1 Objectif

Seconde version de la programmation multithread de l'analyse discriminante prédictive dans Sipina 3.11. Le nombre de threads est paramétrable. Les charges sont équilibrées.

Dans le document « Multithreading pour l'analyse discriminante », nous présentions une implémentation multithread de l'analyse discriminante à destination des machines à processeurs multi-cœurs ou multiprocesseurs[1]. A occupation mémoire égale par rapport à la version séquentielle, elle permettait de réduire les temps de calculs dans des proportions considérables en fonction des caractéristiques des données. La solution présentait néanmoins deux faiblesses : le nombre de cœurs utilisé était tributaire du nombre de classes K dans la base à traiter ; l'équilibrage des charges entre les cœurs dépendait de leur fréquence. Ainsi, sur une des bases d'expérimentation avec K = 2 classes très fortement déséquilibrées, le gain était quasi-nul par rapport à la version monothread.

Dans ce tutoriel nous présentons une nouvelle solution implémentée dans **Sipina 3.11**. Au prix d'une occupation mémoire accrue, que nous préciserons, elle permet de surmonter les deux goulots d'étranglement pointés sur la version précédente. Les capacités de la machine sont pleinement utilisées. Plus intéressant même, le nombre de threads devient paramétrable, permettant à l'utilisateur d'adapter les ressources machines à exploiter pour le traitement des données.

Pour mieux situer les améliorations induites par notre stratégie, nous comparons nos temps d'exécution avec la version multithread parcimonieuse en mémoire développée précédemment, la version séquentielle, et la référence SAS 9.3 (proc discrim).

2 Analyse discriminante linéaire prédictive

Il existe d'excellents supports décrivant l'analyse discriminante sur le web[2]. L'objectif est de prédire les valeurs d'une variable cible Y prenant ses valeurs dans {1, 2, …, K} à partir de *p* variables prédictives X = (X_1, X_2, …, X_p). Nous disposons d'un échantillon de taille *n.* Un individu est noté ω, sa valeur pour la variable cible s'écrit $y(\omega)$. Le nombre d'observations appartenant à la classe Y = k est égal à n_k. Nous avons établi que la durée d'exécution dépendait avant tout du calcul des **K** matrices de variances covariances conditionnelles de dimension (p x p).

$$S_k = \left(\frac{1}{n_k} \sum_{\omega : y(\omega)=k} \left[x_i(\omega) - \bar{x}_{i,k} \right] \times \left[x_j(\omega) - \bar{x}_{j,k} \right] \right)_{i,j=1,\cdots,p}$$

Où $\bar{x}_{i,k}$ représente la moyenne de la variable X_i pour les individus de la classe (Y = k).

Programmation monothread. Dans une programmation séquentielle monothread, le plus simple, et vraisemblablement le plus rapide, est d'effectuer une seule passe sur les données. Cela impose de maintenir en mémoire plusieurs éléments. En double précision (8 octets par valeur), l'espace occupé est précisément égale à :

- « 8 x (p x K) » pour les vecteurs des moyennes conditionnelles ;

[1] http://tutoriels-data-mining.blogspot.fr/2013/05/multithreading-pour-lanalyse.html

[2] Ex sur Wikipédia, http://fr.wikipedia.org/wiki/Analyse_discriminante_linéaire

- Plus « 8 x [(p x p) x K] »[3] pour les matrices S_k.

Soit au total :

$$8 \times [(p \times K) + (p \times p) \times K]$$

$$= 8 \times K \times p \times (p + 1)$$

Par exemple, sur la base MIT FACE IMAGES avec K = 2 classes et p = 361 descripteurs, elle serait de :

$$8 \times 2 \times 361 \times (361 + 1) \approx 2 \text{ Mo}$$

Programmation multithread (Parcimonieuse en mémoire). Dans Sipina 3.10, nous créons des index qui permettent d'associer chaque observation à son groupe d'appartenance. Par la suite, nous lançons K threads pour la construction des matrices S_k associées à chaque groupe. Ainsi, mis à part les index qui peuvent être par ailleurs stockés sur disque, l'organisation et l'occupation mémoire des structures de calcul sont exactement les mêmes par rapport à la version monothread.

