PROJET PARALLEL COMPUTING

Sujet 2

Christopher Houbeiche, Sara Touzani

**Introduction**

Dans le cadre du cours de parallel computing on est amené à optimiser du code R de la librairie enercast qui nous a été mise à disposition sur github. Il nous fournit des outils essentiel pour la prévision de la consommation d’électricité basée sur des séries chronologiques.

Le code d'expériences est constitué de 7 fichier R qui font appel à différent modèle de prédiction (e.g. Saturated Space Model, Kernel Wavelet Functional, Time Series Benchmark, modèle de régression multiple, …). Certaine fonctions du package sont déjà parallélisées comme par exemple le Saturated Space Model ou le Random Forest. Le package enercast a été chargé directement du dépôt de github avec la commande sous R “devtools::install\_git(“cugliari/enercst”)”. Pour charger les fonctions dans notre projet et les utiliser on était obligé d’utiliser les commandes de type “enercast:::predict...” et “enercast:::predict…” pour parcourir toutes les fonctions et les exécuter une fois que la librairie est chargée.

On va alors chercher à optimiser les codes d'expériences avec les modèles qui n’ont vu aucune optimisation.

Le rapport est organisé selon les intitulés des code d'expériences qu’on a cherché à optimiser.

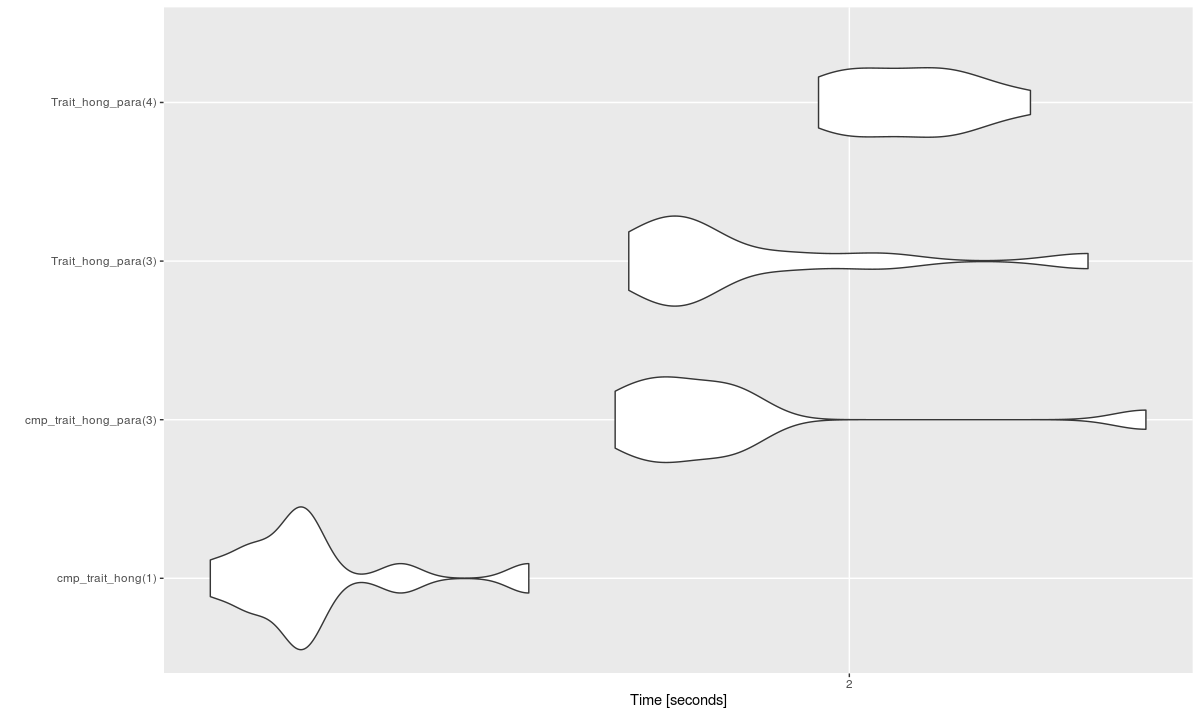
01\_HVB.R

La première approche de prédiction est basée sur un modèle hvb qui nous permet de créer le modèle à chaque itération (une itérations est équivalente à 24h) puis on calcule la prédiction du load d'électricité pour les 24 heures. Finalement, on injecte les données de test traités en train.set pour construire le nouveau modèle lors de l'itération suivante

