REMERCIEMENTS

Mes premiers remerciements vont à Bernard Prum, grande figure de la Statistique française décédée il y a peu, qui m'a ouvert les portes du CNRS et celles du calcul parallèle. Fin 2004, bureau de Bernard : « Bernard, je voudrais m'inscrire à une formation sur le calcul parallèle à Grenoble, es tu d'accord ? ». « Du *parallèle* à Grenoble, je te connais, c'est une formation sur le ski parallèle que tu veux suivre ! ». S'en suivaient de grands éclats de rire. Je dédicace donc ce livre à mon très cher ami Bernard.

Je souhaite remercier le groupe Calcul dans son ensemble, *i.e.* les hommes et les femmes qui considèrent que l'entraide fait partie du travail. Bien sûr, j'adresse une mention spéciale à Violaine Louvet et Thierry Dumont, fondateurs du groupe, avec qui j'ai fait un bon bout de chemin à Lyon, au plus près des nouvelles technologies et techniques du calcul, et parfois même les pieds dans l'eau fraîche des Calanques de Marseille !

Mais il n'y a pas de calcul parallèle sans jolis problèmes scientifiques. Je souhaite en particulier remercier Laurent Duret et Simon Penel pour m'avoir fait confiance au moment où nous nous sommes lancés dans le grand bain des algorithmes parallèles.

Je remercie tout particulièrement mon acolyte Franck Picard qui, fin 2011, m'a dit ceci : « Vincent, tu es spécialiste du calcul parallèle, j'ai un package R qui demande des performances, ça te dirait de creuser les aspects du calcul parallèle dans R ? ».

J'adresse par ailleurs mes plus chaleureux remerciements aux membres du pôle informatique du LBBE qui ont toujours su mettre à ma disposition les dernières technologies et la puissance de calcul du CC LBBE/PRABI.

Je remercie également Stéphane Dray, Laurent Jacob, Martyn Plummer et Aurélie Siberchicot, mes collègues lyonnais de l'écosystème R, pour toute la considération qu'ils ont pu apporter aux différentes initiatives que j'ai eues autour de R.

Vincent Miele

Ce livre est l'aboutissement de nombreuses discussions et interactions avec des personnes issues d'horizons différents, unies dans un même besoin et une même envie de partage d'expériences et d'échanges. Je voudrais remercier collectivement ou nommément ces collègues pour la richesse de nos collaborations.

En premier lieu, mes pensées vont à Jacques Laminie qui m'a mis le pied à l'étrier du calcul intensif. Je tiens à remercier tout particulièrement Thierry Dumont pour la longue route que nous avons suivie ensemble, entre combustion et plasmas, entre méthodes numériques et architectures, entre Python et C++, entre Bedlewo et Grenade.

Je souhaite également remercier l'ensemble des personnes qui s'investissent depuis des années dans l'aventure du Groupe Calcul, mettant à la disposition des autres leurs expériences, leurs compétences et leur dynamisme.

Merci également aux chercheurs qui sont à l'origine de collaborations autour de projets scientifiques passionnants : Marc Massot, Stéphane Descombes, Emmanuel

Grenier, Francis Filbet en particulier.
Pour finir, je remercie tout particulièrement toutes les personnes des différents laboratoires que j'ai fréquentés : le laboratoire de Mathématiques d'Orsay, puis l'Institut Camille Jordan à Lyon, pour m'avoir permis un cheminement professionnel particulièrement riche.
Enfin, merci à Vincent pour m'avoir embarquée dans cette aventure !

Violaine Louvet

Les auteurs souhaitent également remercier Pierre-André Cornillon et Eric Matzner-Løber pour leur avoir donné l'opportunité de publier cet ouvrage. Ils remercient également les différents relecteurs anonymes ou pas (Erwan, Rémy, Martial) pour les commentaires constructifs qui ont permis d'améliorer cet ouvrage.

PREFACE

C'est avec un grand plaisir que je vous présente l'ouvrage « Calcul parallèle avec R » par Vincent Miele et Violaine Louvet, qui traite d'un sujet central pour le calcul haute performance dans R. En tant qu'utilisateur de R depuis 1996, et membre de la *R Core Team* depuis 2002, j'ai été témoin de l'augmentation incroyable de la popularité de R, qui constitue aujourd'hui un outil essentiel pour le traitement des données dans de nombreux domaines scientifiques. L'utilisation de plus en plus fréquente de R par des organisations commerciales à l'ère des « big data » s'est traduite par la formation du *R-Consortium* (`http://R-consortium.org`), un groupe d'entreprises du secteur technologique qui se sont unies pour apporter leur soutien à la communauté des utilisateurs de R. Mais surtout, le langage R possède aujourd'hui une communauté mature d'utilisateurs et de développeurs qui ont créé et partagé des milliers de packages via le *Comprehensive R Archive Network* (`CRAN`, `https://cran.r-project.org`).

L'histoire du succès de R a commencé il y a 30 ans. Une grande partie de la conception du logiciel R dérive du langage `S` développé au sein des laboratoires AT&T Bell dans les années 1980. Si on téléportait un utilisateur contemporain de R au début des années 1990, il n'aurait aucune difficulté à travailler avec la version `S3`, même s'il aurait probablement du mal à se passer de `CRAN`. Certaines caractéristiques fondamentales de R remontent à une époque où le paysage informatique était très différent. C'est pourquoi les limites inhérentes à cette conception ancienne finissent par devenir évidentes à un utilisateur cherchant une performance maximale.

R a toujours offert la possibilité d'améliorer sa performance en convertissant le haut niveau d'interprétation du code R en un langage compilé écrit en `C, C++` ou `Fortran`. Cette possibilité a été encore améliorée par les développeurs du package **Rcpp**, lequel offre une intégration harmonieuse entre R et `C++`. **Rcpp** est devenu la partie la plus importante de l'infrastructure de R, en dehors de sa distribution de base ; **Rcpp** est aujourd'hui utilisée par plus de 1200 packages `CRAN`.

Il faut pourtant constater que l'utilisation de codes compilés n'est plus suffisante pour obtenir des performances maximales. Comme Violaine et Vincent l'ont expliqué très clairement dans ce livre, les fabricants ont cessé de chercher à créer des processeurs plus rapides, il y a environ 10 ans, lorsqu'ils ont plutôt choisi d'augmenter le nombre de cœurs computationnels à l'intérieur des processeurs. L'exploitation de ces cœurs multiples exige pour le programmeur l'adoption de techniques de programmation parallèle. En outre, le traitement d'une très grande quantité de données ne peut plus se faire sur un ordinateur de bureau classique, car il exige d'avoir recours à un cluster. Autrefois réservé aux spécialistes de calcul haute performance, les clusters sont de plus en plus répandus grâce au services de *cloud computing*. La répartition des données et l'utilisation efficace de tous les nœuds d'un cluster nécessitent également une programmation parallèle, bien que dans une perspective quelque peu différente.

La programmation parallèle est bien plus « *close to the metal* » que la programmation séquentielle. Elle exige que le développeur soit familier avec l'architecture

du système qu'il utilise et qu'il en connaisse les limites que ce système impose à l'efficacité des programmes informatiques. Voilà pourquoi ce livre est si important. Il traite non seulement des outils de programmation parallèle R et C++ interfacé avec R en utilisant **Rcpp**, mais aussi des concepts fondamentaux de l'architecture de votre ordinateur qu'il vous faut connaître pour mener à bien la programmation. Enfin, il vous apprend à « penser parallèle ».

le 19 avril 2016,
Martyn Plummer

Table des matières

Chapitre 1

A la recherche de performances

La recherche de performance est une démarche qui s'inscrit dans un cadre global. Il est généralement contre-productif de s'attaquer à l'optimisation d'un code sans mettre en place au préalable des éléments d'organisation, souvent de bon sens (gestion de projet, suivi de version), qui permettent d'améliorer considérablement l'efficacité du travail réalisé. Nous proposons dans ce chapitre des pistes à explorer pour une approche à la fois organisationnelle et technique de l'optimisation de code R : optimiser sa façon de programmer et optimiser ses programmes.

1.1 L'organisation de projet

1.1.1 Ne jamais optimiser prématurément

On nomme *optimisation de code* l'ensemble des efforts réalisés pour augmenter ses performances, c'est-à-dire accélérer la vitesse du code et/ou diminuer son empreinte mémoire (*i.e.* l'espace mémoire qu'il requiert). Mais à partir de quelle phase de développement doit-on commencer à se soucier des performances de calcul ? Herb Sutter nous dit : « ne jamais optimiser prématurément » (Sutter & Alexandrescu, 2004). Et de rajouter, « l'exactitude des résultats, la simplicité et la clarté du code passent en premier ». Il est en effet primordial de respecter un rythme et une méthode de développement qui garantissent avant tout que les résultats sont et resteront exacts tout au long de la vie du code R. Exact au sens où on estime les résultats obtenus conformes aux attentes. L'*exactitude* des calculs sur ordinateur n'est pas toujours simple à définir (voir Annexe A.1 sur la problématique du calcul flottant). Pour préserver la « justesse » des résultats, nous proposons de suivre une démarche de développement itératif (voir 1.1.2). Les performances viendront progressivement, après 1/ validation de l'exactitude des résultats et 2/ analyse fine

de l'efficacité de l'implémentation (voir 1.2.1). Bien sûr, le programmeur expérimenté pourra rapidement faire des choix de structures de données, d'algorithmes, de façon de coder qui favorisent de bonnes performances *a priori* (*i.e* sans les mesurer) tout en veillant à garantir en premier lieu l'exactitude. Gardons en tête cette formule :

« On n'est jamais assez surpris
par les bonnes performances d'un code faux. » (les auteurs)

1.1.2 Le processus de développement

S'organiser pour chercher de la performance, c'est d'abord s'organiser pour mener à bien un projet informatique. Le développeur en R pourra s'inspirer librement des réflexions abouties et mises en œuvre dans le milieu du génie logiciel. En particulier, les approches *agile* (terme choisi par opposition à la lourdeur avérée des projets informatiques du siècle dernier) méritent d'être étudiées. En 2001, dix-sept personnalités du génie logiciel signent l'*Agile Manifesto* (Beck *et al.*, 2001) qui repense la gestion de projets de développement en informatique comme à la fois un processus de développement logiciel, un état d'esprit et un ensemble de bonnes pratiques. Plusieurs méthodes dites agiles verront le jour, en particulier l'*Extreme programming* : celle-ci n'a d'extrême que le nom, Kent Beck définissant d'ailleurs la méthode comme « une tentative de réconcilier l'humain avec la productivité »(Beck & Andres, 2004). Nous proposons ici une sélection libre de principes énoncés dans ces méthodes :