Programmation multithread (Charges équilibrées). L'affaire est plus compliquée cette fois-ci. **Il s'agit de subdiviser les calculs en M blocs, indépendamment de la valeur de K**. Pour ce faire, penchons nous sur une des caractéristiques de la covariance. Les termes de la matrice S_k peuvent s'écrire d'une autre manière :

$$S_k = \left(\frac{1}{n_k} \sum_{\omega : y(\omega)=k} x_i(\omega) x_j(\omega) - \bar{x}_{i,k} \bar{x}_{j,k} \right)_{i,j=1,\ldots,p}$$

$$= \left(\frac{1}{n_k} \sum_{\omega : y(\omega)=k} x_i(\omega) x_j(\omega) - \frac{1}{n_k^2} \sum_{\omega : y(\omega)=k} x_i(\omega) \times \sum_{\omega : y(\omega)=k} x_j(\omega) \right)_{i,j=1,\ldots,p}$$

Tous les termes sont additifs ! De fait, il est possible de constituer arbitrairement M groupes d'individus indépendamment de leur appartenance aux classes, d'effectuer les sommations en parallèle, de réaliser les consolidations pour chaque groupe k avant de procéder au calcul des moyennes et des covariances conditionnelles. Les structures de calcul ci-dessus sont dupliquées M fois, et l'occupation mémoire devient :

$$\mathbf{M \times} [8 \times K \times p \times (p + 1)]$$

Ainsi, avec M = 4 threads pour la base MIT FACE IMAGES, l'occupation mémoire des structures de calcul revient à :

$$4 \times [8 \times 2 \times 361 \times (361 + 1)] \approx 8 \text{ Mo}$$

Ca reste tout à fait raisonnable.

L'intérêt de tout ceci est que : (1) nous pouvons fixer la valeur de M en fonction des ressources que nous souhaitons allouer aux calculs ; et (2) en envoyant le même nombre d'observations ($\frac{n}{M}$) dans chaque thread, les charges sont parfaitement équilibrées.

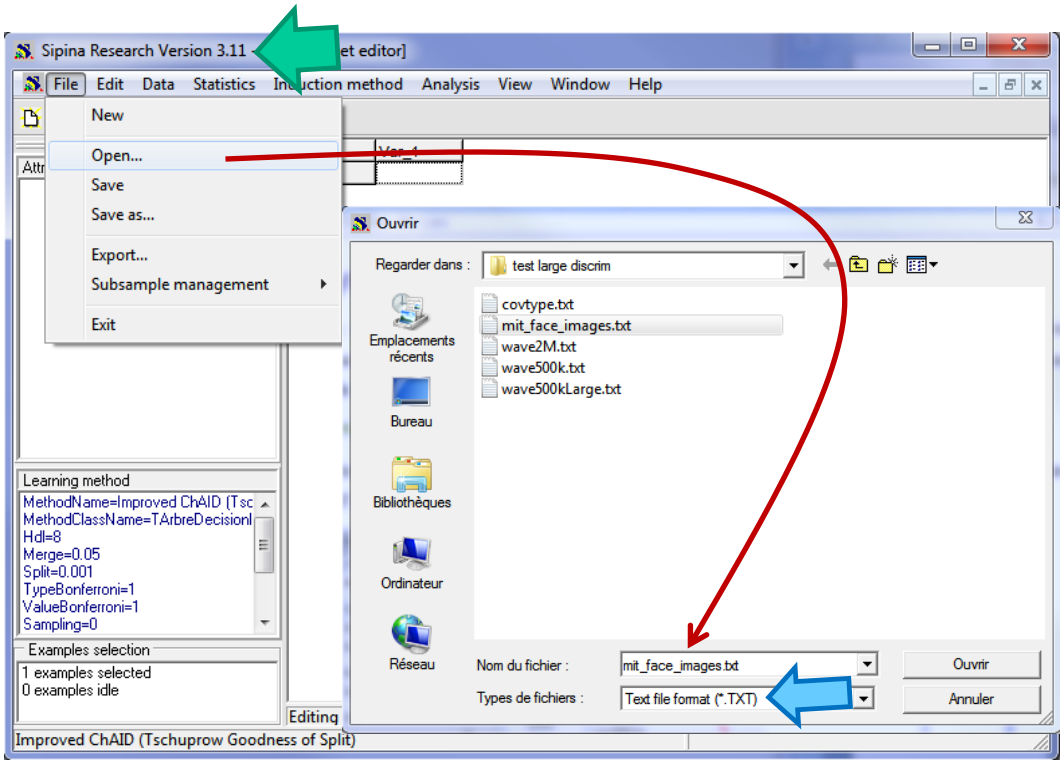
[3] Si on tient compte du fait que la matrice est symétrique, nous pouvons la réduire la taille de S_k à $[\frac{p \times (p+1)}{2} \times 8]$.