Lors de l'analyse du temps d'exécution du 01\_HVB.R on c’est aperçu que l'instruction permettant de construire le modèle Hong(Train.set) est celle qui prend le maximum du temps d'exécution environ ~3.9 sec par rapport à la boucle principale qui prend 40 sec pour 10 itérations du coup le travail s'est focalisé en premier degré :

* À optimiser Hong :
  + En externalisant la conversion des colonnes heures et mois en chaîne de caractères et l'appliquer sur dat$hour et dat$month
  + En utilisant un modèle linéaire “lm” optimisé comme celui du package “speedglm” qui a un temps d'exécution de ~0.3 sec ou “FastLm” du package “Rcppgsl” écrit en C++ (avec les données convertis en format numérique) qui prend ~0.1 sec en total pour s'exécute. Or, le calcul ne retourne pas un paramètre qr dû au calcule de qr(x). Notre dernière solution était de réécrire le lm (l\_m) en enlevant les parties du code (conditions toujours vérifié)
  + Ainsi de créer une fonction cmp\_lm qui est celle compilé de l\_m
* En parallélisant la boucle principale en injectant les données de test pour l'itération allant de 2 à n pour i directement (i=2:n) pour enlever la dépendance de l'itération i de celle qui la précède, or cette approche peut s'avérer chère en terme de temps d'exécution vu qu'il faut envoyer pour chaque itération la matrice des données à traiter
* En créant de nouvelles fonctions de celles compilés cmp\_trait\_hong et cmp\_hong.
* En parallélisant le calcule de l'erreur mape vu que chaque itération est indépendante de celle qui la précède
* En essayant d'optimiser davantage notre code en utilisant le package big memory s’est avéré inutile vu qu'on sera obligé de développer la big matrice avec x[,]
* Avoir accès à la fonction mape à travers performance du même package pour calculer le mape

Ainsi on est parti sur utiliser la fonction “cmp\_trait\_hong()” qui traite les données sous forme séquentiel. Pour 10 itérations, on est arrivé à optimiser le temps d'exécution de la boucle principale de 40 secondes à 2 secondes en parallèle.



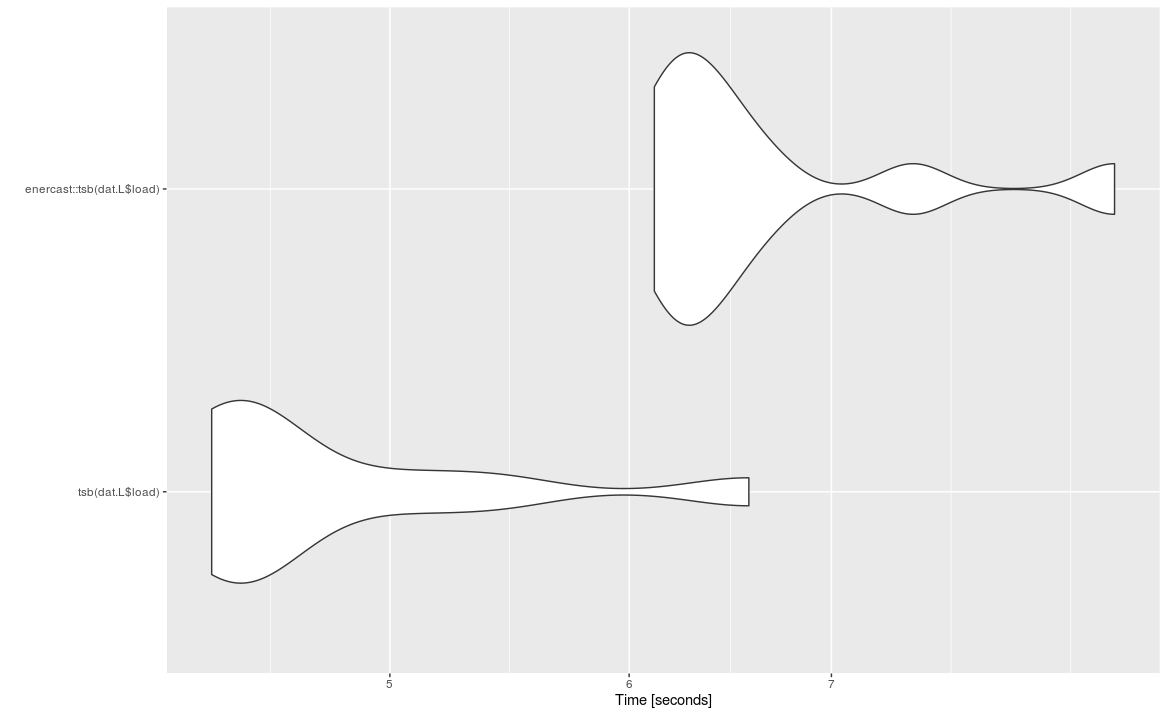
On a aussi comparer les différents temps exécutions sur une machine plus puissante à 8 coeurs où on a exécuté la fonction “cmp\_trait\_hong()” sur la durée total de 365 iterations. On est passé de 14 minutes et 50 secondes en séquentielle à 11 minutes et 42 secondes pour 7 coeurs de temps d'exécution.

