1. Objectif

Montrer les atouts du multithreading lors de l'induction des arbres de décision.

Une grande partie des PC modernes sont équipés de processeurs multi-cœurs. Dans les faits, l'ordinateur fonctionne comme s'il disposait de plusieurs processeurs. Certains d'ailleurs, les gros serveurs notamment, en disposent effectivement. Les logiciels et les algorithmes de data mining doivent être aménagés pour pouvoir en tirer profit. A l'heure actuelle, rares sont les outils à large diffusion qui exploitent ces nouvelles caractéristiques des machines. Comme me l'ont fait remarquer plusieurs utilisateurs, il est un peu dommage de disposer d'une superbe machine « quad-core », et de voir le logiciel s'acharner consciencieusement sur 25% de la puissance disponible. Utiliser le reste, 75% de la capacité de calcul, pour surfer sur internet fait un peu gaspillage.

Mais l'affaire n'est pas simple. Il est impossible de mettre en place une démarche générique qui serait valable quelle que soit la méthode d'apprentissage utilisée. Pour une technique donnée, décomposer un algorithme en tâches que l'on peut exécuter en parallèle est un domaine de recherche à part entière. Les publications scientifiques regorgent de propositions en tous genres, tant au niveau méthodologique (modification des algorithmes) qu'au niveau technologique (implémentation sur les machines). Une grande majorité d'entre elles s'intéressent surtout à l'implantation sur de gros systèmes. Il y a très peu de propositions de solutions légères que l'on peut introduire facilement sur des logiciels destinés aux ordinateurs personnels.

Dans ce didacticiel, je propose une solution basée sur les threads. Le multithreading présente plusieurs avantages décisifs dans notre contexte. (1) Sa gestion est largement simplifiée dans les langages de programmation modernes. Concernant Delphi, le langage de développement que j'utilise, il s'agit simplement de surcharger une classe de base qui nous décharge d'une foule de tâches de bas niveaux destinées à sécuriser l'exécution. (2) Les threads peuvent facilement communiquer entre eux. Ils peuvent également manipuler des objets communs, en lecture et en écriture. Des mécanismes de synchronisation efficaces permettent de se prémunir des incohérences. (3) Il est possible de lancer plusieurs threads même s'il n'y a qu'un seul processeur (et un seul cœur) sur la machine. Il n'y a aucun avantage à le faire, mais en tous les cas, le système ne plante pas. (4) Sur un système multi-cœurs, les threads savent se dispatcher automatiquement sur les cœurs. On exploite au mieux les possibilités du système lorsqu'on lance autant de threads qu'il y a de cœurs sur la machine. Même si, par ailleurs, du fait de l'obligation de synchronisation entre les threads, le temps de calcul n'est pas divisé par le nombre de cœurs dans ce cas.

Dans la section suivante, nous présentons les modifications que nous avons introduites dans l'algorithme d'induction d'arbres de décision pour bénéficier de l'apport du multithreading. Nous montrons les gains obtenus sur la base covertype en faisant varier le nombre de threads sur notre machine équipée d'un Intel Q9400 quad-core cadencé à 2.66 Ghz. Dans la section 3, nous montrons la mise en œuvre du nouvel algorithme implanté dans le logiciel **SIPINA version 3.5**. Dans la section 4 et 5, nous réaliserons la même expérimentation à l'aide de Knime version 2.2.2 et RapidMiner 5.0.011, les seuls logiciels grand public à ma connaissance (avec SIPINA aujourd'hui), à proposer une solution multithread pour la construction des arbres de décision. Dans la conclusion, nous essaierons d'établir un lien entre l'apport du multithreading et les caractéristiques des données traitées.

2. Multithreading pour l'induction des arbres

Induction multithread

La variante proposée est basée sur l'algorithme CHAID[i]. Elle s'appuie sur un dispositif de pré-élagage pour déterminer la bonne taille de l'arbre. Elle est de ce fait bien adaptée au traitement des grandes bases de données. De toute manière, la phase de post-élagage de type C4.5 ne nécessite pas d'accès aux données. Elle est très rapide et ne nécessite pas d'optimisation particulière. Récemment, des auteurs (Aldinucci et al., 2010) travaillant sur l'accélération de C4.5 via le multithreading ont conclu qu'il fallait se concentrer sur la phase d'expansion pour améliorer le temps de calcul dans le contexte multi-cœurs (growing phase en anglais[ii]). C'est la lecture de leur article d'ailleurs qui m'a convaincu de me lancer dans l'implémentation d'une solution pour CHAID. Cela faisait plusieurs années que le sujet me hantait. Mais faute de matériel adéquat, introduire du multithread pour des machines mono-processeur et mono-cœur ne sert strictement à rien, j'avais laissé l'idée dans des cartons.

Durant la phase d'expansion, pour chaque nœud à traiter, nous cherchons la variable de segmentation la plus pertinente, celle qui maximise le critère d'évaluation des partitions. Grosso-modo, l'algorithme peut être résumé comme suit.

```
Fonction Segmenter (liste de variables candidates) : variable sélectionnée
Max = -∞
Variable choisie = NULL
Pour chaque variable candidate
      Pertinence = Evaluer (variable)
      Si (Pertinence > Max) Alors
            Max = Pertinence
            Variable choisie = variable
      Fin Si
Fin Pour
Renvoyer (Variable choisie)
```

L'évaluation des variables peut être lancée de manière indépendante. Cependant, il ne s'agit pas de produire simplement un thread par variable. Si ces dernières sont très nombreuses, nous pénaliserions inutilement le système en le saturant. Il est plus judicieux d'utiliser un système de file d'attente. Chaque thread qui se termine vérifie qu'il y a encore des variables à traiter et, si le nombre maximum de thread à lancer n'est pas atteint, démarre l'exécution suivante. Ajuster le nombre de threads au nombre de cœurs est le plus profitable comme nous le verrons lors de l'expérimentation. SIPINA propose ce paramétrage par défaut. Néanmoins l'utilisateur peut modifier le nombre de threads à lancer à sa guise. Par exemple pour consacrer une partie des ressources à d'autres logiciels qu'il lance en parallèle.