3 Analyse discriminante avec SIPINA

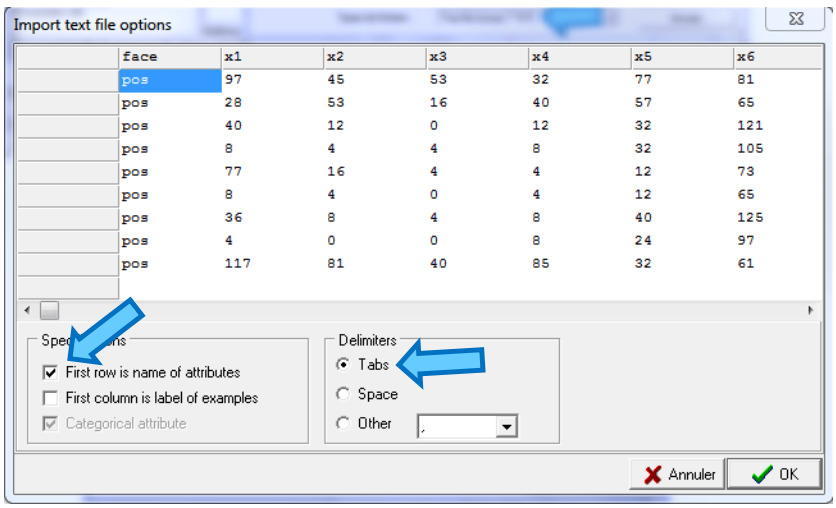
La mise en œuvre de l'analyse discriminante monothread et multithread version « mémoire parcimonieuse » est décrite dans notre précédent tutoriel[4]. Dans cette section, nous détaillons l'utilisation de la nouvelle approche implémentée dans **Sipina 3.11**.

3.1 Importation des données

Nous souhaitons traiter la base MIT FACE IMAGE. Dans un premier temps, nous importons le fichier au format TXT (fichier texte avec séparateur tabulation).