02\_TSB.R

La deuxième approche implémentée dans 02\_TSB.R similaire à 01\_HVB.R dans le traitement effectué dans la boucle principale mais cette fois utilise un modèle construit avec la fonction tsb qui prend la grande part du temps d'exécution et qui est basée sur la minimisation de la fonction loss avec la fonction nlminb où on a:

* Éviter de paralléliser la boucle principale en se basant sur les résultats de l'optimisation du 01\_HVB.R
* Optimisé le tsb on éliminant les conditions qui sont toujours vérifiée, ainsi en utilisant le à la place du nlminb optima avec la méthode "BFGS" et nlm en ajustant la tolérance du calcul
* En parallélisant le calcul de acf et acft effectué dans la fonction loss au sein de losss\_parallel avec la fonction “parallel\_acf\_acft” qui permet de calculer en parallèle les deux instructions :
  + acf1 <- acf(ytc, lag.max = k, plot = FALSE)$acf[-1]
  + acfteo <- enercast:::acft(ar = ar, AR1 = AR1, AR2 = AR2, ma = ma, MA1 = MA1, MA2 = MA2, s1 = s1, s2 = s2, k = k)
  + Or ceci c'est avéré couteux en temps d'exécution du coup on est resté sur utiliser un loss nom parallélisé
* Utiliser des fonctions compilés afin de réduire le temps d'exécution
* Paralléliser le calcul de l'erreur mape

Ainsi on a pu améliorer la fonction tsb comme suit :



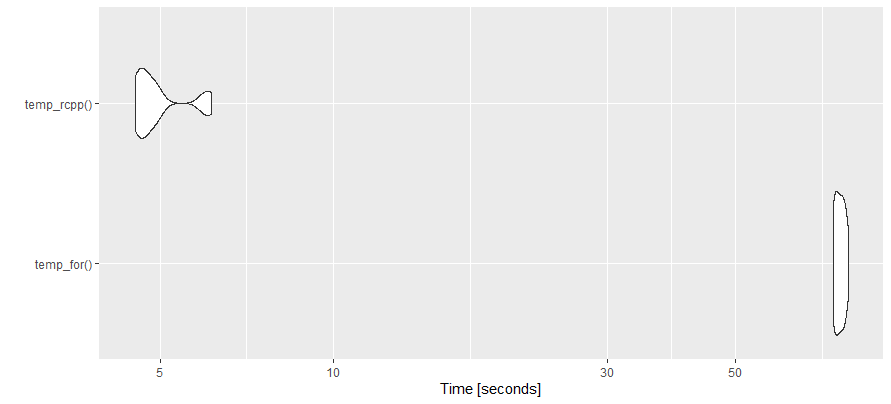
04\_SSM.R

Pour la partie de SSM “State Space Model” on a optimisé que la première boucle “for” pour le calcule des deux variables de temps “DAT$tempM24” et “DAT$tempm24” avec du Rcpp. Pour pouvoir réécrire la boucle R en C++ on devait tenir compte qu’on ne peut pas accéder aux variables de la matrice avec l'opérateur “:” (DAT$temp[1:365]).