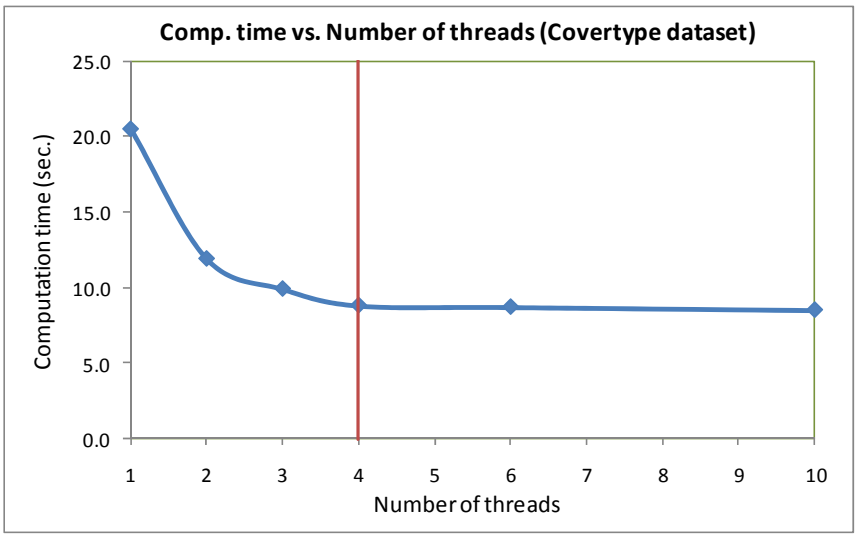
La réduction du temps de calcul n'est pas linéaire par rapport au nombre de cœurs du processeur. La première raison est que les threads manipulent la même variable cible. Mes connaissances systèmes ne sont pas suffisantes pour évaluer toutes les pertes de temps que cela génère. La seconde raison est plus contraignante encore. Pour détecter la variable maximisant la pertinence, les threads doivent mettre à jour des objets communs. Il ne s'agit surtout pas de produire inopportunément des incohérences, à savoir attribuer la pertinence maximum à une mauvaise variable. Pour s'en prémunir, j'utilise des sections critiques dans le code. Il est évident que cette nécessaire synchronisation génère des cycles d'attentes lors de l'exécution de l'algorithme. De fait, nous constaterons que l'adjonction d'un cœur supplémentaire dans les calculs entraîne une réduction marginale de temps d'exécution moins importante. Et, bien évidemment, lorsque le nombre de threads excède le nombre de cœurs disponibles, le temps d'exécution ne change absolument pas (Aldinucci et al., 2010).

Bien sûr, plusieurs pistes ont été explorées avant d'aboutir à la solution que nous avons implantée dans SIPINA version 3.5. Parmi les moins farfelues, je citerai une tentative de regroupement des variables par threads. Elle s'est avérée moins avantageuse. Apparemment, la réduction du nombre de threads créés, censée faire gagner du temps, est défavorablement contrebalancée par l'attente induite par la synchronisation des accès aux objets communs.

Expérimentation sur les données « covertype »

Nous avons utilisé les données « Forest Cover » (covertype) pour évaluer l'intérêt de la solution implantée dans SIPINA version 3.5. La base décrit 581.012 observations à l'aide de 54 variables. La variable cible possède 7 modalités. La base n'est pas faramineuse mais, pour un ordinateur personnel, c'est déjà conséquent. Nous recensons dans le graphique suivant l'évolution du temps de construction d'un arbre, dont le nombre de niveaux est volontairement limité, en fonction du nombre de thread demandé sur une machine

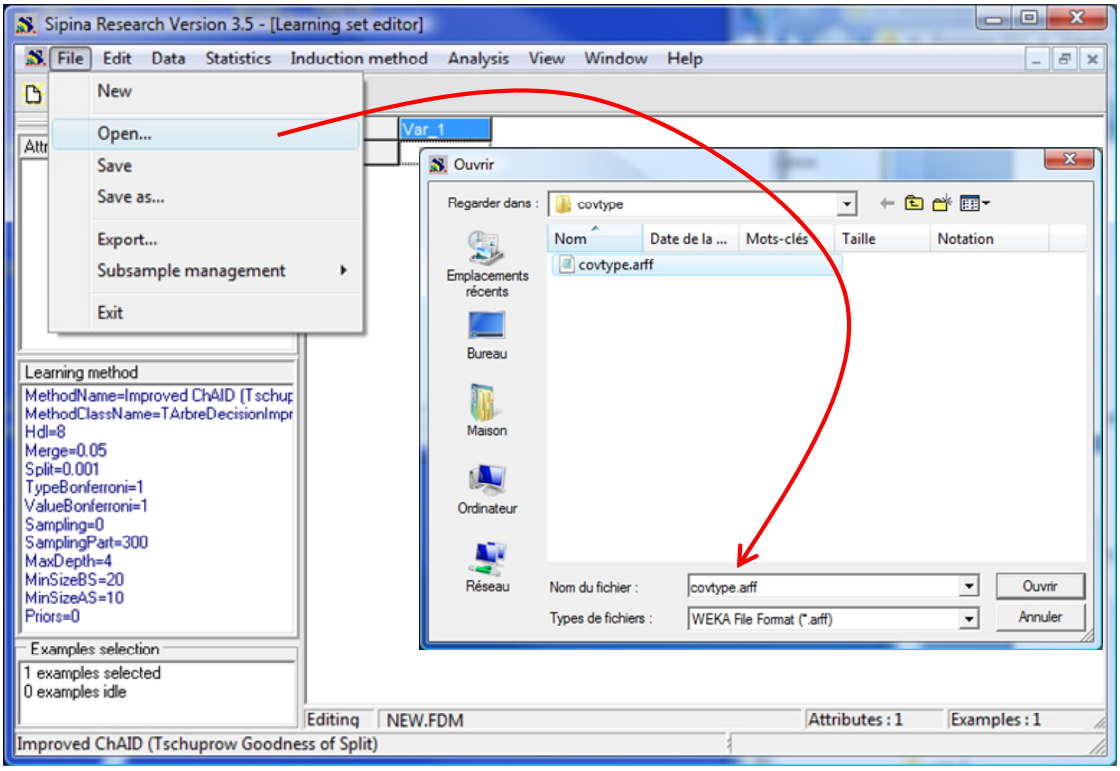
possédant 4 cœurs. Tant que l'on reste dans la limite des cœurs disponibles, l'adjonction d'un thread supplémentaire permet de réduire le temps de calcul. Après, effectivement, que l'on passe à 6 threads ou à 10 threads, il n'évolue plus.