— le principe fondamental est le *développement itératif* qui consiste à adopter des cycles de développement (conception, implementation, tests) très courts. Pour la conception, la recherche de la simplicité, de la solution la plus simple, est suggérée. Le code évolue par petites unités fonctionnelles, il est donc recommandé de mettre en place un code modulaire (fonctions, classes ou données les plus indépendantes possibles) ;

— les modifications sont intégrées très fréquemment (*intégration continue*, éventuellement plusieurs fois par jour) et il est indispensable de réaliser un suivi des modifications grâce à un *gestionnaire de version* (voir 1.1.3) ;

— cependant, toutes ces modifications doivent être testées : l'unité de mesure de la progression du projet n'est pas le nombre de lignes de codes mais bien le nombre de lignes qui fonctionnent et donnent un résultat juste. Ainsi, les tests sont la clé de voûte du développement itératif et doivent être implémentés et conçus avant la fonctionnalité qu'ils sont censés tester : on parle de développement piloté par les tests (voir 1.1.3) ;

— les méthodes agiles recommandant la ré-écriture ou la re-conception (*refactoring*) fréquentes de parties du logiciel non optimales : le changement fait partie de la vie du code ;

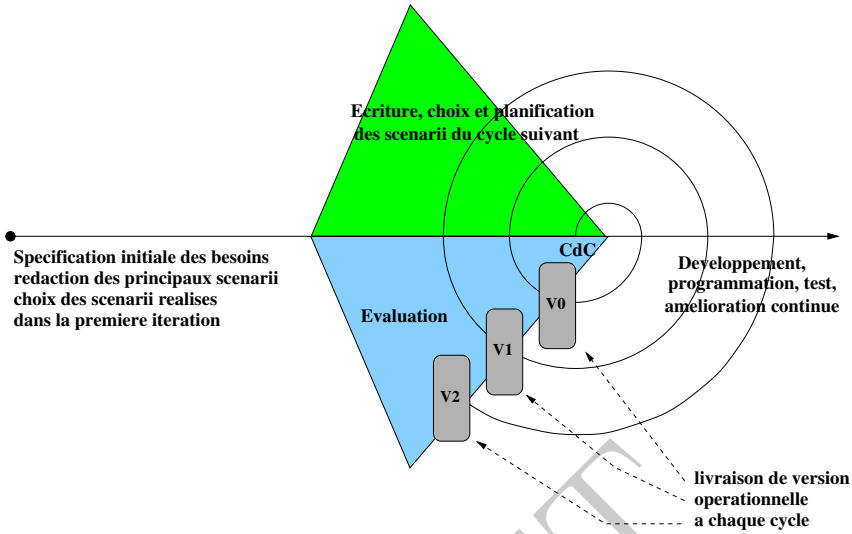


Fig. 1.1 – Illustration du développement itératif, CdC=cahier des charges.

— l'adoption de *conventions de nommage* (règles de choix du noms des éléments du programme) dans le code source est fondamentale (voir 1.1.3) ;

— enfin, l'humain est un facteur fondamental dans la réussite d'un projet informatique. *Primo*, le rythme de développement doit être *durable*, le stress étant considéré comme contre-productif par les promoteurs des méthodes agiles. *Secondo*, les membres de l'équipe (depuis les développeurs jusqu'au responsable du projet) doivent se sentir co-responsables de la vie du logiciel quel que soit leur position dans la chaîne de décision : on parle d'*appropriation collective* du code. Par exemple, le temps de développement associé au *refactoring* doit être accepté par tous et ne doit pas être considéré comme du temps perdu, ni ne doit servir à justifier une chasse aux sorcières contre le développeur qui a implémenté une partie à ré-écrire.

1.1.3 Bonnes pratiques de développement

La mise en place de bonnes pratiques et l'utilisation d'outils *ad hoc* peuvent paraître au premier abord fastidieuses, inutiles et chronophages. Une bonne organisation du développement peut pourtant apporter des gains en temps, en énergie et en sérénité, que l'on soit développeur « isolé » (personne implémentant un code pour son propre usage) ou développeur dans un projet collaboratif.

Les conventions de nommage

Pour R, le développeur pourra s'inspirer des conventions proposées dans la communauté (voir tab. 1.1), mais devra surtout maintenir ces conventions sur toute la durée du projet et dans l'intégralité du code : la rédaction d'un petit document qui fixe les idées peut dès lors être une bonne idée !

Nom de la convention (anglais) `+ example`	Principe
alllowercase `adjustcolor`	tout en minuscule cohérent mais peu lisible
period.separated `plot.new`	minuscules et mots séparés par un point spécifique à R et très utilisé dans le `core`
underscore_separated `seq_along`	minuscules et mots séparés par un tiret-bas recommandé par GNU
lowerCamelCase `colMeans`	première lettre minuscule et mots séparés par une majuscule
UpperCamelCase `Matrix` class	idem mais première lettre majuscule classique pour les noms de classes à combiner avec lowerCamelCase
Bioconductor	`https://www.bioconductor.org/developers/how-to/coding-style/`

Tableau 1.1 – Quelques conventions de nommage : les cinq premières sont présentées dans Bååth (2012) tandis que la dernière convention fait référence au projet Bioconductor.

Le choix des noms de variables et fonctions doit permettre l'*auto-documentation* du code : il s'agit de donner des noms explicites, qui font sens et/ou référence à des éléments présentés dans un article. L'auto-documentation est « agile » car toute modification du code modifie de fait intrinsèquement la documentation (puisque le code est sa propre documentation). A noter que l'anglais est à privilégier, car R est basé sur la philosophie de l'*open source* et le code source doit être lisible par le plus grand nombre « worldwide ».

Dans le code suivant, aucune convention ni cohérence n'est adoptée et, de fait, le code est peu lisible :

```
i <- 1.45 # nommage pas explicite
          # et i mal choisi car entier en général

Y <- 5    # alternance de majuscules-minuscules
          # sans règle
z <- matrix(0,2,2)
```

```
func63 <- function(a, b) return sqrt(a^2+b^2)
          # nom de fonction pas explicite
```

A contrario, les bonnes pratiques de nommage sont mise en application dans le code ci-dessous :

```
impact.factor <- 1.45  # nommage explicite, en minuscule
nb.journal <- 5        # et choix du . pour séparer les mots

M <- matrix(0,2,2)     # choix des majuscules pour les matrices

euclidian.distance <- function(x, y) sqrt(sum((x - y) ^ 2))
                       # nom de fonction explicite
```


Avant tout : les tests

Le but des tests est de garantir que les modifications apportées à un code (depuis les premières lignes de code) sont vérifiées et n'altèrent pas l'exactitude des résultats. Il est donc plus que nécessaire de mettre en œuvre une organisation qui fera mentir l'adage suivant :

> « *Le problème, ce n'est pas qu'on n'a pas fait de tests, mais c'est qu'on ne les a pas gardés... et qu'on ne peut plus les refaire tourner automatiquement.* »
> (Wickham, 2014).

Il existe un éventail de tests qui suivent la terminologie suivante :

- *tests unitaires* : la règle numéro un, proposée dans les méthodes agiles, est d'écrire le test d'une fonctionnalité/d'un module... avant justement d'implémenter celle/celui-ci. Ce sont les tests unitaires. On notera par exemple que le projet R Bioconductor incite les développeurs à implémenter des tests unitaires et propose un ensemble de bonnes pratiques à leur destination (Bioconductor, 2016) ;
- *tests d'intégration* : il s'agit ici de vérifier l'assemblage des composants élémentaires pour réaliser un composant de plus haut niveau. Le concept d'intégration continue consiste à systématiser et automatiser ces tests dès lors que les composants élémentaires sont modifiés (eux-même testés avec les tests unitaires) ;
- *test de non-régression* : les tests de (non-)régression permettent de s'assurer que des défauts n'ont pas été introduits ou découverts dans des parties non modifiées, lorsqu'on ajoute une fonctionnalité ou qu'on réalise un changement dans une autre partie du programme. Ces tests sont fastidieux car ils nécessitent d'être le plus exhaustif possible.

Au niveau de R, une approche minimaliste peut consister à créer un répertoire (`devtests` par exemple), y installer des données et des scripts de tests remplis de `stopifnot()` comme ci-après et lancer régulièrement ces scripts avec `Rscript` :

```
log.likelihood <- compute.loglikelihood("data_test1.txt")
stopifnot(abs(log.likelihood - (-772.0688)) < 1e-8)
```

Ceci peut être adapté à la mise en œuvre de tests de régression (car peu nombreux) mais il s'agit cependant de « bricolage » et rien n'est automatique ici.
Pour gagner en robustesse, le développeur se tournera vers les packages **RUnit** et **testthat** qui sont les deux alternatives actuelles les plus communes pour aider à la réalisation de tests automatiques, en particulier les tests unitaires (associés à des modules, des classes, des fonctionnalités indépendantes). Nous évoquerons brièvement ici l'utilisation de **testthat** (Wickham, 2011). Avec **testthat**, voici la feuille de route, à implémenter dans le cadre de la structure d'un package R que nous appellerons **mypkg** :

1. Ajouter `testthat` dans le champ `Suggests` de `DESCRIPTION`.
2. Créer un répertoire `tests/testthat`
3. Dans ce répertoire, créer un ensemble de tests aux noms explicites de la forme `test-modulexxx.R` ou `test-fonctionnalitéxxx.R`. Ces tests contiennent des appels aux fonctions de vérifications `expect_equal, expect_identical, expect_match, expect_output`, etc.

   ```
   library(mypkg)
   context("Unit test for the model log-likelihood")
   test_that("Produces the correct log likelihood.", {
       expect_equal(object=compute.loglikelihood("data_test1.txt"),
       expected=-772.0688, tolerance = 1e-8)
   })
   ```

4. Créer un fichier `tests/testthat.R` qui va rassembler les tests lancés par `R CMD check` et qui contient les lignes suivantes :

   ```
   library(testthat)
   test_check("mypkg")
   ```

Pour terminer, on aura bien noté ici que, dans les deux cas `stopifnot()` ou `expect_equal`, l'égalité entre valeurs réelles n'est pas requise, les calculs étant réalisés en arithmétique flottantes (voir Annexe A.1). C'est pour cela que l'on compare ici les valeurs réelles avec un seuil (`tolerance`) à 1e-8.