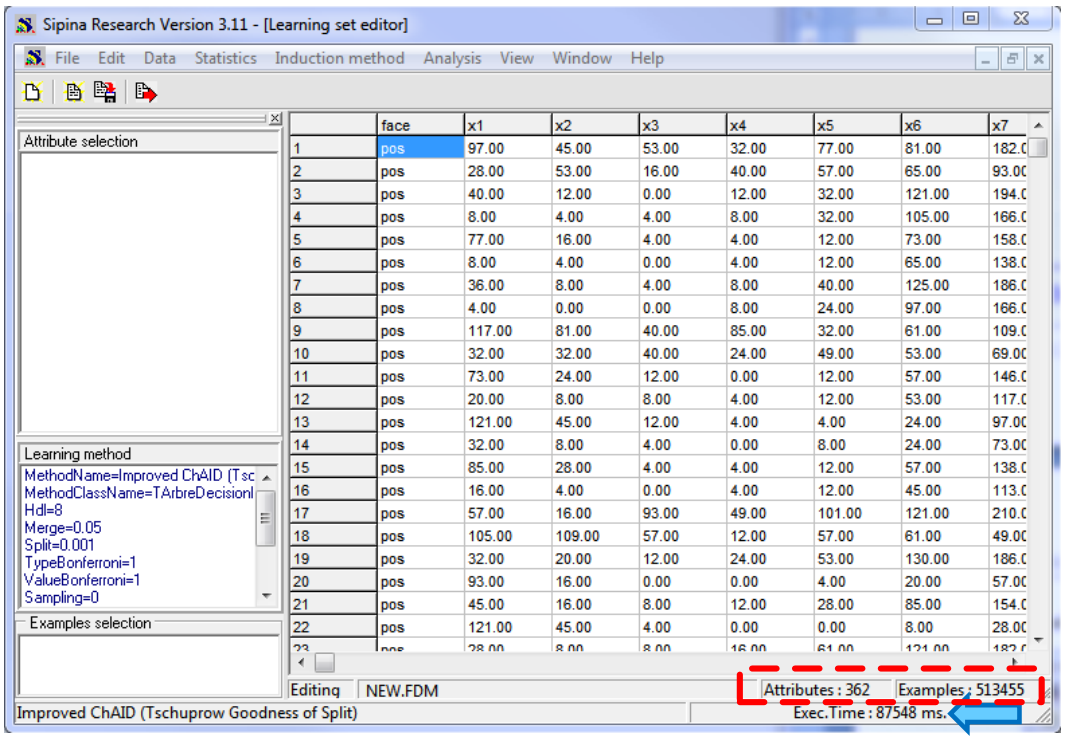


Un assistant apparaît. Il vous permet de définir les spécificités du fichier (type de séparateur, la première ligne correspond aux noms des variables). Nous validons en cliquant sur OK.



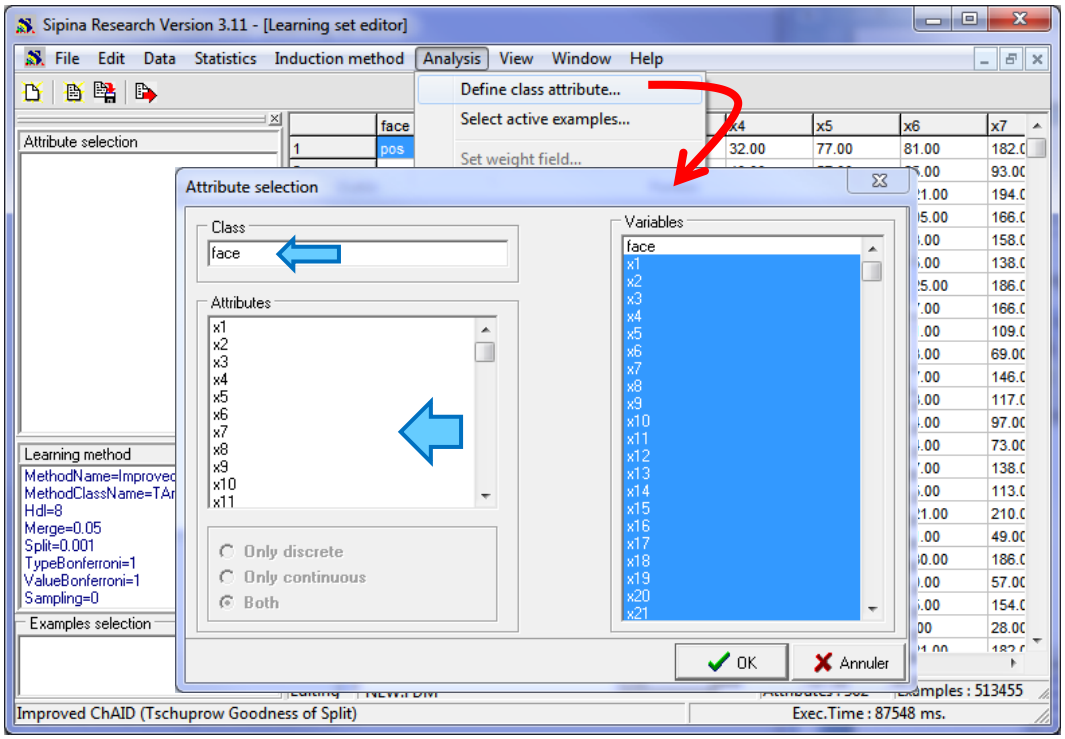
[4] http://tutoriels-data-mining.blogspot.fr/2013/05/multithreading-pour-lanalyse.html

Le chargement des 513.455 individus et 362 variables dans la grille de données a duré 87 secondes.