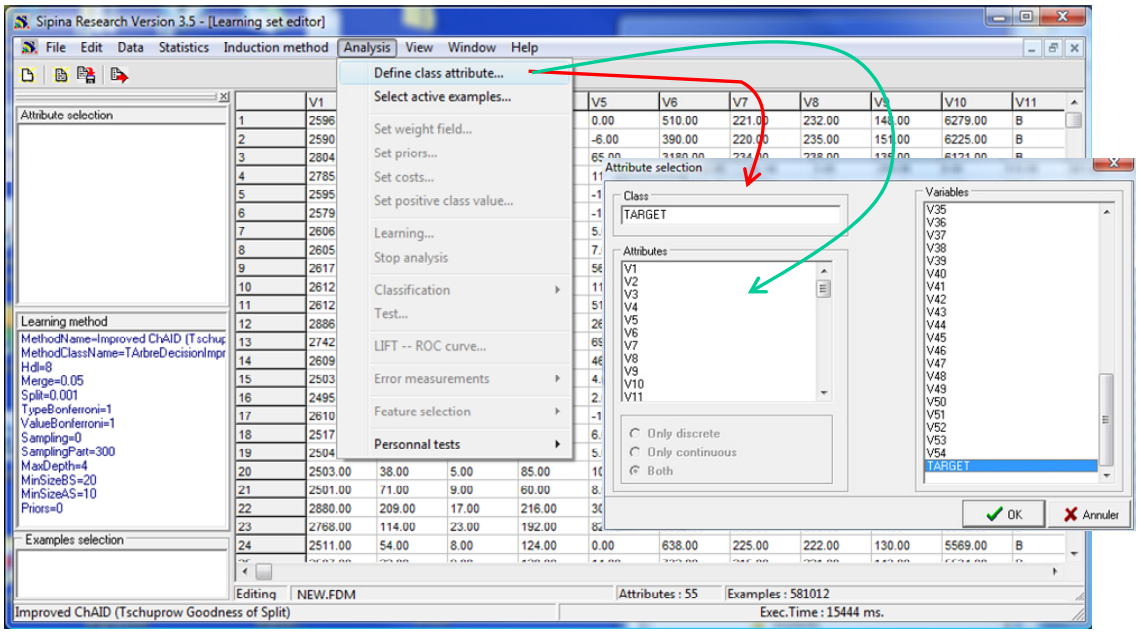
3. Multithreading avec SIPINA 3.5

Chargement des données et définition du problème

Après avoir démarré SIPINA, nous actionnons le menu FILE / OPEN pour charger la base « covertype.arff » au format ARFF (format Weka, il s'agit d'un format texte avec des balises supplémentaires pour décrire les variables).

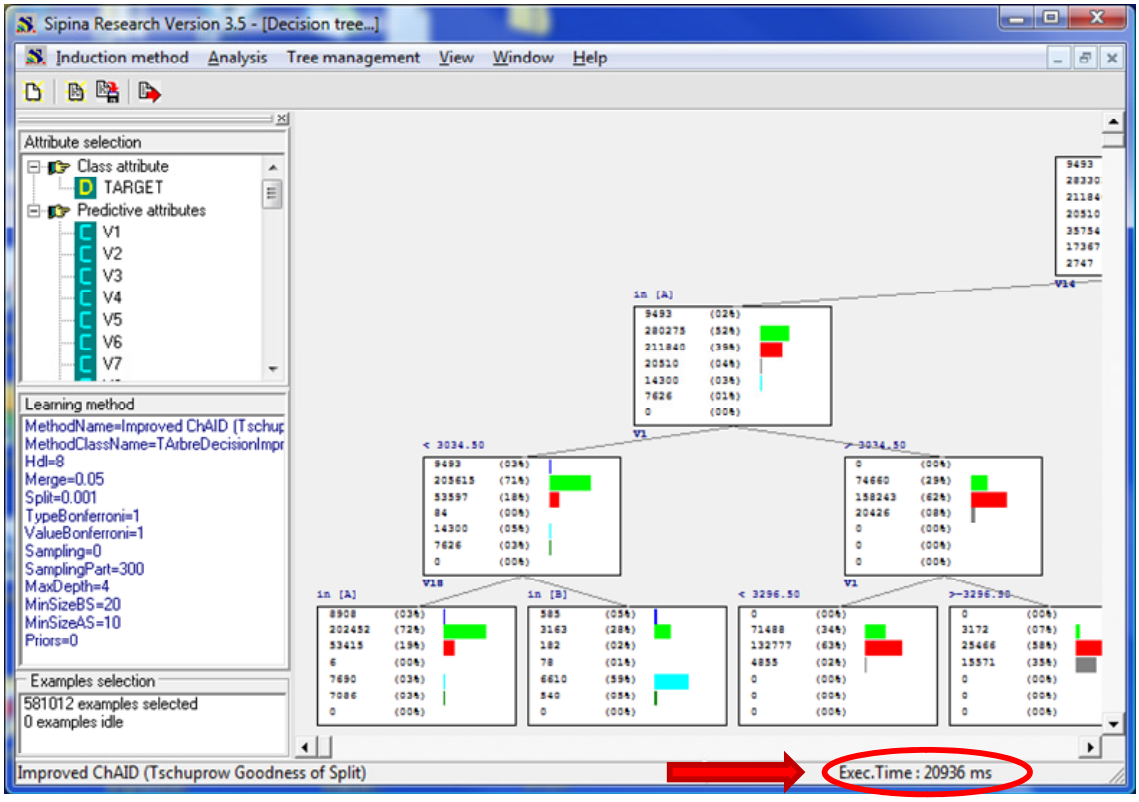


Au bout de 15 secondes, les données sont chargées. Nous cliquons sur le menu ANALYSIS / DEFINE CLASS ATTRIBUTE pour spécifier la variable cible (TARGET) et les prédictives.

