués dans une grande boucle ce qui limitait grandement l'évolutivité de l'application, et notamment l'ajout de nouvelles caractéristiques. Au-delà de l'aspect purement fonctionnel de l'application, une partie importante du travail réalisé nécessitait de trouver une solution viable pour effectuer un grand nombre d'entrées/sorties parallèles sur du matériel classique. Les objectifs à atteindre étaient donc de repousser les limites du programme original et d'accélérer les temps de traitement, de distribuer les calculs en gérant au mieux les grandes quantités de lectures de données en parallèle, mais également d'améliorer son évolutivité.

2. Bilan

Ce travail a été réalisé en trois grandes étapes :

1. la réalisation de l'ensemble des objectifs pour l'exécution du programme sur une seule machine ;
2. la distribution des calculs sur cluster pour réduire davantage les temps de traitement ;
3. la réalisation d'un outil permettant le déploiement du programme sur un cluster de manière relativement simple.

a. Amélioration de la version non distribuée sur un nœud

La limite de taille du fichier qu'il est possible de traiter a été grandement repoussée en revoyant la manière dont le fichier était lu. Plutôt que de mettre l'intégralité du fichier en mémoire, il a été choisi de ne lire et traiter qu'une fraction du fichier à la fois, et de répéter cette opération jusqu'à ce que le fichier ait été entièrement lu. Malheureusement, il n'est

pas possible de supprimer complètement la limite, car les résultats des calculs des segments doivent être conservés pour être normalisés avant la phase de classification.

De manière théorique, j'évalue que la limite de taille de fichier traitable a été repoussée de près de 69000%. Ainsi, si la taille maximale qu'une machine est capable d'analyser est de 1Go, elle serait capable de traiter des fichiers de près de 690Go.

Les temps de traitement ont également été diminués dans la version non distribuée, en améliorant la parallélisation OpenMP pour réaliser l'étape de classification sur plusieurs cœurs. L'utilisation d'une bibliothèque reconnue de calcul de transformées de Fourrier rapide, FFTW, a également permis d'améliorer les performances de l'application, mais il est toutefois possible de se passer de l'utilisation de cette librairie pour utiliser le code de calcul original des FFT.

L'amélioration de l'évolutivité a été réalisée en convertissant une partie du code en objet. Cela a permis la réalisation d'une classe permettant d'allouer dynamiquement des tableaux à une ou deux dimension, peu importe son type, en n'ayant pas à se soucier de la libération de la mémoire. Les quatre caractéristiques actuelles héritent d'une classe mère disposant des outils communs, tels que le calcul de la variance ou de la moyenne. Le code propre à chaque caractéristique est ainsi réduit au strict minimum.

b. Conception d'une solution distribuée sur plusieurs nœuds

Pour la distribution de l'application, nous avons utilisé MPI qui est un standard dans le domaine du calcul distribué. La grande taille des fichier à traiter, ainsi que le nombre de machines devant réaliser les calculs nous ont obligé à trouver une solution intelligente pour la gestion des entrées/sorties. La solution adoptée consiste à spécialiser un certain nombre de processus pour leur faire effectuer la lecture et l'envoi des données à traiter à d'autres processus afin que ces derniers ne réalisent que les calculs. Chaque processus *reader* dispose ainsi d'un certain nombre de processus *worker*, et forment ensemble un groupe de lecture. La taille, et la composition des groupes est entièrement configurable ce qui permet de ne pas être bloqué dans une configuration qui ne serait peut-être pas performante sur certains clusters.

c. Déploiement et exécution du programme sur un cluster

Pour exécuter l'application distribuée sur un cluster, il est nécessaire d'éditer un fichier texte contenant une ligne par processus à exécuter sur une machine. L'ordre des machines dans ce fichier détermine si le processus sera un *reader* ou un *worker*, ainsi que le rang de son groupe, et son rang au sein du groupe. La création de ce fichier est fastidieuse à faire manuellement, mais est nécessaire lors d'une nouvelle allocation de nœuds par OAR ou lors d'un changement de configuration d'exécution. Pour cette raison, un outil de déploiement réalisé en Perl a été développé. Celui-ci permet de choisir un mode d'exécution simple

utilisant des heuristiques déterminées expérimentalement pour un certain nombre de nœuds sur un cluster donné. Il permet également de choisir exactement la composition des groupes, en intervenant sur des paramètres simples tels que le nombre de processus par nœuds ou le nombre de groupes de lecture à utiliser. Cet outil a d'ailleurs été utilisé pour la réalisation des fichiers d'heuristiques et des graphiques du Chapitre 5.