On a alors utilisé la fonction range pour accéder aux différentes valeurs à l'intérieur des matrices pour pouvoir ensuite trouver le maximum et le minimum. Le “ifelse” a été remplacé par des boucles conditionnelle if() et else().

La fonction Rcpp en fin d'exécution retourne deux matrices “tempM24” et “tempm24” qu’on a choisi de les stocker sous forme de liste nommée “ret”.

Le graphique ci-contre montre le gain en temps d'exécution en passant de la boucle “for” de R en Rcpp étant d’environ 12 fois plus rapide en moyenne avec du Rcpp. Le graphique a été construit en exécutant les deux fonctions 10 fois. Pour le calcul du temps d'exécution de la fonction temps\_rcpp() on a aussi intégrer une recompilation du code C++ à chaque fois pour tenir compte du temps de compilation.



Pour la partie modèle et prédiction la fonction “ssm” utilisée de la librairie Enercast implémente déjà une parallélisation avec un “foreach” d'où la non nécessité de faire des modifications pour réduire son temps d'exécution.

06\_KWF.R

La partie du Kernel Wavelet Functional est basée sur un modèle de prédiction qui fait appel à la fonction “wavethresh”.

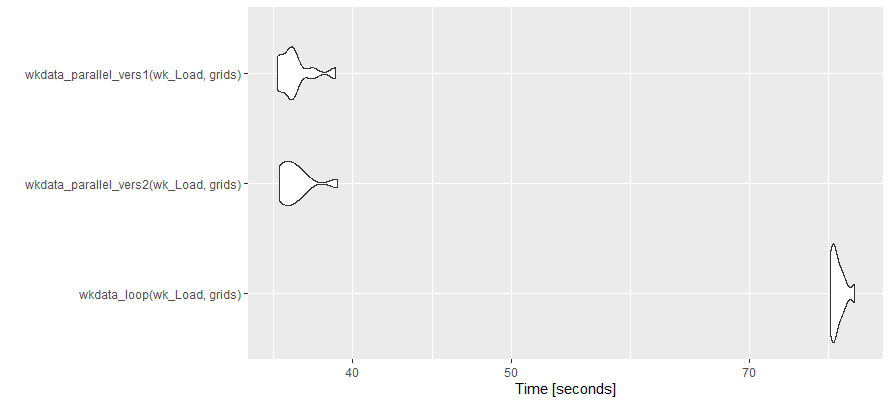
Pour la parallélisation du modèle de prédiction on s'est notamment penché sur la boucle “for” de la prédiction de “wkdata” qui prend un temps d'exécution de 65 secondes environs pour 365 jours. On a alors développé deux versions parallèle du modele de prediction.

La première version utilise comme paramètre pour la boucle foreach “.combine = cbind” qui permet d’effectuer la prédiction de “wkdata” en faisant une liaison des colonnes une par une.

La deuxième version utilise une boucle foreach et a comme paramètre “.combine = c” qui ensuite renvoi le résultat final dans la variable “pred\_version2” d’une dimension de 1 sur 8760. La variable de prédiction finale ayant une dimension non conforme on lui affecte une nouvelle dimension de 24 sur 365 et on la transposer pour que les lignes et les colonnes soit dans le même ordre que la matrice de prédiction de la boucle “for”.

Lors de la parallélisation de la boucle “for” on a aussi modifier l’affectation finale dans la matrice “pred\_Load” pour qu’elle peut prendre en entrer les données sous forme parallèle et non séquentielle. Si la variable d’affectation finale de la prédiction est gardée sans aucune modification “pred\_Load[q,]”, le résultat finale ne sera pas stocké et on aura en sortie que la prédiction pour le première modèle.

Le graphique ci-contre illustre le gain en temps d'exécution en parallélisant la boucle for du modèle de prédiction “wk\_data”. Finalement les deux fonctions parallel reste à peu près les mêmes en temps d'exécution. On va alors se contenter de garder uniquement la fonction “wkdata\_parallel\_vers1”. La fonction “wkdata\_parallel\_vers1” étant la version “foreach” parallel qui utilise comme paramètre “.combine = cbind”.