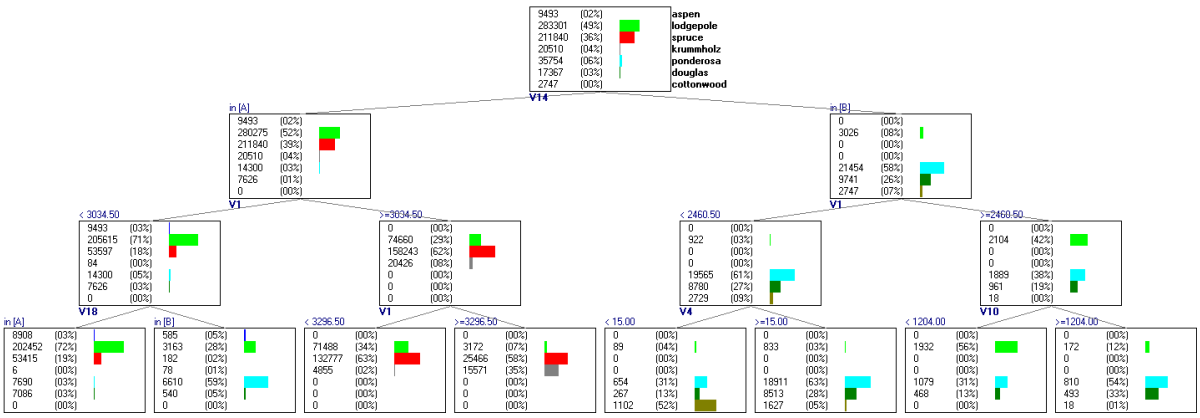


Arbre de décision (ChAID) monothread

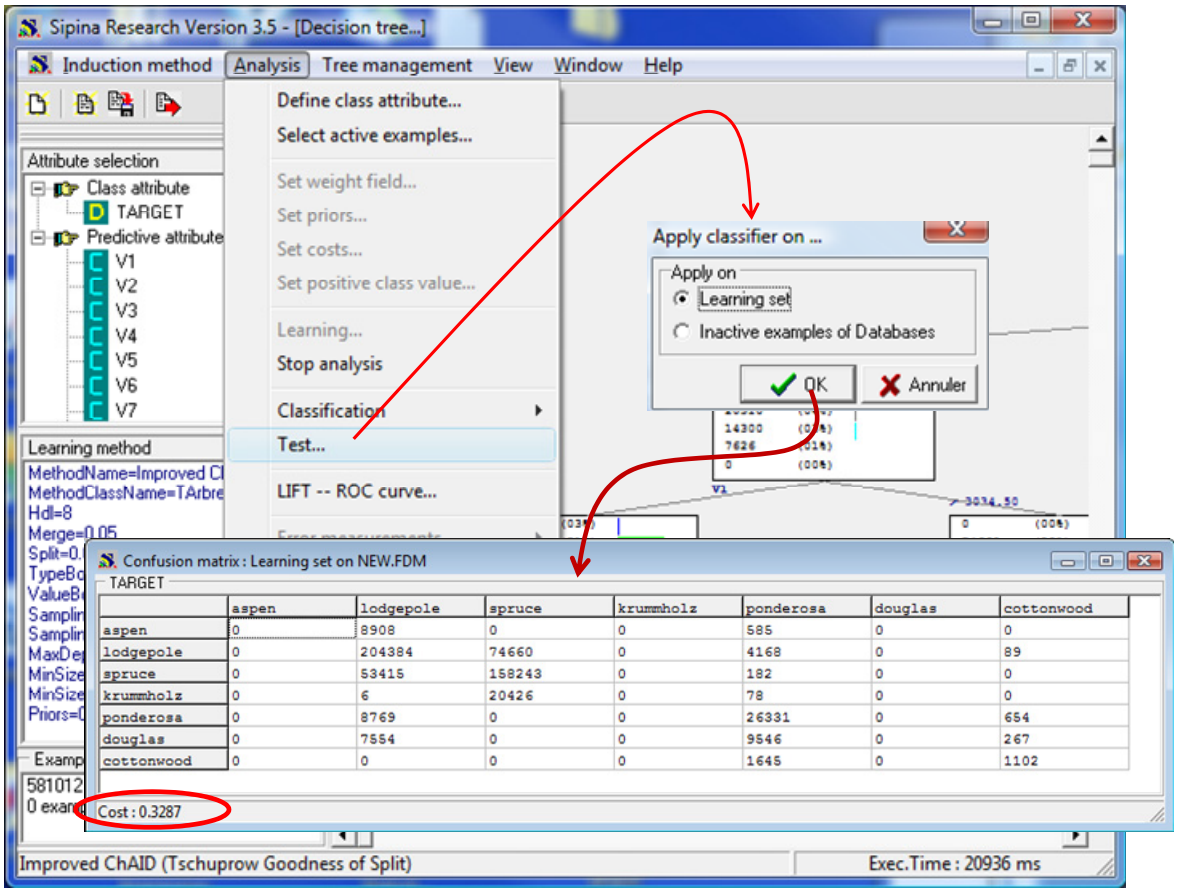
Par défaut, SIPINA utilise la méthode CHAID monothread en limitant le nombre de niveaux de l'arbre à 4. Ce paramétrage est amplement suffisant dans une phase exploratoire. Nous actionnons directement le menu ANALYSIS / LEARNING.



Au bout de 20.9 secondes, l'arbre s'affiche. Voici le détail de l'arbre.



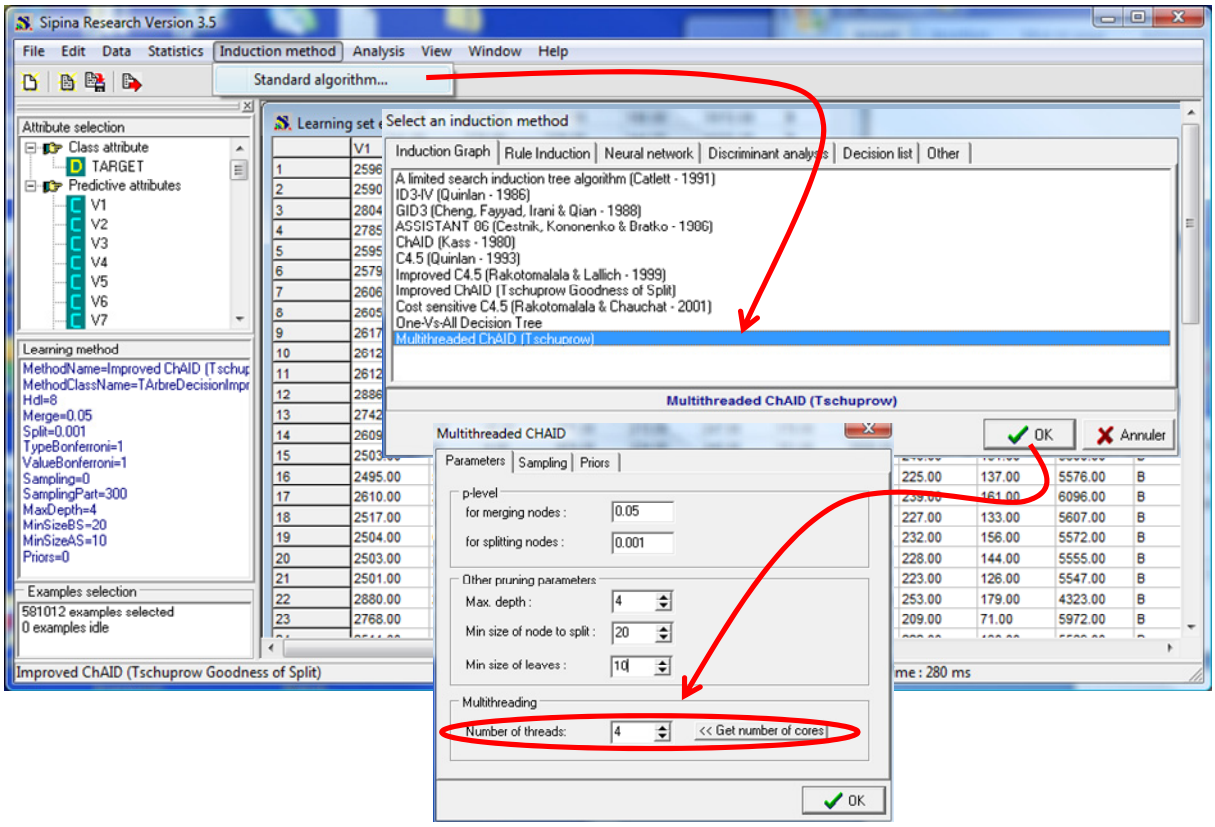
Nous calculons la matrice de confusion sur les données d'apprentissage pour avoir un élément de comparaison lors de la construction multithread des arbres. Nous cliquons sur le menu ANALYSIS / TEST et nous sélectionnons l'option LEARNING SET dans la boîte de paramétrage.