3. Problèmes rencontrés et limitations éventuelles

Un des premiers problèmes rencontrés fut d'offrir la possibilité d'utiliser FFTW ou le code original pour le calcul des transformées de Fourrier rapides. Cela nécessitait la duplication d'une grande partie du code, et l'utilisation de variables de types différents suivant la méthode choisie. La méthode retenue a été de créer une classe et des macros permettant d'uniformiser l'accès aux variables et de ne compiler que le code nécessaire en utilisant des macros #ifdef.

Sans être un problème en soit, un des majeurs challenges de ce projet a été la distribution des traitements sur cluster. N'étant pas du métier, imaginer les communications minimales nécessaires au bon fonctionnement du programme, tout en tentant de les recouvrir au maximum avec les phases de calculs était un réel défi.

Avant d'utiliser le système de cartographie contenant les bornes des trames et segments, je tentais de calculer ces informations au moment où j'en avais besoin dans le code. Cela s'est révélé être une mauvaise idée car le code ne fonctionnait que pour certains fichiers, sans doute à cause d'arrondis. Avec la cartographie, chaque processus calcule à l'initialisation les bornes de chaque segment ainsi que la liste de la première et dernière trame que chaque *worker* devra calculer à chaque cycle. Cette étape est très rapide et exécutée une seule fois par processus. Au final cette méthode est bien plus simple, et permet d'être sûr des résultats. La boucle permettant de faire ces calculs est basée sur la boucle de lecture en parcourant « virtuellement » le fichier grâce à une variable évoluant entre 0 et la taille du son à traiter (en nombre d'échantillons).

Les limitations éventuelles de l'application sont principalement liées au mode de communication choisi. Les échanges entre processus ont l'avantage d'être le plus simple possible, mais l'inconvénient majeur de cette solution est que si un nœud, quel qu'il soit, ne fonctionne pas correctement, l'ensemble des processus sont pénalisés car chacun attend des résultats d'un autre pour continuer. Le temps de traitement dépend du processus et donc du nœud le plus lent.

Les entrées/sorties sont une composante critique des applications parallèles aujourd'hui. Les expériences scientifiques génèrent de plus en plus de données. A titre d'exemple, le « Large Hadron Collider » du Cern génère environ 15 pétaoctets par an[6]. Cela équivaut à une écriture

[6] L'informatique au LHC : http://public.web.cern.ch/public/fr/lhc/Computing-fr.html

constante à plus de 475 Mo/s. S'il est possible de trouver des solutions fiables et bon marché pour accélérer de grandes quantités de calculs, cela n'est malheureusement pas encore le cas pour le stockage et la récupération de grandes quantités de données.

Dans notre cas, si une des machines rencontre un problème pendant l'exécution d'un traitement, il n'y a pas de moyen de reprise. Il est alors nécessaire de recommencer le traitement depuis le début. Comme la probabilité qu'une machine du cluster subisse une avarie au cours d'un long traitement augmente avec la taille du cluster, il est de plus en plus nécessaire de sauver régulièrement l'état du programme en cours d'exécution (« *checkpointing* »).

Cela explique l'importance des entrées/sorties distribuées dans les systèmes de calcul haute performance, autant pour lire des données à traiter ou enregistrer des résultats que pour faire du *checkpointing*.

Afin d'optimiser la lecture des données à partir d'un serveur de stockage, nous nous sommes penchés sur MPI-IO. Une analyse préliminaire révèle que les fonctions permettant d'effectuer la lecture en parallèle avec MPI-IO ne permettent pas de faire ce dont nous avons besoin. En effet, il ne semble pas possible d'aboutir simplement à ce que plusieurs nœuds lisent des données majoritairement différentes avec un léger chevauchement. De plus, utiliser MPI-IO impliquerait d'adapter la bibliothèque que nous utilisons (libsndfile) pour réaliser la lecture à partir de fichiers son (voir page 9), ou d'en développer une nouvelle. De plus, l'ensemble des travaux traitant d'MPI-IO trouvés sur le net ne traitent que l'utilisation de ces fonctions avec un système de fichier parallèle. IBM semble confirmer cela dans la documentation de sa bibliothèque « Parallel Environment » [20]. L'option d'utiliser MPI-IO a donc été écartée.