La gestion de version

Le *gestionnaire de version* est l'outil le plus courant et sans doute le plus indispensable pour toute personne travaillant sur un document, que ce soit les sources d'un code, une publication... ou un livre ! Le principe est simple : il permet de conserver la trace chronologique de toutes les modifications qui ont été apportées aux

fichiers. En corollaire, il est ainsi relativement aisé de gérer les différentes versions d'un code et de créer des *branches de développement* : celles-ci sont des bifurcations dans la vie du code qui permettent le test de nouvelles fonctionnalités sans perturber les évolutions de la version principale par exemple. Le gestionnaire de version permet aussi facilement de développer à plusieurs en proposant la gestion des conflits et la fusion des modifications lorsque plusieurs personnes ont travaillé sur le même fichier.
Si les avantages d'un outil de gestion de version sont évidents pour un travail collaboratif, quel intérêt présente-il pour quelqu'un qui développe un « petit » code tout seul ? Une réponse pragmatique à cette question consiste à en poser une autre :

« Quel développeur n'a jamais effectué des modifications dans son programme qu'il a immédiatement regrettées, et pour lequel il a du réaliser des « undo » sans pouvoir revenir parfaitement à la situation de départ ? »

Il existe essentiellement deux types de systèmes de gestion de version : les systèmes *centralisés* comme `subversion` (`https://subversion.apache.org/`), et les systèmes *décentralisés* comme `git` (`https://git-scm.com/`). Dans le premier cas, il n'existe qu'un seul dépôt de l'ensemble des fichiers et des versions qui fait référence, ce qui en simplifie la gestion. Dans le second, plusieurs dépôts cohabitent, permettant plus de souplesse dans le travail individuel. On notera que les systèmes décentralisés prennent de plus en plus le pas sur les systèmes centralisés, en particulier du fait de la possibilité de faire des enregistrements fréquents de versions (des *commit*) localement sans les propager à tout le système.

Supposons que l'on souhaite la chronologie des modifications apportées au fichier source `estimate.dynsbm.R`. La commande `svn log` de l'outil `subversion` répond à nos besoins : elle retourne les *logs* (des messages), *i.e.* les informations écrites par les développeurs successifs qui ont modifié le code, la date ainsi que le volume des modifications. Ci-dessous, `svn log` nous donne les logs des versions 53 à 56 :

```
svn log
------------------------------------------------------------------------
r56 | louvet | 2016-01-05 12:48:10 +0100 (mar. 05 janv. 2016) | 1 ligne
Adding two additionnal parameters in estimate.dynsbm.

------------------------------------------------------------------------
r55 | vmiele | 2016-01-05 12:25:16 +0100 (mar. 05 janv. 2016) | 1 ligne
Setting present to NULL by default.

------------------------------------------------------------------------
r54 | vmiele | 2016-01-04 17:49:53 +0100 (lun. 04 janv. 2016) | 1 ligne
New stopping criteria in estimation procedure. Tests OK.
```

```
------------------------------------------------------------------------
r53 | louvet | 2016-01-04 16:52:10 +0100 (lun. 04 janv. 2016) | 1 ligne
Modification of unit test for estimation procedure
```

La commande `svn diff` permet de visualiser les différences entre deux versions sous une écriture standardisée :

- Entête : `---` désigne le premier fichier à comparer, ici la version 54 du code source `estimate.dynsbm.R` et `+++` le second. La suite encadrée par `@@` contient l'indication des lignes différentes dans chacun des fichiers. Dans la version 54, 9 lignes sont affectées, la première étant la ligne 13, et dans la version 56, 14 lignes sont modifiées, la première étant la ligne 13.
- Listes des lignes modifiées, avec le symbole `-` pour celle du premier fichier, et `+` pour celles du second.

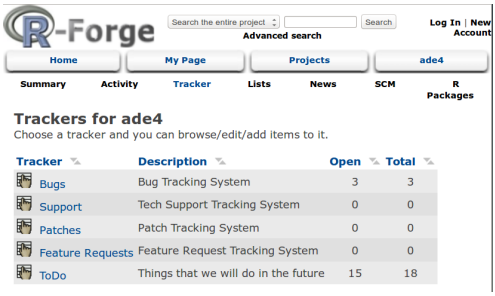
Dans le résultat du `svn diff` suivant, on retrouve bien les modifications indiquées par les développeurs dans les logs des différentes versions :

```
svn diff -r54:56

Index: estimate.dynsbm.R
===================================================
--- estimate.dynsbm.R (révision 54)
+++ estimate.dynsbm.R (révision 56)
@@ -13,9 +13,14 @@

-estimate.dynsbm <- function (Y, present=matrix(1L,N,T), Q) {
+estimate.dynsbm <- function (Y, present=NULL, Q,
+                         perturbation.rate=0., iter.max=20) {
```

Les forges, pour le développement collaboratif

Il existe des systèmes globaux appelés *forges logicielles* qui mettent à disposition plusieurs outils très utiles pour les développeurs de code. Tous ces outils sont disponibles au travers d'une interface web : gestionnaire de version, de documentation, listes de discussion, outil de suivi de bugs, gestionnaire de tâches...
Il existe de multiples instances de ces forges, en particulier `RForge` (`https://r-forge.r-project.org/`, pour la communauté des développeurs R), `SourceSup` (`https://sourcesup.cru.fr/`, plateforme d'hébergement de projets informatiques de l'Enseignement supérieur et de la Recherche française) et `GitHub` (`https://github.com/`, partage de code sur internet).
Le développeur R gagnera à utiliser une forge, en particulier dès lors qu'il s'agira d'un projet collaboratif où la question de la communication est primordiale. Pour

preuve, la plupart des grands projets (**Rcpp** par exemple) sont enregistrés sur une forge...



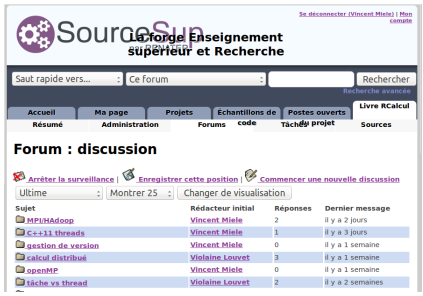


Fig. 1.2 – Captures d'écran de l'interface web des forges logicielles `RForge` (à gauche) et `SourceSup` (à droite).

1.2 Les différentes approches en R pour gagner en performances

1.2.1 Apprendre à mesurer les performances temps et mémoire

Donald Knuth a écrit :

> « *Les programmeurs gâchent énormément de temps en se focalisant sur des parties de leurs codes qui au final ne sont pas critiques !* » (Knuth, 1974).

A l'heure de l'optimisation, il est effectivement important de concentrer ses efforts sur les parties du codes qui sont réellement critiques, et pas sur celles que le programmeur pense critiques. Il devient donc indispensable d'adopter une méthodologie pour mesurer la (l'absence de) performance. La recette est simple et nécessite trois ingrédients :

1. un test et son jeu de données et/ou de paramètres ;
2. un idée de la complexité des algorithmes en fonction des données/paramètres (non abordé en détail dans ce livre, voir Annexe A.2) ;
3. des outils de mesure des performances en temps et en mémoire.

Nous proposons ici de présenter ces outils qui se focalisent tantôt sur le temps d'exécution, tantôt sur l'empreinte mémoire (en effet, les gains de performances peuvent venir d'une meilleure utilisation de la mémoire ; voir 2.2.2).

Remarque
Les exemples de cet ouvrage (sauf Chapitre 5) ont tous été testés sur un processeur quad-core Intel Xeon W3550 3.07 GHz avec 8 GB de mémoire, sous Linux avec `R 3.1.2`.

Monitorer le temps de calcul

L'approche la plus basique consiste à encadrer des sections de code par une commande `system.time` qui donne le temps écoulé (façon chronomètre, c'est celui que nous retiendrons dans la suite), le temps système (les opérations réalisées par le système d'exploitation Linux, Mac OSX ou Windows) et le temps utilisateur (pas utile ici) :

```
x <- runif(100)
system.time( for(i in 1:1e5){sqrt(x)} )
utilisateur     système      écoulé
0.128       0.000       0.126
```

Dans la suite, on ne retiendra que la troisième information :

```
system.time( for(i in 1:1e5){x^(1./2.)} )[3] # temps écoulé
elapsed
0.946
```

Cependant, la précision de `system.time` est faible et il convient de répéter une même opération plusieurs fois si celle-ci est très courte, comme dans l'exemple ci-dessus. Le package **microbenchmark** propose une alternative bien plus séduisante, en permettant d'une part la mesure du temps d'une ou plusieurs opérations avec une grande précision (nanosecondes), et en calculant d'autre part la distribution du temps de calcul par la réalisation d'un nombre approprié de réplicats (paramètres `times`, le plus grand possible tout en gardant un temps de calcul raisonnable) :

```
library(microbenchmark)
microbenchmark( sqrt(x), x^(1./2.), times = 1e5)
Unit: microseconds
expr    min  lq  mean median uq  max
sqrt(x) 1.10 1.16 1.22 1.19 1.22 3.94
x^(1/2) 9.18 9.34 9.59 9.43 9.52 26.08
```

L'exemple ci-dessous illustre parfaitement l'avantage de **microbenchmark** sur `system.time`, qui n'est pas assez précis :

```
x=9
system.time( sqrt(x) )[3]
elapsed
0
```

alors que **microbenchmark** permet d'étudier des opérations de calcul avec une précision de l'ordre de la microseconde :

```
microbenchmark(
 sqrt(x),
 x^0.5,
 x^(1./2.),
 exp(log(x)/2)
)
Unit: microseconds
expr    min  lq  mean median uq  max
sqrt(x) 1.10 1.15 1.22 1.18 1.21 4.43
x^0.5 8.98 9.05 9.22 9.13 9.18 17.82
x^(1/2) 9.19 9.28 9.40 9.35 9.47 10.15
exp(log(x)/2) 8.34 8.50 8.88 8.58 8.70 26.02
```

Néanmoins, l'utilisation de ces outils de mesure ne permet pas une analyse exhaustive et sans *a priori* du temps d'exécution de chaque partie du code. Le développeur doit désormais réaliser le *profilage* (ou *profiling*) de son code pour rechercher les points chauds ou goulots d'étranglements (ou *bottleneck*). Ceux-ci trouvés, l'étape suivante sera la ré-écriture en privilégiant les axes présentés dans la suite (voir 1.2.2), puis de nouveau une étape de profilage... on itère ainsi jusqu'à épuisement !