3.2 Choix des variables

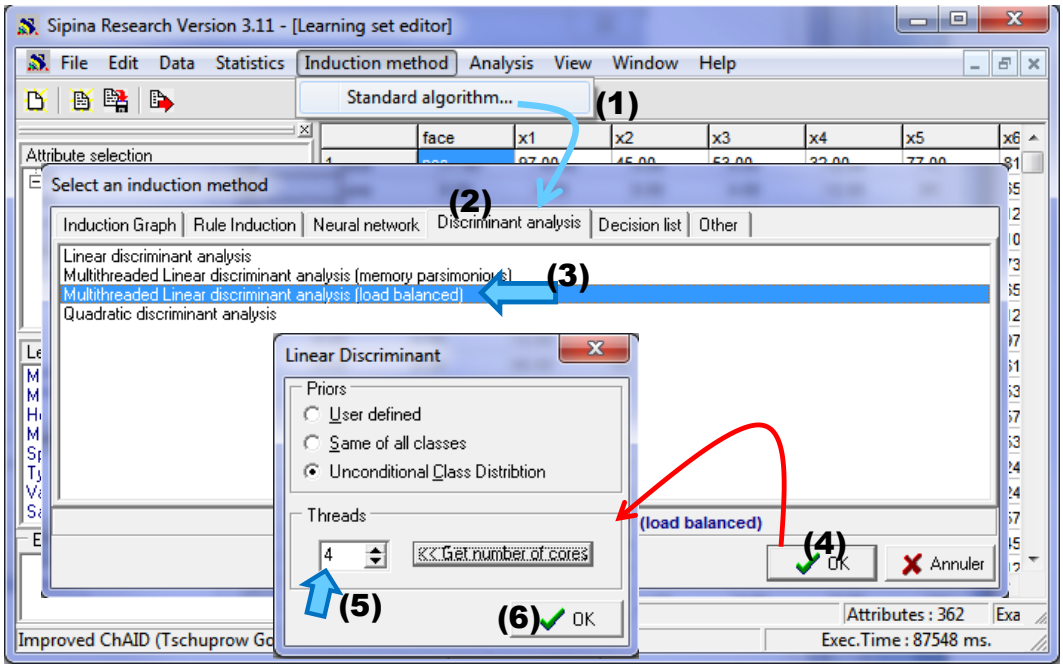
Pour définir le rôle des variables, nous actionnons le menu ANALYSIS / DEFINE CLASS ATTRIBUTE. Par « glisser-déposer », nous plaçons FACE en CLASS, toutes les autres variables en ATTRIBUTES.



Nous validons en cliquant sur OK. Les variables sélectionnées apparaissent dans la partie gauche de la fenêtre principale. Le symbole « D » désigne une variable discrète (catégorielle), « C » une variable continue (quantitative).

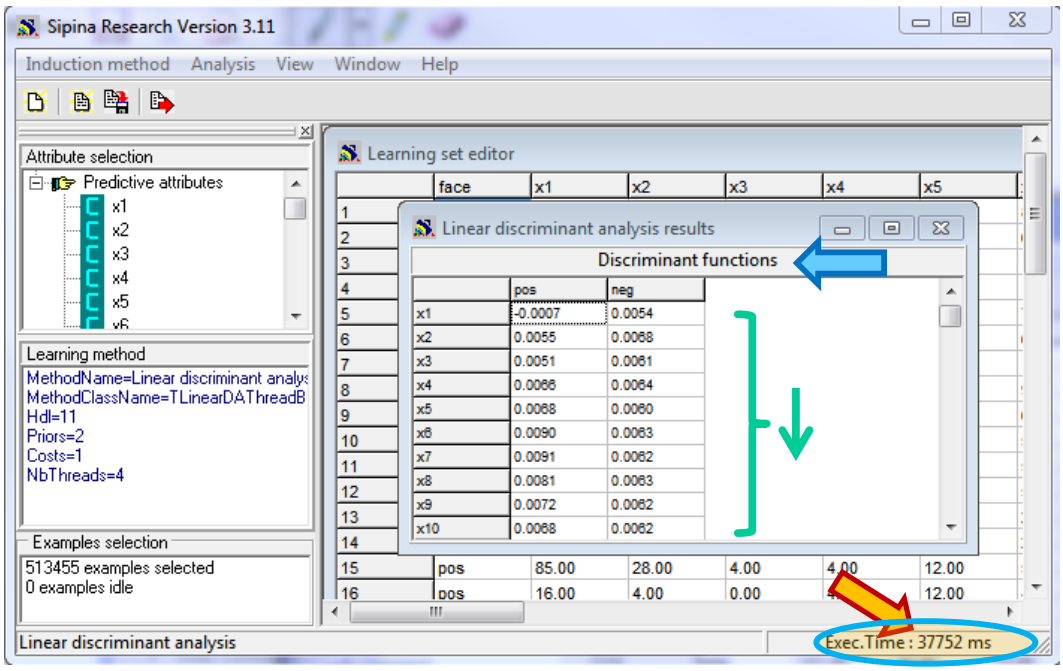
3.3 Algorithme multithread équilibré

Nous souhaitons modéliser à l'aide de l'analyse discriminante linéaire. Pour sélectionner la méthode, nous actionnons le menu INDUCTION METHOD / STANDARD ALGORITHM **(1)**.