4. Perspectives

La lecture en réseau est actuellement réalisée sur un serveur NFS connecté à un NAS de Supélec. Une des perspectives intéressantes serait de passer à une architecture de stockage basée sur un système de fichier parallèle. Un tel système est toutefois complexe à mettre en œuvre et également relativement couteux du fait de l'investissement nécessaire en nouveau matériel.

Pour améliorer la reprise sur erreur, il serait intéressant de mettre en place une politique de *checkpointing* efficace sur simples clusters de PC. Réaliser des entrées/sorties parallèles efficaces avec des ressources bon marché est justement une des préoccupations du groupe IDMaD de Supélec.

Un autre axe d'amélioration serait de réaliser un certain nombre d'opérations sur GPU, notamment le calcul de la FFT grâce à des bibliothèques existantes comme cuFFT. Il serait

également possible de recouvrir les calculs sur GPU et ceux réalisés sur CPU pour améliorer davantage les performances.

Pour parer aux éventuelles limitations citées précédemment, une démarche algorithmique intéressante serait de revoir la méthode de distribution et de fonctionner sur un mode ou chaque processus demande des données à traiter dès qu'il a fini afin qu'un nœud lent pénalise le moins possible le temps de traitement.

5. Remarques personnelles

Durant les mois passés en stage à Supélec, j'ai eu l'occasion de voir l'ensemble des facettes de ce projet. Au-delà du coté informatique, ce travail de mémoire m'a permis de découvrir certains aspects du traitement de signal audio. Sans avoir la prétention de comprendre à 100% le fonctionnement ou l'intérêt de certaines caractéristiques dans la classification de sources sonores, je conçois mieux le fonctionnement général d'un tel programme.

Le travail tel qu'il a été réalisé peut être scindé en deux sous projets :

- L'amélioration de l'existant sur une seule machine ;
- La distribution des calculs sur cluster de PCs multi-cœur.

Chacun de ces sous projets s'est déroulé suivant un plan classique en entreprise :

- Acquisition de connaissances sur l'existant ;
- Spécification des besoins ;
- Etude et recherche d'informations ;
- Conception ;
- Tests ;
- Mesures de performances ;
- Validation.

Ce projet m'a également permis de découvrir des choses intéressantes dont je n'ai pas l'occasion de me servir dans le cadre de mon métier d'administrateur réseau, tels que l'utilisation de clusters de calculs, de MPI ou d'OpenMP. Cela m'a également permis de découvrir un autre métier me permettant ainsi de pouvoir mieux faire la part des choses lors de la recherche et du choix de mon futur emploi.

Bibliographie

[1] K. M. Fristrup et W. A. Watkins, «Marine Animal Sound Classification,» WHOI Technical Report WHOI-94-13, Woods Hole Oceanographic Institution, 1993.

[2] W. Chou et L. Gu, Robust Singing Detection in Speech/Music Discriminator Design, Salt Lake City, UT: IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 2001.

[3] H. Harb et L. Chen, Robust Speech Music Discrimination Using Spectrum's First Order Statistics And Neural Networks, Paris: Seventh International Symposium on Signal Processing and Its Applications (ISSPA), 2003.

[4] E. Scheirer et M. Slaney, Construction And Evaluation Of A Robust Multifeature Speech/Music Discriminator, Munich: IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 1997.

[5] K. El-Maleh, M. Klein, G. Petrucci et P. Kabal, Speech/Music Discrimination For Multimedia Applications, Istanbul: IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 2000.

[6] J. Ajmera, I. A. McCowan et H. Bourlard, Robust HMM-Based Speech/Music Segmentation, Orlando: IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 2002.

[7] T. Beierholm et P. Baggenstoss, Speech music discrimination using class-specific features, Proceedings of the 17th International Conference on Pattern Recognition (ICPR), 2004.

[8] M. P. Norton et D. G. Karczub, Fundamentals of Noise and Vibration Analysis for Engineers, Cambridge University Press, 2003.