Réaliser le profilage du code

L'expérience montre que quelques lignes de code sont souvent responsables d'un grand pourcentage du temps de calcul global. Trouver ces lignes, c'est la mission de l'outil de profilage, le *profiler*, en particulier `Rprof`. `Rprof` procède par échantillonnage : il stoppe l'exécution du code par intervalles de quelques millisecondes (intervalle à définir avec le paramètre `interval`) et écrit dans un fichier quelle fonction est en train de s'exécuter. Ce fichier servira à réaliser une synthèse globale grâce à la fonction `summaryRprof`. `Rprof` propose deux informations distinctes et complémentaires :

1. le temps de calcul réalisé par chaque fonction par elle-même *i.e.* en excluant le temps de calcul des fonctions qu'elle appelle (`self.time` ou `self.pct` pour le pourcentage rapporté au temps total) ;
2. le temps de calcul global réalisé entre le première et la dernière ligne de chaque fonction (`total.time` ou `total.pct`) ; ce temps englobe les temps de calcul des fonctions appelées.

Supposons que l'on souhaite réaliser le profilage de la fonction `faire.courses` qui appelle les fonctions `boulangerie` et `poissonnerie` qui elle-même contiennent un appel à `faire.la.queue`, `commander` et `payer` :

```
faire.courses <- fun(){
```

```
 boulangerie()
 poissonnerie()
}
boulangerie <- fun(){
 faire.la.queue()
 commander()
 payer()
}
poissonnerie <- fun(){
 faire.la.queue()
 commander()
 payer()
}
```

Si le `total.time` de `boulangerie` est 5 minutes, son `self.time` sera petit car ce sont les fonctions qu'elle appelle qui prennent du temps ; *a contrario*, le `self.time` de `faire.la.queue` sera le plus élevé car cette fonction apparait dans `boulangerie` et `poissonnerie`. Les résultats (fictifs) de `Rprof` seront donc les suivants :

```
$by.self
                 self.time
faire.la.queue   6
commander        2
payer            2

$by.total
                total.time
faire.courses   10
faire.la.queue   6
boulangerie      5
poissonnerie     5
commander        2
payer            2
```

Prenons un exemple qui consiste à lister les nombres premiers inférieurs à un nombre n. On notera l'utilisation du nom de fichier `Rprof.out` que l'on peut changer afin de garder les résultats du profilage de différentes configurations dans différents fichiers :

```
is.prime <- function(n) n == 2L ||
                        all(n %% 2L:ceiling(sqrt(n)) != 0)
all.prime <- function(n){
        v <- integer(0)
```

```
        for(i in 2:n) if(is.prime(i)) v <- c(v,i)
        v
}

Rprof("Rprof.out", interval = 0.001)
prime.numbers <- all.prime(1e5)
Rprof(NULL)
summaryRprof("Rprof.out")

$by.self
self.time self.pct total.time total.pct
"!="            0.087    38.67      0.087     38.67
"c"             0.039    17.33      0.039     17.33
"%%"            0.030    13.33      0.030     13.33
"all"           0.021     9.33      0.021      9.33
":"             0.019     8.44      0.019      8.44
"is.prime"      0.018     8.00      0.177     78.67
"all.prime"     0.009     4.00      0.225    100.00
"=="            0.002     0.89      0.002      0.89

$by.total
total.time total.pct self.time self.pct
"all.prime"      0.225    100.00     0.009     4.00
"is.prime"       0.177     78.67     0.018     8.00
"!="             0.087     38.67     0.087    38.67
"c"              0.039     17.33     0.039    17.33
"%%"             0.030     13.33     0.030    13.33
"all"            0.021      9.33     0.021     9.33
":"              0.019      8.44     0.019     8.44
"=="             0.002      0.89     0.002     0.89
```


Dans cet exemple, on peut être surpris de constater que l'opérateur `c()` prend 17.33 % du temps total : cette observation inattendue, révélée par le profilage, pourra conduire à une ré-écriture de cette fonction. En effet, on gagnera à préallouer `v` (détails sur la préallocation en 1.2.2) de taille $1.2 * n/log(n)$ (borne supérieure du nombre de nombres premiers $< n$) et l'opérateur `c()` disparaitra du profilage.

Au-delà de cet ouvrage, le lecteur pourra s'orienter vers différents packages, en particulier **lineprof** et **proftools** qui proposent des outils modernes de post-traitement des sorties du profilage.

La mémoire utilisée

L'utilisation de la mémoire par R est pour beaucoup mystérieuse, et il convient d'adopter une démarche pragmatique : comprendre, c'est bien, tester et monito-

rer, c'est bien aussi. Dans cet esprit, l'incontournable package **pryr** propose un ensemble de fonctions de monitoring de l'empreinte mémoire des objets R et permet de souligner l'existence de copies inutiles. Le programmeur se familiarisera avec `object_size(), mem_used()` et `mem_change()` qui donnent respectivement la taille mémoire d'un objet, l'occupation mémoire du processus R et les changements en mémoire effectués. Dans l'exemple suivant, on verra l'impact en mémoire de l'utilisation d'entiers (4 octets) ou de réels (8 octets) :

```
library(pryr)
u <- rep(1L,1e8)  # on précise bien que u contient des entiers 1L
object_size(u)
400 MB

v <- rep(1,1e8)  # v contient des réels car 1 est le réel 1.0
object_size(v)   # la taille de v est le double de celle de u
800 MB
```

pryr met également à disposition `tracemem()`, la fonction de surveillance des changements de place en mémoire d'un objet (sous entendue une copie implicite). Enfin, les fonctions plus techniques `adress()` et `refs()` indiquent respectivement l'adresse mémoire d'un objet et le nombre d'objets qui pointent sur une même adresse (*i.e.* des objets synonymes en quelque sorte). L'exemple suivant montre que la mécanique interne de R est transparente pour le programmeur mais occasionne des opérations en mémoire insoupçonnées :

```
v=1:1e8
address(v)
[1] "0x7f1a12c5d010"
refs(v)
[1] 1
mem_used()
422 MB

w = v
mem_used() # pas de copie inutile de w dans v !
422 MB

c(address(v), address(w))
[1] "0x7f1a12c5d010" "0x7f1a12c5d010"
refs(v)    # v et w ont la même adresse mémoire
[1] 2

v[1] = 0L  # w et v deviennent différents car v est modifié
mem_used()
822 MB
```

```
c(address(v), address(w)) # v et w n'ont plus la même adresse
[1] "0x7f19faee4010" "0x7f1a12c5d010"
```

On notera enfin qu'il est également possible de réaliser un profilage mémoire avec `Rprof` en activant l'option `memory.profiling=TRUE` qui permettra l'ajout d'une colonne `mem.total`.

Remarque
La commande `top`, présenté en 2.2.1, sera également un outil indispensable pour surveiller dynamiquement la consommation mémoire de R.

1.2.2 Améliorer/optimiser le code R

Il existe des nombreuses possibilités (mais pas des règles absolues) pour améliorer l'efficacité d'un code R en recodant différemment en R (*refactoring*) des sections de programme identifiées au préalable (comme indiqué en 1.2.1). Plusieurs articles et ouvrages en font état (Wickham, 2014; Matloff, 2015; Lim & Tjhi, 2015; Visser *et al.*, 2015) et nous énumérons ici quelques pistes incontournables.

Vectoriser. Vectoriser son code consiste dans un premier temps à éviter les boucles car le programmeur peut facilement mal les implémenter sans s'en rendre compte. Ceci peut être réalisé en particulier grâce aux fonctions `*apply` (*i.e.* `apply`, `lapply`, `sapply` et `vapply`) mais attention celles-ci ne garantissent absolument pas des performances optimales. Par contre, l'utilisation des fonctions *vectorisées*, *i.e.* qui agissent sur un vecteur et qui sont écrites en `C`, doit être une priorité. Parmi elles, nous pouvons citer `rowSums, colSums, rowMeans, colMeans, max.col, cumsum, diff, pmin` et `pmax`.

Mettons-nous dans la peau d'un programmeur débutant en R. Supposons que l'on ait besoin d'une fonction qui fait la somme des éléments de chaque ligne d'une matrice. Ni une ni deux, le programmeur peut mettre en application ce qu'il a appris en cours de programmation généraliste, à savoir faire des boucles :

```
myRowSums <- function(mat){
 sums <- rep(0,nrow(mat))
 for (i in 1:nrow(mat))
  for (j in 1:ncol(mat))
   sums[i] <- sums[i] + mat[i,j]
 sums
}
```

Puis il apprend qu'il est recommandé d'utiliser les fonctions `*apply`... alors une version possible du calcul de la somme sera :

```
apply(mat,1,sum)
```

Pour finir, le programmeur découvre la fonction `rowSums` qui est vectorisée. Laquelle des trois solutions doit-on choisir ? **microbenchmark** nous donne la réponse :

```
mat = matrix(rnorm(1e3*1e3),1e3,1e3)
microbenchmark(
 myRowSums(mat),
 apply(mat,1,sum),
 rowSums(mat),
 times=10
)
Unit: milliseconds
expr              min     lq      mean    median  uq      max
myRowSums(mat)    1438.03 1441.45 1455.47 1447.32 1474.66 1477.36
apply(mat,1,sum) 22.36    22.59   22.86   22.91   23.21   23.26
rowSums(mat)      2.57    2.58    2.60    2.60    2.61    2.61
```

Sur cet exemple naïf, on observe un écart de performance béant de 1 à 550.

Préallouer la mémoire . Dans R, les objets sont alloués dynamiquement, c'est-à-dire que la mémoire nécessaire aux objets leur est associée au fil de l'exécution du programme. Dès lors, à chaque fois qu'un vecteur change de taille (par un ajout d'élément(s) par exemple), R copie son contenu vers un nouveau vecteur sur un nouvel espace mémoire plus grand et détruit l'ancien vecteur. On le verra en 2.2.2, les opérations de création et de destruction d'objets en mémoire demandent du temps. Pour gagner en performance, il devient alors primordial de préallouer les objets, à partir de la connaissance *a priori* de la taille finale de ceux-ci, comme ci-dessous dans la fonction `withprealloc` :

```
noprealloc <- function(n){
 result <- c()
 for (i in 1:n) result <- c(result,rnorm(1))
  result
}
withprealloc <- function(n){
 result <- double(n)  # préallocation de la mémoire
 for (i in 1:n) result[i] <- rnorm(1)
  result
}

microbenchmark(
 noprealloc(1e4),
 withprealloc(1e4),
```

```
 times=10
)
Unit: milliseconds
expr                min    lq     mean   median uq     max
noprealloc(10000)   150.40 152.07 155.61 152.27 153.27 185.73
withprealloc(10000) 34.52  34.85  35.62  35.79  36.223 36.80
```

On notera d'ailleurs que les fonctions `*apply` tirent profit de la préallocation pour gagner en performances. Ainsi, `vapply` peut s'avérer plus rapide que `sapply` car il faut préciser le type exact du retour (la préallocation aura lieu dans la foulée).