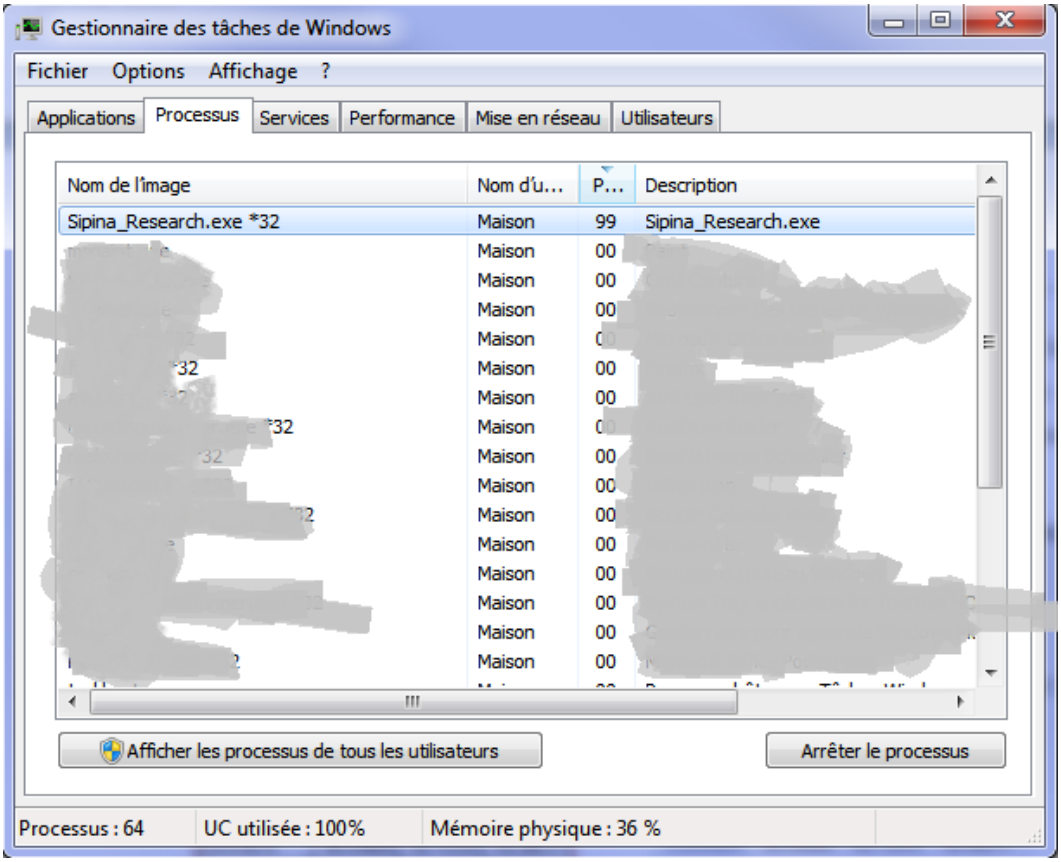
Dans l'onglet DISCRIMINANT ANALYSIS **(2)**, nous avons choisi l'algorithme multithread équilibré MULITHREAD LINEAR DISCIMINANT ANALYSIS (LOAD BALANCED) **(3)**. Nous validons **(4)**. Dans la boîte de paramétrage, nous spécifions le nombre de threads à utiliser **(5)**. Nous le fixons à **4 threads** sachant qu'il est possible de connaître le nombre de cœurs disponibles sur la machine en actionnant le bouton « Get number of cores ». Nous confirmons ces paramètres **(6)**.

Il ne nous reste plus qu'à cliquer sur le menu ANALYSIS / LEARNING pour lancer les calculs. Au bout de **37,752 secondes**, les fonctions discriminantes sont affichées dans une nouvelle fenêtre.



Remarque : Il s'agit de durée réelle ici, calculée en effectuant la différence entre deux appels à la fonction GetTickCount() de Windows. Ainsi, puisque tous les cœurs sont sollicités par SIPINA, le temps mesuré peut être fortement affecté si une autre application est en cours d'exécution (ex. un navigateur internet avec FlashPlayer, etc.).