[9] G. Peeters, A Large Set of Audio Features for Sound Description, IRCAM Analysis/Synthesis Team, IRCAM Research Report, 2004.

[10] MBWF / RF64 : An extended File Format for Audio, European Broadcasting Union Technical Specification: EBU-TECH 3306, 2009.

[11] B. Chapman, G. Jost et R. Van Der Pas, Using OpenMP, MIT Press, 2008.

[12] G. Williams et D. P. W. Ellis, Speech/Music Discrimination Based On Posterior Probability Features, Budapest: Sixth European Conference on Speech Communication and Technology (EUROSPEECH), 1999.

[13] M. Dorier, G. Antoniu, F. Cappello, M. Snir et L. Orf, «Damaris: Leveraging Multicore Parallelism to Mask I/O Jitter,» Joint INRIA/UIUC Laboratory for Petascale Computing, INRIA Research Report 7706, 2011.

[14] Y. Gu et R. Grossman, Towards Efficient and Simplified Distributed Data Intensive Computing, IEEE Transactions on Parallel and Distributed Systems, 2011.

[15] D. Chau, J. Lin, M. Matczynski et N. Palmer, MORRIS : A Distributed File Système for Read-Intensive Applications, MIT Unpublished Report, 2005.

[16] R. Hedges, K. Fitzgerald, M. Gary et D. M. Stearman, Comparison of Leading Parallel NAS File Systems on Commodity Hardware, Lawrence Livermore National Laboratory Technical Report: LLNL-TR-461793, 2010.

[17] GCC, the GNU Compiler Collection, Online Technical Documentation : http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, Accessed in 2012.

[18] W. Gropp, E. Lusk et A. Skellum, Using MPI, MIT Press, 1999.

[19] D. Padua, Encyclopedia of Parallel Computing, Volume 4, Springer Reference, Springer , 2011.

[20] Determining which nodes will participate, in "Parallel file I/O in Parallel Environment Runtime Edition (Version 1 Release 2) : Operation and Use", IBM, 2012.

Table des illustrations

Résumé :

La classification de sources sonores est un domaine du traitement de signal, dont l'utilité ne cesse de croître dans l'ère numérique actuelle. L'intérêt d'un tel procédé est important, allant de l'indexation de contenus à l'identification de mammifères marins. La possibilité d'appliquer, de manière complètement automatique, un traitement sur un fichier audio en fonction du type de sons contenus dans celui-ci est également une utilisation possible.

Hormis la qualité des résultats obtenus, les autres facteurs primordiaux affectant l'utilité d'une telle technologie sont la vitesse de traitement, la capacité de traiter de grands volumes de données et la facilité d'ajouter ou de modifier des algorithmes de classification.

Ce mémoire d'ingénieur s'attache à retracer les étapes qui ont été nécessaires afin d'adresser ces besoins. Un outil existant à Supélec a ainsi été fondamentalement revu afin de convertir le cœur du programme en code orienté objet pour rendre plus souple l'ajout de nouveaux algorithmes. Une version distribuée sur cluster de PC multicoeurs a ensuite été a ensuite été conçue et développée afin de réaliser « un passage à l'échelle », c'est-à-dire de repousser fortement les limites liées à la taille du problème à traiter.

Mots clés : classification de sources sonores, architecture objet, calcul distribué, entrées/sorties parallèles, cluster de calcul, MPI.

Abstract:

Sound classification is a field of signal processing. In this digital age it is constantly becoming more useful. The value of such a process is substantial, ranging from content indexing to the identification of marine mammals. One of the other great possible uses for this technology is the ability to apply, given the type of sound in a file, different audio processing algorithms in a fully automated fashion.

Important factors, aside from the quality of the results, are processing speed, the ability to handle large data sets and the ease of adding and changing classification algorithms.

This engineering thesis endeavors to retrace the steps that were needed to address these needs. An existing tool at Supélec has been fundamentally revised in order to convert the heart of the program from procedural to object oriented code for more flexibility in adding new algorithms. A distributed version was then designed and developed to achieve a good scalability on a multicore PC cluster, *i.e.* to push beyond the size limit of problems that can be processed

Keywords: sound classification, object-oriented architecture, distributed computing, parallel I/O, computational cluster, MPI.