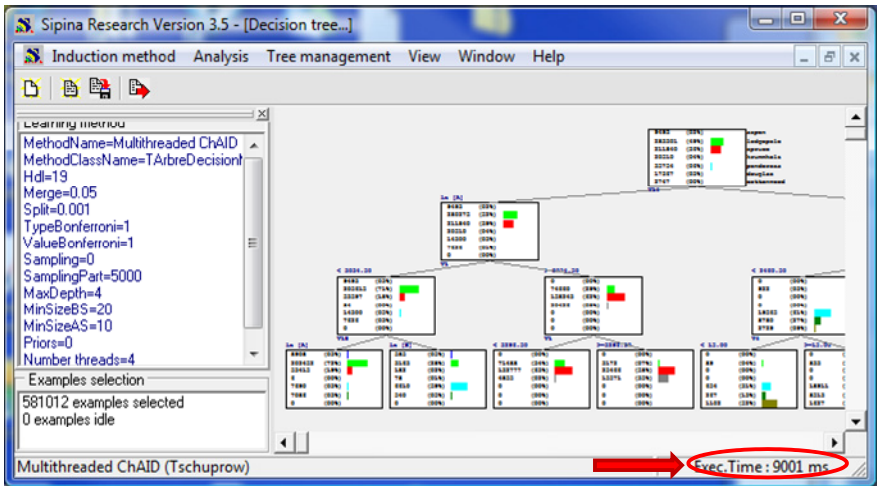
Le taux d'erreur est de 32.87%. Qu'importe la valeur, il ne s'agit pas tellement de mesurer les performances en prédiction mais surtout de vérifier que nous obtenons bien les mêmes arbres par la suite.

Arbre de décision (ChAID) multithread

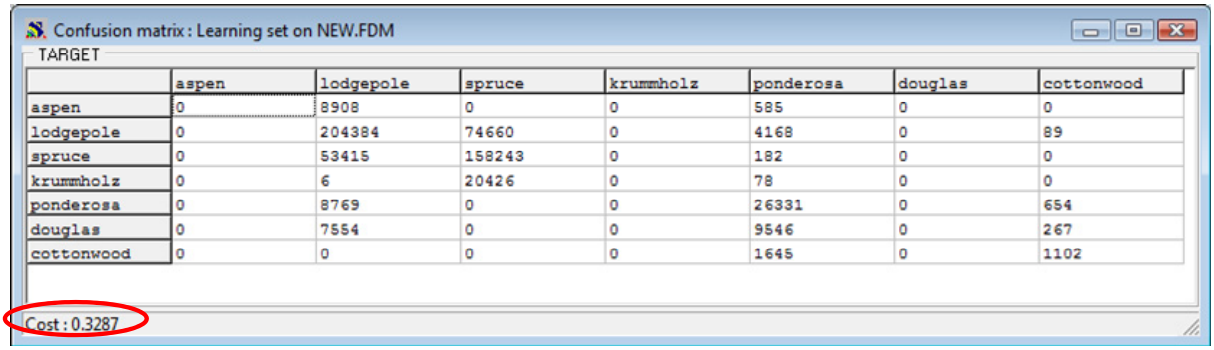
Nous stoppons l'analyse courante en actionnant le menu ANALYSIS / STOP ANALYSIS. Nous choisissons un nouvel algorithme d'apprentissage en cliquant sur INDUCTION METHOD / STANDARD ALGORITHM. Dans la boîte de dialogue qui apparaît, nous sélectionnant MULTITHREAD CHAID (TSCHUPROW). Nous cliquons sur OK. La fenêtre de paramétrage apparaît. Le bouton GET NUMBER OF CORES détecte automatiquement le nombre de cœurs sur notre machine, 4 en ce qui concerne mon matériel. Nous validons.



De nouveau, nous actionnons le menu ANALYSIS / LEARNING pour lancer la construction de l'arbre. Il est affiché au bout de 9 secondes.



En inspectant l'arbre, nous constatons qu'il s'agit bien du même que précédemment. Résultat confirmé par le calcul de la matrice de confusion.