Éviter les duplications en mémoire et les objets temporaires . Pour commencer, il vaut mieux « appeler un chat un chat » : un entier doit être suffixé de la lettre `L` sinon il sera considéré réel. Dans l'exemple suivant, il y aura des opérations en mémoire et une occupation mémoire inutile du fait de la conversion non souhaitée d'entier vers réel (4 octets *versus* 8) :

```
v <- 1:1e8 # vecteur d'entier
object_size(v)
400 MB

tracemem(v)
[1] "<0x7fbc6fc8d010>"
v[1]<-0 # au lieu de 0L => conversion en réel signalée par tracemem
tracemem[0x7fbc6fc8d010 -> 0x7fbc4019c010]:

object_size(v)
800 MB
```

Plus généralement, R peut réaliser des copies d'objets fréquentes, sans que cela soit visible. En conséquence, il est souhaitable de tester régulièrement l'occupation mémoire avec les outils présentés en 1.2.1, et de vérifier s'il n'y a pas des copies cachées. Enfin, pour éviter l'inflation mémoire, le programmeur pensera à détruire les objets temporaires qui ne sont plus nécessaires avec la fonction `rm()`.

Travailler moins pour gagner plus . Pour une fois, on gagnera à être paresseux... En effet, avant d'écrire des lignes de codes, il faut se demander si quelqu'un n'a pas déjà implémenté ce dont on a besoin, pour ne pas avoir à le faire soi-même. Pécher de paresse ? Non, plutôt confiance dans la capacité de la communauté à faire émerger des implémentations efficaces des algorithmes les plus courants. Se baser sur le travail de la communauté, c'est aussi ça la force de l'*open source*. Des solutions avantageuses sont présentes par exemple dans les packages **plyr**, **dplyr** et **data.table** pour travailler sur les **data.frame** (voir exemple suivant), dans le package **fastcluster** pour améliorer la vitesse de `hclust`, **RcppEigen** pour l'algèbre

linéaire (et une alternative rapide à `lm`), **xts** pour les séries temporelles, etc.

Considérons l'objet `dtf` qui contient le poids et le groupe (catégorie sociale, âge, ...) de 100 000 individus :

```
dtf <- data.frame(poids=rchisq(100000,10),
                  groupe=factor(sample(1:10,100000,rep=T)))
```

Une analyse de données naturelle consistera à calculer la moyenne et l'écart type du poids par groupe. Comment implémenter cela efficacement ?
La première idée consistera à faire une boucle sur les groupes pour extraire un *subset* par groupe :

```
# préallocation du résultat
r <- data.frame(poids=1:10, mean=rep(0,10), sd=rep(0,10))

microbenchmark(
for(groupe in 1:10){
 tmp <- dtf$poids[dtf$groupe==groupe]
 r$mean[groupe] <- mean(tmp)
 r$sd[groupe] <- sd(tmp)
}, times=10)
Unit: milliseconds
min   lq    mean   median  uq   max
87.81 88.72 103.83 89.15 126.81 127.24
```

La fonction `aggregate` est également à envisager (sa syntaxe est particulièrement lisible et explicite), néanmoins ses performances sont ici catastrophiques :

```
microbenchmark(
 aggregate(poids ~ groupe, data=dtf,
           FUN = function(x) c(mean=mean(x), sd=sd(x))),
 times=10
)
Unit: milliseconds
min    lq     mean   median uq     max
478.76 516.81 519.01 518.55 520.21 561.13
```

Le package **plyr** est reconnu pour ses possibilités de fractionnement de données en sous-unités qui peuvent ensuite être traitées. Il convient donc parfaitement à notre problème et il améliore le temps de calcul :

```
library(plyr)
microbenchmark(
```

```
 ddply(dtf, ~groupe, summarise, mean=mean(poids),sd=sd(poids)),
 times=10
)
Unit: milliseconds
min   lq    mean  median  uq  max
20.62 20.75 21.94 21.01 21.66 29.64
```

L'arrivée plus récente du package **dplyr** a apporté des fonctionnalités proches de celles de **plyr** avec un focus sur l'efficacité. Son utilisation dans notre contexte est édifiante, avec un net gain de performance dû à son utilisation :

```
library(dplyr)
microbenchmark(
 dtf %>% group_by(groupe) %>% summarise(mean(poids),sd(poids)),
 times=10
)
Unit: milliseconds
min  lq   mean median uq max
4.75 4.77 4.91 4.79 4.81 6.01
```

Pour finir, il semble inévitable d'utiliser le package **data.table** qui propose une autre implémentation des `data.frame` et qui permet des manipulations de données d'une extrême efficacité. Sur notre exemple, il conduit à une performance optimale :

```
library(data.table)
dt <- data.table(dtf)
microbenchmark(
 dt[,list(mean=mean(poids),sd=sd(poids)),by=groupe],
 times=10
)
Unit: milliseconds
min  lq   mean median uq max
3.18 3.23 3.69 3.24 3.27 7.78
```

De 500 à 3 millisecondes, nos différentes implémentations n'ont pas la même efficacité...

Changer d'algorithme et/ou de structure de données . Pour gagner en performance temps ou mémoire, il peut s'avérer utile de... réfléchir. Est-ce que ma structure de données est optimale ? Non ? Une étape primordiale consistera à analyser la complexité mémoire ou temps (voir rappels en Annexe A.2) de l'algorithme d'une fonction ou d'une structure de données pour éventuellement trouver

une alternative de meilleure complexité. On passera par exemple de `Matrix` à `sparseMatrix` pour réduire l'empreinte mémoire des grandes matrices creuses.

Supposons que l'on a des identifiants d'individus dans le vecteur `all.ids` et que l'on doit vérifier si l'individu d'identifiant `id` est présent dans ce vecteur. On réalise alors une recherche linéaire (*i.e.* un parcours élément par élément du vecteur) et la complexité en temps est linéaire. Pour cela, voici deux implémentations possibles :

```
all.ids <- sample(1e9, 1e7) # identifiants tirés au hasard
id <- sample(1e9,1)
microbenchmark(
 id %in% all.ids,
 any(all.ids==id),
 times=25
)
Unit: milliseconds
expr            min    lq     mean   median uq     max
id %in% all.ids 503.22 515.12 546.48 538.70 578.84 632.45
any(all.ids==id) 54.62  59.79  67.94  73.20  73.94  76.52
```

Si il nécessaire de réaliser de nombreuses recherches dans ce vecteur, alors on préférera le trier puis réaliser une recherche dichotomique (*binary search*) dans ce vecteur trié : en effet, la recherche dichotomique est de complexité en temps logarithmique. Cette stratégie est implémentée par la suite grâce à la fonction `binsearch` :

```
system.time( all.ids.sorted <- sort(all.ids) )[3]
1.711
library(gtools)
microbenchmark(
 binsearch(fun=function(i) all.ids.sorted[i]-id,
           range=c(1,length(all.ids.sorted)))
)
Unit: microseconds
min    lq     mean   median uq     max
157.04 159.75 162.19 161.05 162.63 205.48
```

Le gain de temps pour la recherche est conséquent. Le tri a certes un coût, mais celui-ci sera négligeable si on fait de nombreuses recherches par la suite.

Eviter la saturation de la mémoire . Dans le contexte actuelle des données massives, le programmeur peut être amené à manipuler des données dont la taille

dépasse la capacité maximale de la mémoire de sa machine.
Une première alternative visera à éviter de charger l'intégralité d'un fichier de données dans un objet R. En effet, s'il est possible de réaliser le traitement voulu sur les lignes de données au fur et à mesure, alors la lecture par bloc de lignes avec `readLines` permettra de réduire l'empreinte mémoire nécessaire.
Une autre solution pourra consister à s'appuyer sur le package **big.memory** (le package **ff** existe également) et son principe des fichiers *memory-mapped*. L'objet manipulé est une *big.matrix* qui est stockée partiellement en mémoire et partiellement sur le disque ; le chargement depuis la mémoire vers le disque ou l'inverse sera réalisé en fonction de l'utilisation de telle ou telle section de matrice. Nous verrons en 2.2.2 que les accès au disque introduisent une forte lenteur et l'utilisation de *big.matrix* sera intrinsèquement moins performante en temps.

Compiler certaines fonctions R . Il peut être intéressant de « compiler » une fonction avant son utilisation grâce au package **compiler** : celui-ci effectuera une traduction sous forme de « bytecode » de la fonction. Le bytecode sera lui-même traduit en code machine au fur et à mesure de l'exécution avec des gains de performance dans certaines conditions sans risque de perte. Cette approche n'est pas adaptée dès lors que cette fonction fait appel à d'autres fonctions, *a fortiori* si elles sont déjà optimisées (`rowSums` par exemple). Nous mentionnons donc brièvement cette approche en l'illustrant avec l'exemple basique suivant :

```
library(compiler)
f <- function(n){ x <- 1; for (i in 1:n) x <- (1 + x)^(-1); x}
g <- cmpfun(f)

microbenchmark(f(1000), g(1000))
Unit: microseconds
expr    min    lq     mean   media   uq    max
f(1000) 530.33 540.89 572.17 553.39 578.98 1352.23
g(1000) 158.10 161.56 185.58 163.10 183.55  989.72
```

1.2.3 Implémenter les points chauds de calcul avec des langages compilés

Si on réalise par exemple le profilage de la fonction `princomp` (analyse en composantes principales), on note la présence de lignes qui mentionnent les fonctions `.C` et `.Call`... ces fonctions sont tout simplement des fonctions d'interfaçage entre R et `C/C++`. De quoi s'agit-il ?