Nous avons surveillé le gestionnaire de tâche de Windows. Effectivement, toutes les ressources processeur (99%) sont bien affectées au calcul sous Sipina.



3.4 Expérimentations sur plusieurs bases

Pour évaluer le comportement de la solution multithread face à l'implémentation séquentielle, nous avons mesuré les temps de traitements sur différentes bases[5]. Pour la série de WAVE, le nombre de classes est toujours K = 3, et surtout elles sont équilibrées (les effectifs par classes sont quasi-identiques) : WAVE500K que nous avons décrit ci-dessus ; WAVE500KLARGE, toujours avec 500.000 observations mais avec p = 121 descripteurs (100 variables supplémentaires par rapport à la base originelle) ; WAVE2M avec n = 2.000.000 et p = 21. Ce sont des bases artificielles que nous pouvons faire évoluer à notre guise. L'idée est d'étudier l'impact de l'évolution du nombre d'observations et du nombre de descripteurs par rapport à la référence WAVE500K ci-dessus.

Nous avons ensuite traité les bases réelles COVTYPE avec K = 7 classes, mais assez déséquilibrées (2 modalités de la variable cible concentrent une grande partie des observations) ; et MIT FACE IMAGE, binaire (K = 2), avec des effectifs très déséquilibrés.

[5] Les mêmes que dans le tutoriel de présentation de la première version multithread de l'analyse discriminante linéaire - http://tutoriels-data-mining.blogspot.fr/2013/05/multithreading-pour-lanalyse.html

Nous recensons dans le tableau ci-dessous les temps de traitement. Les durées sont en secondes. Nous mettons en référence les performances de la version monothread. « Ratio » indique la réduction du temps d'exécution lors du passage au multithread, il est défini comme suit :

$$Ratio = \frac{Durée\ exec.monothread}{Durée\ exec.multithread}$$

Par exemple, Ratio = 3 = (1.30 / 0.44) pour la base WAVE500K indique que la version multithread « mémoire parcimonieuse » est 3 fois plus rapide que la version monothread (ou encore, le temps de traitement a été divisé par 3).

Notre machine d'expérimentation est équipée d'un processeur Intel Q9400 **Quad-core**. Pour la version « load balanced », nous avons systématiquement sollicité les 4 cœurs disponibles.

Dataset	K	n	p	SIPINA (multithread) Load balanced	SIPINA (multithread) Memory Parsimonious	SIPINA (single thread)
Wave 500k	3	500000	21	0.38	0.44	1.30
Wave 500k Large	3	500000	121	4.87	6.13	19.19
Wave 2M	3	2000000	21	1.39	1.83	5.16
Covtype	7	581012	52	1.44	2.67	5.30
Face Images	2	513455	361	37.75	135.46	142.73

	Ratio against single thread	
Wave 500k	3.5	3.0
Wave 500k Large	3.9	3.1
Wave 2M	3.7	2.8
Covtype	3.7	2.0
Face Images	3.8	1.1

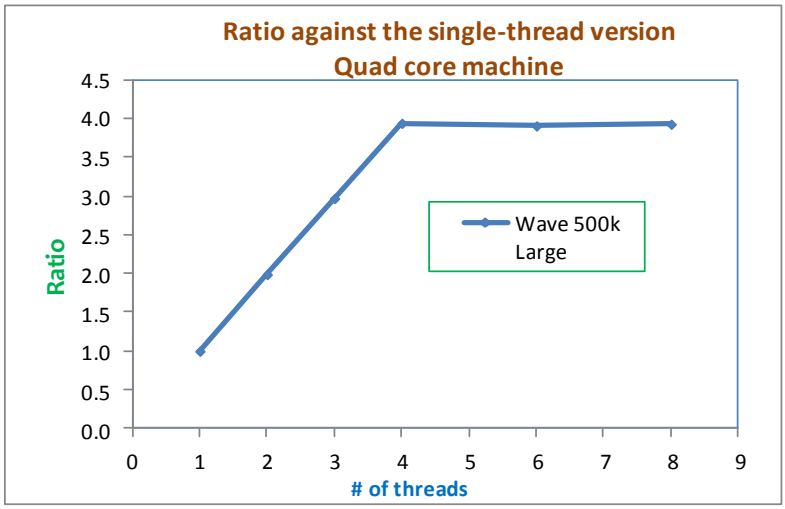
Ces résultats nous inspirent plusieurs pistes de réflexion :