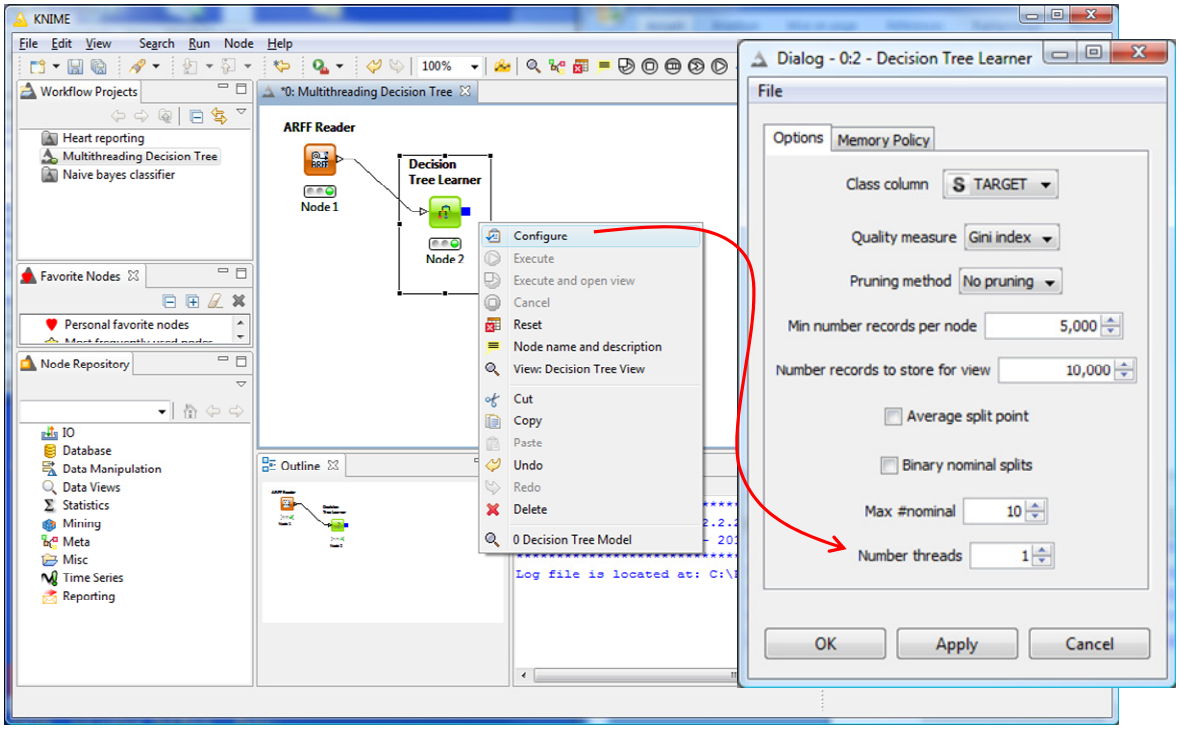


	aspen	lodgepole	spruce	krummholz	ponderosa	douglas	cottonwood
aspen	0	8908	0	0	585	0	0
lodgepole	0	204384	74660	0	4168	0	89
spruce	0	53415	158243	0	182	0	0
krummholz	0	6	20426	0	78	0	0
ponderosa	0	8769	0	0	26331	0	654
douglas	0	7554	0	0	9546	0	267
cottonwood	0	0	0	0	1645	0	1102

4. Multithreading avec Knime 2.2.2

Knime est un logiciel que je suis régulièrement. Il intègre des fonctionnalités très intéressantes. C'est un des rares (le seul à ma connaissance, mais je ne peux pas tout savoir) à proposer le multithreading dans l'induction des arbres de décision.

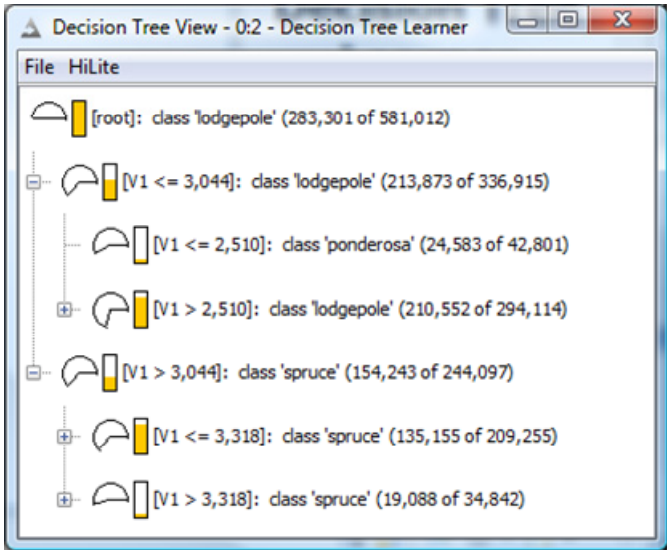
J'ai défini le diagramme suivant, toujours à partir de la base « covertype ». Voyons le paramétrage du composant de construction d'arbres.



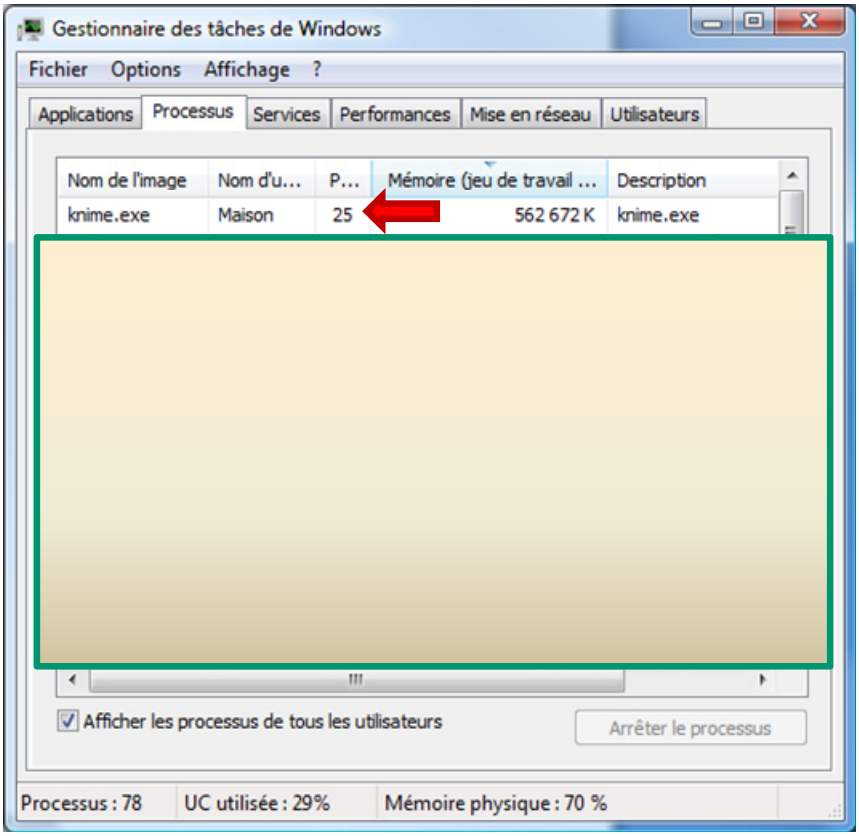
Dans un premier temps, nous demandons un seul thread pour la construction de l'arbre. Nous avons volontairement restreint la taille de l'arbre en demandant 5000 observations au minimum sur les feuilles. Qu'importe. Le plus important pour nous est de mesurer le différentiel de temps de traitements en fonction du nombre de threads.

Pour mesurer le temps d'exécution, nous utilisons un chronomètre puisque Knime ne donne aucune indication à ce sujet.