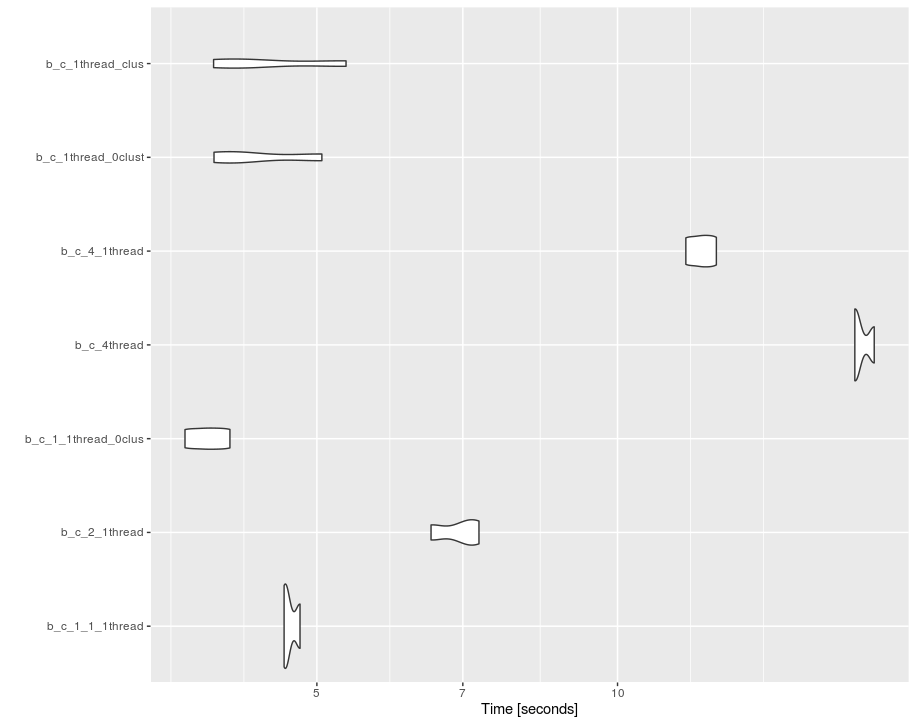


Le graphique a été construit en exécutant les trois fonctions 10 fois.

07\_GAM.R

La dernière approche qu'on a étudié était d'implémenter dans 07\_Gam.R une méthode de traitement similaire aux autres où le modèle construit avec la fonction bam prend la plus grande partie du temps d'exécution. Il utilise en principe des paramètres propres à la fonction comme le parallélisme avec les clusters et la notion du multithreading

On a cherché à étudier le comportement de la fonction en lui donnant à chaque fois différents paramètres pour contrôler le multithreading et le parallélisme cette étude se résume sur le graphique suivant :



Ou l'on remarque que le meilleur des résultats apparaît lorsque le paramètre nthreads est bas nthreads = c(1,1) et pas lorsqu’il est de 4. Faire appel aux clusters ne fait que augmenter le temps d'exécution tandis que le blas de référence qu'on utilise est en singlethread, une autre alternative de réduire le temps d’exécution modestement que ce qu'on a obtenu est en utilisant le MKL BLAS qui est multithreaded par défaut ( où il faut quand on est sous Linux mettre la variable d'environnement MKL\_NUM\_THREADS=1 avant de lancer sous mode monothread)

Or, en passant des paramètres de plus dans la fonction de prédiction nous a permis de réduire le temps d'exécution total de 10 secondes “pred <- predict(fit, TEST.SET,discrete = TRUE, nthreads= 4)”.

**Conclusion**

Malgré les nombreux problème rencontrés au cours du projet comme l'incapacité d'exécuter certaine fonction comme dans le fichier de 03\_RFS.R, 04\_SSM.R ou le 05\_SPM.R ces fonctions n'était déjà pour la plupart paralléliser ou même voir charger correctement la librairie enercast. On a eu l'opportunité de voir sur le cadre d’un projet basé sur les séries temporelles l'efficacité de faire appel à des méthodes de parallélisation ou de code Rcpp. On a réussi aussi à optimiser quelque fonction avec grand succès alors que d’autre reste encore toujours relativement lourde en temps d'exécution.

Pour conclure ce projet était une belle opportunité qui nous a permit de renforcer nos compétence technique en optimisation de code et du travail ainsi celle théorique en élaborant le travail de recherche basé sur les méthodes et le code de l’implémentation de Cugliari et Poggi.