La grande tendance en développement moderne de logiciel de calcul consiste à interfacer les langages pour prendre le meilleur de chacun, *i.e* programmer des blocs de codes dans différents langages et permettre la transmission des données entre ces blocs. Cette approche permet de tirer profit des forces de chaque langage

sans en subir les faiblesses (voir Annexe A.3 pour une typologie des langages). Par exemple, R (au contraire de C++) offre de nombreuses possibilités pour construire facilement des graphiques, alors que C++ (au contraire de R) garantit de bonnes performances calculatoires : on associera donc R et C++ en les interfaçant. D'ailleurs, de nombreuses fonctions R ne sont en fait que des appels à du C.
Le schéma classique d'un code de calcul (et de beaucoup de packages R) consiste à implémenter :

1. la lecture des données et le pré-traitement dans un langage haut niveau interprété (voir Annexe A.3) ;
2. les étapes de calcul dans un langage de niveau intermédiaire compilé ;
3. le post-traitement, la visualisation et/ou l'écriture des données de nouveau dans un langage de haut niveau interprété.

Dans le document de référence « Writing R extensions » (RCoreTeam, 2016), une section dédiée à l'interfaçage entre R et les langages compilés propose une approche parfois délicate à mettre en oeuvre (en particulier la gestion en mémoire des objets partagés entre langages). Toutefois, D. Eddelbuettel et R. François proposent depuis 2009 le package **Rcpp** qui encapsule cette approche dans un ensemble de fonctionnalités conviviales pour le programmeur dans le but d'interfacer R et C++ (voir Eddelbuettel (2013) pour plus de détails). Dans R, il devient donc particulièrement recommandé de coder ou recoder (du *refactoring*, de nouveau) les points chauds de calcul en C++ avec **Rcpp**, C++ étant par ailleurs un langage moderne, efficace et orienté objet. Pour cela, il y a deux solutions :

1. lors de la réalisation d'une package, on ajoutera alors à la racine du projet un répertoire `src` qui contient les fichiers C++ et on suivra les étapes nécessaire à l'utilisation de **Rcpp** (Eddelbuettel, 2013) ;
2. pour une utilisation sporadique de C++, par exemple lors de l'écriture de petits scripts ou lorsque l'on interagit avec l'invite de commande R, il est possible d'utiliser indirectement **Rcpp** au travers du package **inline** (Eddelbuettel, 2013). En particulier, la fonction `cppfunction`, qui prend comme argument une section de code C++, permet de déclarer une fonction utilisable dans R mais écrite en C++ (voir des exemples d'utilisation de `cppfunction` en Annexe A.5).

Dans la suite du livre, nous nous appuierons sur la seconde solution basée sur **inline**.

Les gains de performances à attendre peuvent être conséquents. Tout dépend cependant de la qualité du code R qui est réécrit en C++. Si celui-ci ne fait appel qu'à des fonctions R qui cachent elles-même du langage compilé, alors les gains seront minimes. Néanmoins, pour des algorithmes de calcul complexes, il n'est pas rare d'observer des gains $\times 5$ à $\times 100$ et l'effort fourni est toujours récompensé.

Y aurait-il moyen d'optimiser la fonction `sum` de R ? Il est possible de proposer une implémentation en pur R :

```
sumR <- function(x){
 n <- length(x)
 total <- 0
 for(i in 1:n) total <- total+x[i]
 return(total)
}
```

Mieux, on proposera une implémentation C++ interfacé avec R grâce à **Rcpp** et **inline** :

```
library(Rcpp)
library(inline)
cppFunction('
double sumC(NumericVector x) {
 int n = x.size();
 double total = 0;
 for(int i = 0; i < n; ++i) total += x[i];
 return total;
}
')
```

On remarque ci-après que la version **Rcpp** est aussi rapide que la fonction R `sum` qui, en fait, n'est autre qu'un appel à une fonction écrite en C :

```
v = runif(1e6)
microbenchmark(
 sumR(v),
 sum(v),
 sumC(v),
 times=25
)
Unit: microseconds
expr    min    lq     mean    median   uq     max
sumR(v) 343553 350884 356831  352346  355944  390618
sum(v)  987.43 1031.8 1139.5  1111.2  1227.94 1377.52
sumC(v) 957.31 1025.5 1111.94 1074.17 1177.88 1305.29
```

Bien-sûr, aucun développeur R avisé n'aurait proposé la fonction `sumR`, néanmoins l'écart de performance stratosphérique entre `sumR` et `sumC`, certes caricatural, peut éveiller l'esprit du programmeur R...

Remarque
Le problème des comparaisons de performances entre langage est qu'il est rare de trouver un expert qui soit expert de tous les langages comparés. Les versions comparées peuvent être variablement optimales du fait du degré d'expertise variable du programmeur. Il apparaît donc sain de se méfier des comparatifs, en particulier ceux de la blogosphère :

« Si l'on souhaite savoir qui du vélo et de la planche à voile est le plus rapide, on ne demande pas à un champion cycliste de faire un kilomètre de vélo puis un kilomètre de planche... » (les auteurs).

1.2.4 Utiliser plusieurs unités de calcul

Après avoir épuisé les recettes des différentes approches précédemment abordées, le programmeur peut avantageusement se tourner vers l'idée de combiner la puissance de plusieurs unités de calcul pour gagner en performance. Nous verrons dans le chapitre 2 que ceci est en parfaite adéquation avec l'écosystème matériel qui fait apparaître la multiplicité des unités de calcul à différents niveaux. La question du *parallélisme* devient centrale, cette idée qui suggère qu'on n'est jamais aussi rapide qu'à plusieurs ! Dans la vraie vie, pour repeindre une grande pièce, on gagne à se mettre à plusieurs... sous réserve de s'être organisé, synchronisé et d'avoir communiqué. Dans R ou dans le C++ interfacé avec R, ces mêmes principes gouvernent et un ensemble de techniques basées sur différents packages est à la disposition du programmeur « éclairé », éclairé par les chapitres 3 à 5.

Chapitre 2

Fondamentaux du calcul parallèle

2.1 Évolution des ordinateurs et nécessité du calcul parallèle

Imaginons Monsieur R. User qui débute sa carrière au début des années 2000. Il a à sa disposition une certaine quantité de données dont l'analyse lui permet de comprendre certains mécanismes de son domaine de recherche. Il réalise cette analyse en développant un code R lui permettant d'utiliser les packages réalisés par d'autres collègues.
Au fil des années, les technologies évoluent et lui permettent d'accéder à une quantité de données de plus en plus importante. En parallèle, l'avancée de ses recherches le conduisent à complexifier sa modélisation.
Le code développé s'enrichit au fur et à mesure, et les exécutions sont de plus en plus longues. Qu'à cela ne tienne, notre chercheur investit dans un nouvel ordinateur dont l'utilisation accélère de façon importante la rapidité de son code.
Le processus suit son cours, nécessitant deux ans plus tard l'achat d'une nouvelle machine, afin que le code continue de tourner dans des délais raisonnables.
Les technologies d'acquisition de données s'améliorent, permettant à notre chercheur de bénéficier de données toujours plus massives et complexes. Le code s'enrichit également et les temps de calcul sont de plus en plus longs. Monsieur R. User se dit tout naturellement que l'investissement dans un nouveau serveur de calcul va régler le problème comme cela a été le cas au cours des années précédentes. Il acquière donc une machine haut de gamme

aux performances largement supérieures à son ancien serveur. Et réalise les calculs dont il a impérativement besoin pour finaliser son article en cours. Sa déception est alors grande lorsqu'il constate que le code tourne sur cette nouvelle machine légèrement plus lentement que sur son ancien serveur ! Pourtant la performance affichée pour ce nouveau matériel est largement supérieure !
Monsieur R. User est démuni devant cette situation, se demandant comment il va bien pouvoir traiter la masse de données qui sortira de la toute nouvelle machine de production haut débit qui vient d'être installée sur son lieu de travail...

De façon simplifiée, jusque vers les années 2005, la performance d'une machine était directement lié au nombre d'opérations qu'elle était capable de réaliser en une seconde, c'est-à-dire à la *fréquence* de son unité de calcul (le *processeur*). Et chaque nouvelle génération de machine voyait la fréquence de son processeur augmenter régulièrement (voir Figure 2.1).

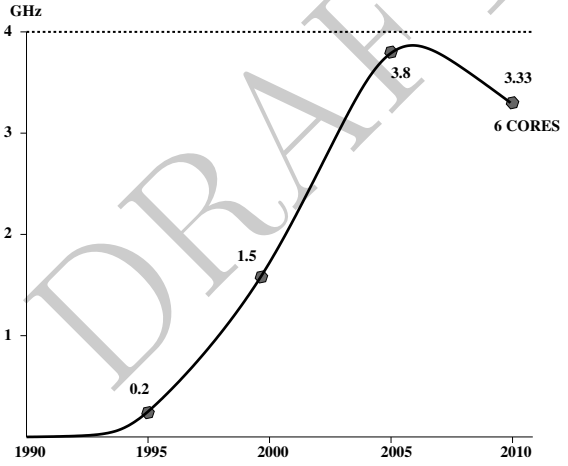


Fig. 2.1 – « *The free lunch is over* » : on ne peut plus se contenter de se reposer sur l'augmentation de la fréquence du processeur pour accroître les performances des codes de calcul.

L'augmentation de la fréquence d'un processeur n'est pas anodine : elle entraîne une augmentation de la quantité de chaleur à dissiper. Et il est vite devenu économiquement non rentable pour les fondeurs (constructeurs de processeurs) de refroidir des processeurs à fréquence élevé. Les fréquences se sont donc élevées jusqu'aux alentours de 4 GHz pour finalement se stabiliser entre 3 et 4 GHz.
En parallèle, les machines ont subi une petite révolution : les fondeurs ont réussi

à graver leurs transistors de plus en plus finement, et donc à pouvoir mettre sur le même bout de silicium un nombre de transistors de plus en plus élevé... ce qui a fait, entre autres, qu'au lieu de mettre une seule unité de calcul dans le processeur, on a pu en mettre 2 puis 4, 6 ou 8. Donc potentiellement, les machines devenaient alors 2 ou 4 fois plus performantes. Potentiellement seulement car le code qui tournait sur une machine avec un seul processeur doté d'une seule unité de calcul ne pouvait pas, tel quel, utiliser les 2 ou 4 unités de calcul. De plus, pour des questions de dissipation thermique, les fondeurs ont légèrement baissé la fréquence des processeurs pour pouvoir accueillir un plus grand nombre d'unités de calcul...

> Mais alors, comment la nouvelle machine de Monsieur R. User peut-elle être plus performante ? Parce que dotée de plusieurs unités de calcul !
> Et pourquoi le code de Monsieur R. User a fini par aller quasiment moins vite d'une génération de machine à l'autre ? Parce que la fréquence des unités des calcul a baissé pour pouvoir en assembler plusieurs sur un même processeur !
> Monsieur R. User doit désormais « penser parallèle » pour s'en sortir.