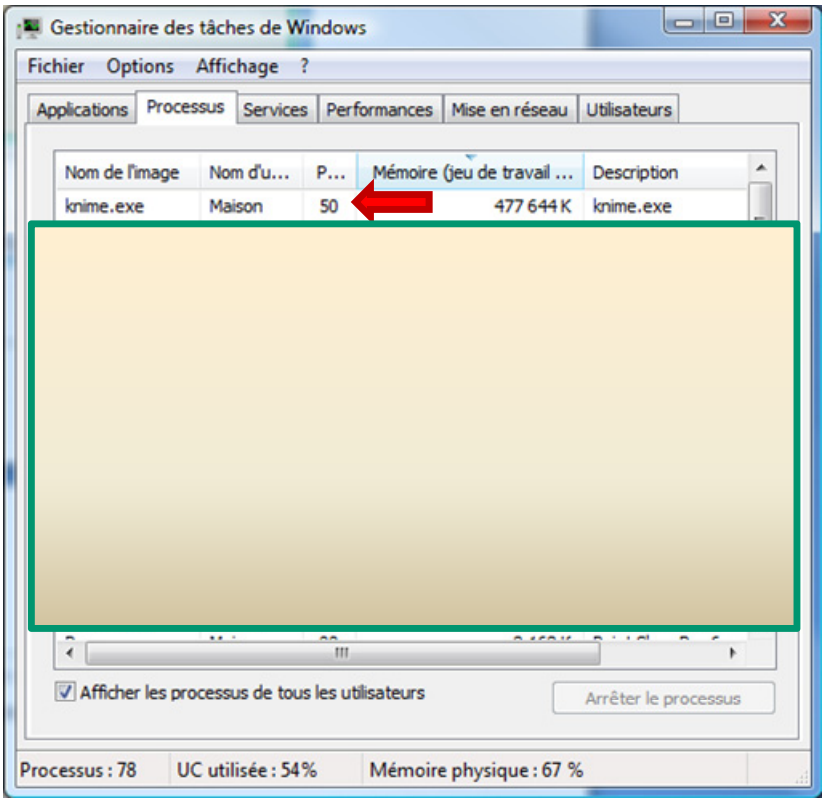
Au bout de **110 secondes** (la méthode et l'arbre obtenu étant différents de SIPINA, les temps calculs ne sont pas comparables), nous obtenons l'arbre dont voici les premiers sommets.



Mais, surtout, en surveillant le Gestionnaire de tâches de Windows, nous notons que Knime, en conformité avec le paramétrage que nous avons spécifié, n'exploite qu'un cœur.



Voyons maintenant ce qu'il en est lorsque nous demandons 4 threads. Nous modifions les paramètres (NUMBER THREADS = 4) et nous relançons les calculs.

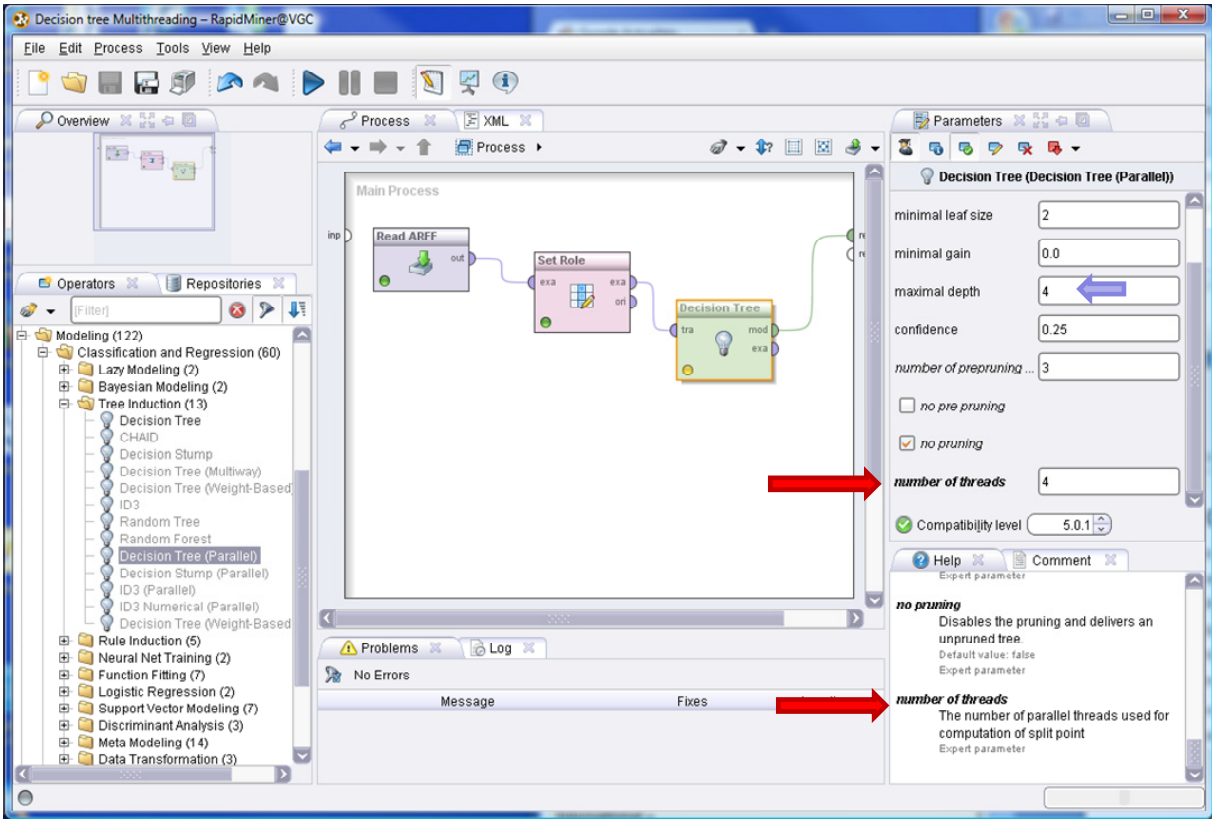


Le temps de calcul est passé à **85 secondes**. Curieusement, Knime exploite seulement 50% de la puissance disponible. Comme si deux threads uniquement étaient lancés. On peut le comprendre en partie. La nécessaire synchronisation des threads font qu'on ne peut pas exploiter tout le temps la totalité des cœurs, même si nous avons demandé 4 threads. Mais il y a une autre explication. Il semble que Knime implémente différemment le multithreading lors de l'induction de l'arbre. Pour l'avoir moi-même expérimenté, j'imagine qu'il affecte aux threads le traitement des nœuds dans sa globalité (et non pas le traitement des variables à l'intérieur d'un nœud). Comme l'arbre est binaire, deux threads seulement sont créés à chaque étape durant le processus d'apprentissage. Mais cela reste une simple supposition. Il faudrait se plonger dans le code source pour en avoir confirmation. Avis aux férus d'informatique.