- Les performances de la version multithread parcimonieuse en mémoire, et surtout leurs relations avec les caractéristiques des données, ont largement été détaillées dans notre précédent document. Nous n'y reviendrons pas, sauf à noter que dans des configurations particulièrement défavorables, le gain est quasi-nul (ex. ratio = 1.1 pour MIT FACE IMAGES).
- Il en est tout autrement en ce qui concerne la version « charges équilibrées » (Load Balanced). Les temps de calcul sont déconnectés des spécificités des données. Le ratio est à peu près le même quelle que soit la base. La réduction pour MIT FACE IMAGES est particulièrement impressionnante.
- Mais le ratio n'est pas égal à 4 parce que (1) nous utilisons le temps réel pour mesurer les durées, le système a en permanence d'autres tâches à exécuter par ailleurs ; (2) une partie des traitements reste monothread.

Conclusion. Par rapport à la précédente (Memory Parsimonious), nous avions obtenu exactement ce que nous souhaitions dans cette seconde version : le nombre de cœur à utiliser est paramétrable, les charges sont bien équilibrées. Bien sûr, rien n'est gratuit en ce bas monde, cela fut au prix d'une augmentation de l'occupation mémoire des structures de calcul. Mais après expertise, je me suis rendu compte qu'elle restait tout à fait raisonnable sur la majorité des bases de données.

3.5 Influence du nombre de cœurs utilisés

Dans quelle mesure rajouter des threads permet-il d'améliorer les performances ? Dans le graphique ci-dessous, nous montrons les ratios en fonction du nombre de thread utilisés pour la base « WAVE500K Large ».



Sur une machine Quad-core (à 4 cœurs), rajouter un thread permet d'améliorer les temps de traitement tant que M < 4. Notons que l'évolution du ratio est linéaire en fonction du nombre de threads c.-à-d. le calcul global est bien subdivisé en autant de cœurs utilisé. Le ratio n'est pas parfait cependant. Il sera très légèrement inférieur à M parce que, entres autres raisons, une partie des calculs reste monothread (calcul de la matrice de variance covariance intra-classes et son inversion).

Au-delà de M = 4, un thread additionnel n'amène rien. Mais il ne dégrade pas non plus les performances, lesquelles sont véritablement plafonnées par le nombre de cœurs disponibles.

3.6 Comparaison avec SAS

Nous avons testé les performances de SAS 9.3 (proc discrim) sur les mêmes bases. Ex.

```
proc discrim data = mesdata.wave500k;
   class onde;
   priors proportional;
run;
```

Nous comparons les durées d'exécution avec la nouvelle version multithread de Sipina :

Dataset	K	n	p	SIPINA (threads) Load balanced	SAS
Wave 500k	3	500000	21	**0.38**	1.65
Wave 500k Large	3	500000	121	**4.87**	9.09
Wave 2M	3	2000000	21	**1.39**	6.19
Covtype	7	581012	52	**1.44**	4.29
Face Images	2	513455	361	**37.75**	39.12

Nous constatons que :

- Sipina est systématiquement supérieur. Utiliser toute la puissance disponible d'une machine en passant par les threads ne peut être que bénéfique.
- Ce résultat est d'autant plus remarquable que SIPINA, de par ses structures internes de stockage des données, est désavantagé lorsque la base comporte un grand nombre de variables (ex. WAVE500KLARGE, MIT FACE IMAGES…). Le passage au multithread permet de dépasser cet inconvénient.

Remarque : Je sais qu'il est possible de passer au multithreading sous SAS en utilisant l'option THREADS. Mais elle n'est utilisable que pour certaines méthodes à l'heure actuelle (SAS 9.2). J'imagine que la modification de la « proc discrim » surviendra à un moment ou un autre.

4 Conclusion

Très enthousiasmé par une première version multithread de l'analyse discriminante implémentée dans SIPINA 3.10, j'ai essayé d'améliorer la procédure en déconnectant ses performances des caractéristiques des données à traiter. Introduite dans **SIPINA 3.11**, cette version utilise mieux les ressources machines en exploitant (éventuellement) tous les cœurs disponibles, et en répartissant mieux les charges.