Pour conserver une performance proportionnelle au potentialité de la machine, il faut pouvoir exploiter toutes les unités de calcul en même temps. C'est le sens de l'expression « *The free lunch is over* » : on ne peut plus gagner en performance de façon gratuite, juste en recompilant le code... Il faut « penser parallèle » !

2.2 Les architectures parallèles

On vient de parler de processeurs et d'unités de calcul. Au siècle dernier, les deux notion étaient synonymes... ce n'est plus si simple ! Désormais, plusieurs termes reviennent régulièrement dans le monde des architectures matérielles : *CPU, processeur, cœur, nœud, cluster, thread, socket, processus*... Il faut distinguer ici les éléments purement matériels de ceux liés aux systèmes d'exploitation et à la programmation. Si on caricature beaucoup les choses, un ordinateur est constitué d'une carte mère sur laquelle on peut plugger :

- un ou plusieurs processeurs,
- de la mémoire,
- une carte réseau,
- une carte graphique...

mais il est désormais utile aux programmeurs R d'en connaître un peu plus !

Remarque
Nous utilisons ici l'expression unité de calcul pour désigner l'objet global qui permet de réaliser l'exécution des programmes sur un ordinateur. Il ne faut pas les

confondre avec les unités arithmétiques et logiques ou les unités de calcul en virgule flottante qui sont des éléments internes au processeur et dont nous ne parlerons pas dans cet ouvrage.

2.2.1 Éléments de vocabulaire autour du processeur

Processeur vs cœur vs nœud

Le *processeur* (ou CPU pour *Central Processing Unit*) désigne de façon globale l'ensemble des composants en charge de l'exécution des instructions machines, et donc des programmes. Les processeurs sont connectés sur la carte mère via des réceptacles d'accueil appelés *socket*.

Jusqu'au début des années 2000, le processeur ne contenait qu'une seule unité de calcul (au sens de la remarque précédente 1). Depuis cette période, les processeurs sont dit *multi-cœurs* c'est-à-dire qu'ils intègrent plusieurs unités de calcul, les *cœurs*, gravés au sein de la même puce. Les serveurs de calcul actuels sont en général *multi-processeurs* car ils contiennent plusieurs processeurs multi-cœurs. Le terme *multi-sockets* est également employé pour désigner cette configuration.

Remarque
Attention, le terme *socket* a différentes significations. On parle ici d'un réceptacle matériel. Ce terme est également utilisé, comme nous le verrons par la suite au chapitre 3, dans le cadre de la programmation, pour désigner une interface logicielle avec les connecteurs réseau du système d'exploitation.

Processus vs. thread

Les précédentes notions concernent les caractéristiques matérielles. Lorsqu'on exécute un programme sur un serveur, le système d'exploitation joue un rôle primordial : c'est lui qui est chargé du lancement de l'exécution sous la forme d'un *processus* auquel il associe un environnement mémoire propre. Un processus correspond donc à l'exécution d'un programme.

Les processus ont un identifiant unique, le *PID* (*Process IDentifier*). Ils peuvent avoir la capacité, si le programme associé au processus a été écrit en ce sens, de pouvoir lancer des sous-processus appelées *processus légers* ou *threads* qui partageront en grande partie les ressources du processus maître, et notamment la mémoire. Le système d'exploitation est en charge de la gestion complète des processus : création, destruction, ordonnancement, utilisation des ressources.

Remarque
A noter que la notion de *thread* est également présente au niveau de la programmation, ce que nous verrons lors du chapitre 4.

Le meilleur ami de Monsieur R. User, avide de performances, sera sans nul doute l'outil `top` (disponible sous Linux et Mac OSX, les utilisateurs de Windows trouveront un équivalent comme `Process Explorer`). Celui-ci permet de visualiser l'état dynamique du système et de tous les processus actifs à un moment donné, comme dans la figure suivante :

```
Tasks: 197 total,   4 running, 193 sleeping,   0 stopped,   0 zombie
%Cpu(s): 46,3 us,  0,7 sy,  0,0 ni, 53,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
KiB Mem:   8157592 total,  4406156 used,  3751436 free,   557300 buffers
KiB Swap:  3905532 total,   396464 used,  3509068 free.  1500356 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
23891 vmiele    20   0  115760  35772   6952 R  92,6  0,4   0:25.88 R
23882 vmiele    20   0  115648  36004   6960 R  91,6  0,4   0:25.73 R
  840 root      20   0  973008 174528  95760 S   2,0  2,1  76:22.53 Xorg
20369 vmiele    20   0 1446944 481092  94536 S   2,0  5,9  13:54.78 firefox
23343 root      20   0       0      0      0 S   1,0  0,0   0:01.80 kworker/0:2
    1 root      20   0  182480   4156   2864 S   0,0  0,1   0:06.62 systemd
    2 root      20   0       0      0      0 S   0,0  0,0   0:00.04 kthreadd
```

Plusieurs informations sont disponibles : PID, propriétaire, priorité, statut du processus (S=sleeping, D=dead, R=running, Z=zombie), informations sur la mémoire (voir 2.2.2, en particulier `%MEM` le pourcentage de la mémoire de la machine utilisée par le processus), sur le taux d'utilisation du processeur (`%CPU`) et son temps d'activité (`TIME`).

Si un processus a lancé m threads, le `%CPU` peut monter jusqu'à $m \times 100\%$. A noter enfin qu'un `%CPU<100%` indiquera que les calculs sont freinés par des éléments pénalisant, qui viennent très souvent des accès à la mémoire dont nous parlerons dans la section suivante.

2.2.2 Éléments de vocabulaire autour de la mémoire

Registre vs. cache vs. RAM vs. disque

Ce qu'il est important de retenir de l'architecture des machines actuelles est qu'elles comportent en général plusieurs processeurs, eux-même possédant plusieurs cœurs de calcul. Cette complexité pour le programmeur n'est malheureusement pas la seule à prendre en compte. Pour calculer efficacement plusieurs opérations en même temps, ce qui est finalement le but ultime du calcul parallèle, il est nécessaire que les cœurs en charge de l'exécution des calculs puissent accéder de façon tout aussi efficace aux données sur lesquels doivent porter ces calculs. Or, c'est loin d'être le cas. Ces données sont dans la *mémoire* (à ne pas confondre avec le *disque*, appelé *mémoire de masse*). Bien sûr, les technologies liées aux mémoires ont aussi beaucoup progressé, mais la vitesse d'accès à ses composants mémoires n'a malheureusement pas suivi tout à fait la même courbe que celle de la performance des

processeurs. La rumeur suivante court donc dans le milieu du calcul :

Si « ça rame », en général, c'est un problème d'accès aux données dans la mémoire !

.

En résumé, les processeurs (par l'intermédiaire de leurs cœurs de calcul) sont capables de calculer beaucoup plus vite que la mémoire ne peut fournir comme données. Il a fallu donc contourner ces difficultés en intégrant dans les serveurs de calcul une hiérarchie mémoire que nous allons survoler pour en comprendre les mécanismes.

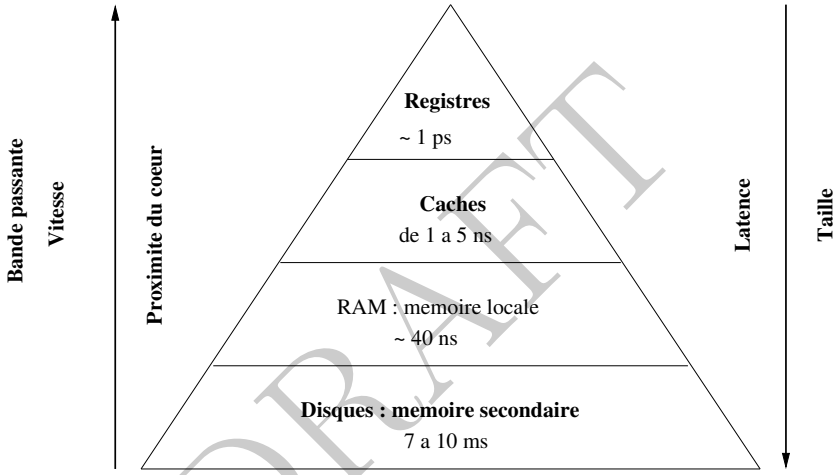


Fig. 2.2 – Différents niveaux de mémoire : taille *vs.* efficacité.

Dans un monde idéal, les données arriveraient au cœur de calcul à la même vitesse que le cœur est capable de les traiter. Cette mémoire existe : ce sont les *registres* du cœur. Mais alors, où est le problème ? Il est purement économique : la technologie mémoire utilisée dans les registres est extrêmement coûteuse, elle ne peut donc être intégrée qu'en infime quantité. En fait, plus une mémoire est efficace, plus elle coûte chère, et moins il y en a dans les serveurs. C'est ainsi que les constructeurs ont intégré dans leurs machines une hiérarchie mémoire, avec des mémoires de moins en moins rapide, mais des tailles mémoires inversement proportionnelle à leur efficacité.
Plus on est proche du cœur de calcul, plus la mémoire doit être rapide. La hiérarchie mémoire que l'on trouve actuellement dans les serveurs de calcul est le plus souvent constituée de la façon suivante, et est schématisée par la figure 2.3 :

— le cache de premier niveau (L1), très rapide et de petite taille ;

— le cache de second niveau (L2), un peu moins rapide et de taille un peu plus importante ;

— le cache de troisième niveau (L3), encore plus lent mais de taille encore plus grande.

Les caches L2 et L3 peuvent être situés à l'intérieur ou hors du cœur. Lorsqu'un cache est localisé à l'extérieur du cœur de calcul, il est en général partagé entre tous les cœurs du processeur. Le fonctionnement de ce système de cache est relativement simple :

— le cœur demande une information ;

— le cache L1 vérifie s'il possède cette information. S'il la possède, il la retransmet au cœur (on parle alors de succès de cache, *cache hit* en anglais). Si il ne la possède pas, il la demande au cache L2 (on parle alors de défaut de cache ou *cache miss*) ; et ainsi de suite ;

— le cache qui possède la donnée la renvoie au demandeur (en cascade s'il y a eu plusieurs défauts de cache), le cheminement final étant depuis L1 vers les registres du cœur ;

— le cache L1 la stocke pour utilisation ultérieure et la retransmet au cœur au besoin.