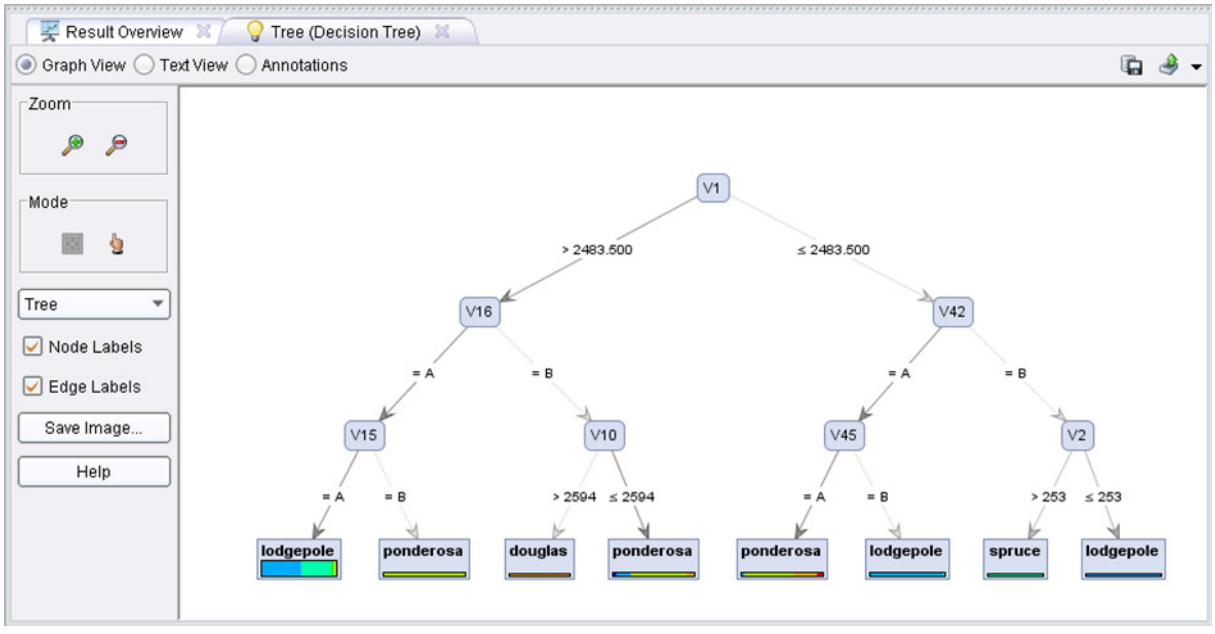
5. Multithreading avec RapidMiner 5.0.011

RapidMiner 5.0.011 intègre le multithreading dans l'induction des arbres de décision. Il propose à la fois les versions classiques et les versions « parallèles » des algorithmes d'apprentissage. La comparaison des performances est facilitée. Il n'affiche que le temps de calcul global en revanche, nous ne fournirons que le différentiel de temps de calcul entre le processus mono-thread et celui avec 4 threads.

Nous avons élaboré le diagramme suivant dans RapidMiner. Nous avons paramétré l'apprentissage de manière à produire un arbre à 4 niveaux au maximum.



Nous obtenons l'arbre suivant à la sortie.



Avec un seul thread, le temps de calcul global, incluant le chargement des données (toujours mono-threadé lui), est de 87 secondes. Avec 4 threads, il passe à 57 secondes. Nous observons donc un gain effectif de 30 secondes lors de la construction de l'arbre. Dans le gestionnaire de tâches Windows, nous constatons que les 4 cœurs du processeur sont

réellement utilisés. Si on se réfère à la documentation, il apparaît que les stratégies multithreading de RapidMiner et de Sipina sont assez similaires.

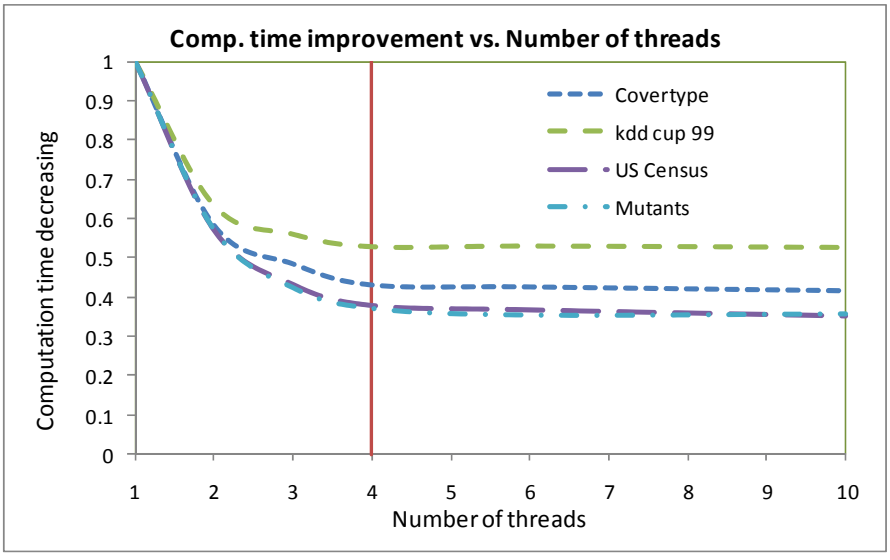
6. Conclusion

Dans ce tutoriel, nous avons montré que le multithreading judicieusement incorporé au processus d'induction d'arbres de décision permet de réduire le temps de calcul sur des machines équipées de processeurs multi-cœurs. Il reste cependant deux aspects importants à explorer : le gain dépend certainement des caractéristiques de l'arbre induit (profondeur, nombre de nœuds) ; il dépend aussi des caractéristiques des données traitées (nombre d'observations, nombre et type des variables prédictives).

Concernant le second thème, nous avons testé le comportement du multithreading sur plusieurs bases de taille plus ou moins importante.

Dataset	# Instances	# all predictors	# continuous predictors
Covertype	581012	54	10
Kdd-cup 99	4817099	41	0
US Census 1990	2458285	67	67
Mutants	16592	5408	5408

Rapporté au temps de calcul de l'algorithme n'utilisant qu'un seul thread, nous observons la décroissance relative du temps d'exécution selon le nombre de thread dans le graphique suivant (l'arbre CHAID est limité à 4 niveaux toujours).



Ces expérimentations confirment que mettre plus de threads que de cœurs (du processeur) ne sert à rien. Apparemment, l'augmentation du nombre de threads (dans la limite de nombre de cœurs disponibles) est surtout bénéfique pour les bases comportant une proportion élevée de descripteurs continus (US Census 1990, Mutants), gourmands en

ressources CPU. En effet, le traitement de ces variables lors de la segmentation d'un nœud, concernant la stratégie programmée dans SIPINA en tous les cas, nécessite le tri des observations et la recherche du point de discrétisation optimal.

[i] Rakotomalala, « Arbres de décision », Revue Modulad, N°33, juin 2005.
http://www-roc.inria.fr/axis/modulad/archives/numero-33/tutorial-rakotomalala-33/rakotomalala-33-tutorial.pdf

[ii] Aldinucci, Ruggieri, Torquati, « Porting Decision Tree Algorithms to Multicore using FastFlow », Pkdd-2010.
http://www.di.unipi.it/~ruggieri/Papers/pkdd2010.pdf