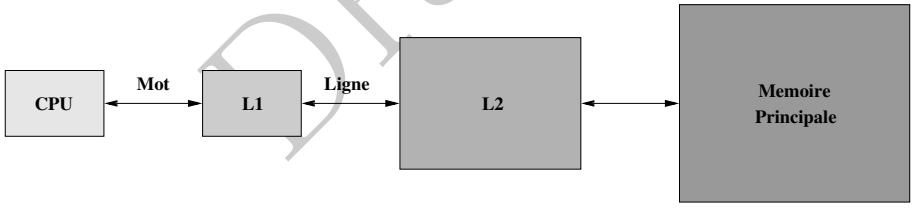
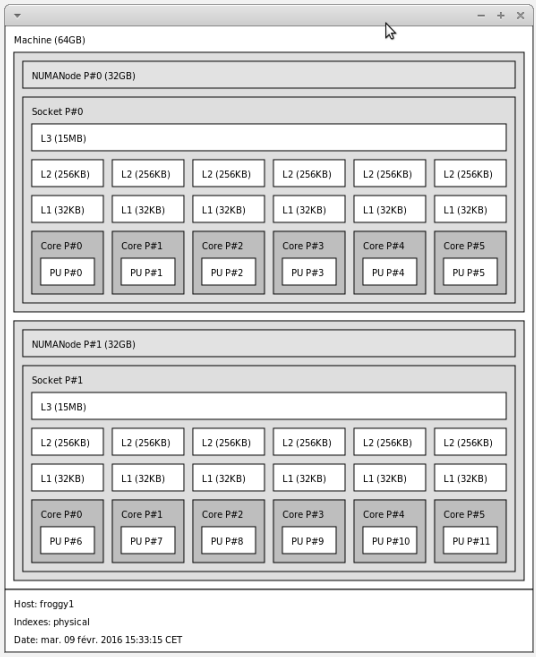


Fig. 2.3 – Niveaux de caches.

Monsieur R. User s'interroge sur les caractéristiques de son serveur Intel SandyBridge bi-sockets hexa-cœurs. Il se tourne donc vers l'outil `hwloc-ls` qui est très utile pour visualiser la topologie locale de sa machine : processeurs, cœurs et niveaux de mémoire hiérarchique. Sur la figure suivante, on voit très clairement que cette commande met en évidence les deux *sockets*, donc deux processeurs, contenant eux-même chacun six cœurs :



On distingue aussi très bien la hiérarchie mémoire avec les trois niveaux de cache, dont le L3 qui est partagé par six cœurs.

Il s'agit bien ici de la configuration matérielle de la machine, aucune mention du système d'exploitation ne figure par exemple. Cet outil est disponible sur Linux et Mac OSX.

Lignes de caches vs. localité

Lorsque le cœur de calcul a besoin d'une donnée, celle-ci doit d'abord être chargée en mémoire cache. Cela entraîne le chargement en cache des éléments qui la suivent de façon contiguë en mémoire (cet ensemble est appelé *ligne de cache*). Cette contiguïté est appelée *localité spatiale*. En général, quand on travaille sur des matrices, on traite un des éléments de la matrice, puis le suivant, puis les autres : c'est ce que l'on appelle la *localité temporelle*, *i.e.* travailler séquentiellement sur des éléments proches en mémoire. Lorsqu'il y a bien conjonction entre localité spatiale et temporelle, l'efficacité de l'accès à la mémoire est garantie par le système des lignes de cache. En effet, si on tient compte du fait que le temps de chargement des données en mémoire est très supérieur au temps d'exécution d'une opération de calcul (addition, ...), cette notion de localité a un impact très fort sur les temps de calcul.

Il est important de noter qu'en R, lorsque l'on instancie une matrice, celle-ci est stockée en mémoire de façon contiguë colonne par colonne. Attention, cette convention varie d'un langage à l'autre : c'est la même chose en `Fortran` par exemple, mais c'est l'inverse en `C/C++` qui implémente un stockage ligne par ligne (il faudra se méfier lors de l'implémentation d'interfaces entre R et `C++`) !

Les accès mémoires pénalisent les codes de calcul... « j'aimerais bien voir ça ! », se dit Monsieur R.User. Pour cela, il se focalise sur une sous-partie de son code très simple qui consiste à retourner le vecteur des sommes des éléments de chaque ligne d'une matrice :

$$v_i = \sum_{j=1,m} A_{ij}, \quad i = 1, n$$

La matrice A sera dans la mémoire colonne par colonne sous la forme :

$$A_{11}\ A_{21}\ A_{31}\ \ldots\ A_{n1}\ A_{12}\ A_{22}\ \ldots\ A_{n2}\ \ldots\ A_{nm}$$

L'algorithme correspondant à cette sommation peut s'écrire de deux façons différentes :

1. deux boucles imbriquées avec la boucle extérieure sur les lignes

```
Pour i de 1 à n
     Pour j de 1 à m
          v[i] = v[i] + A[i,j]
```

2. deux boucles imbriquées avec la boucle extérieure sur les colonnes

```
Pour j de 1 à m
     Pour i de 1 à n
          v[i] = v[i] + A[i,j]
```

Illustrons très schématiquement le temps d'exécution pour chacun de ces deux algorithmes précédents sur les premières itérations. On notera T_a les temps de calcul pour l'exécution de l'addition et T_c celui pour la mise en cache d'une ligne de données :

1^{er} algorithme		
Instruction à exécuter	Données en cache	Temps d'exécution
`v[1] = v[1] + A[1,1]`	`A[1,1], A[2,1], A[3,1], ... A[k,1]`	$T_a + T_c$
`v[1] = v[1] + A[1,2]`	`A[1,1], A[2,1], A[3,1], ... A[k,1]` `A[1,2], A[2,2], A[3,2], ... A[k,2]`	$T_a + T_c$

`v[1] = v[1] + A[1,3]`	`A[1,1], A[2,1], A[3,1], ... A[k,1]`
	`A[1,2], A[2,2], A[3,2], ... A[k,2]`
	`A[1,3], A[2,3], A[3,3], ... A[k,3]`
		$T_a + T_c$

...	...	...
Temps Total $= 3T_a + 3T_c + ...$		

2^e algorithme		
Instruction à exécuter	Données en cache	Temps d'exécution
`v[1] = v[1] + A[1,1]`	`A[1,1], A[2,1], A[3,1], ... A[k,1]`	$T_a + T_c$
`v[2] = v[2] + A[2,1]`	`A[1,1], A[2,1], A[3,1], ... A[k,1]`	T_a
`v[3] = v[3] + A[3,1]`	`A[1,1], A[2,1], A[3,1], ... A[k,1]`	T_a
...	...	...
Temps Total $= 3T_a + T_c + ...$		

Dans le premier algorithme, la localité temporelle ne correspond pas à la localité spatiale, et donc il y a un ralentissement dû au temps de mise en cache de lignes de données. Ceci est d'ailleurs amplifié par le fait que $T_a \ll T_c$.

Illustrons concrètement ce comportement par une implémentation des deux algorithmes. `C++` étant de plus bas niveau que `R` (plus « proche » de la machine, voir Annexe A.3), il est privilégié ici pour illustrer ces questions de localité. Les deux algorithmes sont codées en `C++` et interfacés avec **Rcpp** et **inline** :

```
# 1er algorithme
cppFunction('NumericVector rowSumsIJ(NumericMatrix x){
  int n = x.nrow();
  int m = x.ncol();
  NumericVector rsums(n);
  for (int i=0; i<n; i++)
    for (int j=0; j<m; j++)
      rsums[i] += x(i,j);
  return rsums;
}')

# 2ème algorithme
cppFunction('NumericVector rowSumsJI(NumericMatrix x){
  int n = x.nrow();
  int m = x.ncol();
  NumericVector rsums(n);
  for (int j=0; j<m; j++)
    for (int i=0; i<n; i++)
      rsums[i] += x(i,j);
  return rsums;
}')

n <- 1e3
```

```
m <- 1e4
x <- matrix(rnorm(n*m),n,m)

microbenchmark(rowSumsIJ(x),
               rowSumsJI(x) )
Unit: milliseconds
expr         min   lq    mean  median uq   max
rowSumsIJ(x) 28.41 28.65 28.80 28.75 28.96 29.49
rowSumsJI(x) 10.56 10.62 10.76 10.74 10.86 11.32
```

Le premier algorithme (mauvaise localité) est 3 fois moins efficace que le second (bonne localité).

On retrouve d'ailleurs le même différentiel sur une implémentation en R pur :

```
m <- 1e4
x <- matrix(rnorm(m,m),m,m)
system.time( rowSums(x) )[3]
0.32
system.time( colSums(x) )[3]
0.122
```

Toutefois, la consultation du code source de R montre que `colSums` et `rowSums` dérivent de la même implémentation en `C` du second algorithme (colonne par colonne, les développeurs aillant d'ailleurs laissé un commentaire sur la localité : `/* by columns to improve cache hits */`). Pourtant, sur cet exemple, il existe un différentiel...

2.2.3 Typologie des infrastructures de calcul

Passons désormais à une échelle plus macroscopique. Les ordinateurs actuels utilisés pour du calcul ou des traitements massifs de données peuvent être catégorisés en différentes classes de machines.
Une machine simple multi-cœurs (depuis un portable jusqu'à un serveur de calcul) est composée d'un ou plusieurs processeurs comprenant chacun plusieurs cœurs de calcul. L'ensemble de la mémoire est accessible par toutes les unités de calcul de façon plus ou moins efficace mais transparente pour le développeur. La gestion de la mémoire n'est pas du ressort du programmeur, sauf s'il faut éviter des *accès concurrents* à la mémoire par plusieurs unités de calcul.
Une machine simple multi-cœurs avec une *carte accélératrice* est composé du même matériel que précédemment auquel s'ajoute un ou plusieurs éléments de type carte graphique (*GPU*) ou *many-cœurs* (voir annexe A.4). L'exploitation des cartes n'est

pas simple, même si de nombreux efforts ont été réalisés dans la mise à disposition d'outils permettant de programmer sur ces matériels. Nous n'aborderons pas en détails ici ce type d'architecture (voir en Annexe A.4 des pistes pour approfondir le sujet).

Un *cluster* de calcul multi-nœuds est un troupeau de plusieurs machines des types précédents (appelés *nœuds de calcul*) qui communiquent entre elles à travers un réseau rapide, comme illustré sur la figure 2.4. C'est au développeur que revient la charge d'expliciter l'utilisation des différentes machines et leur interactions lorsqu'elles sont nécessaires. En effet, dans ce cas de figure, chaque nœud accède à sa mémoire directement mais ne peut atteindre la mémoire des autres nœuds que via des communications par le réseau.

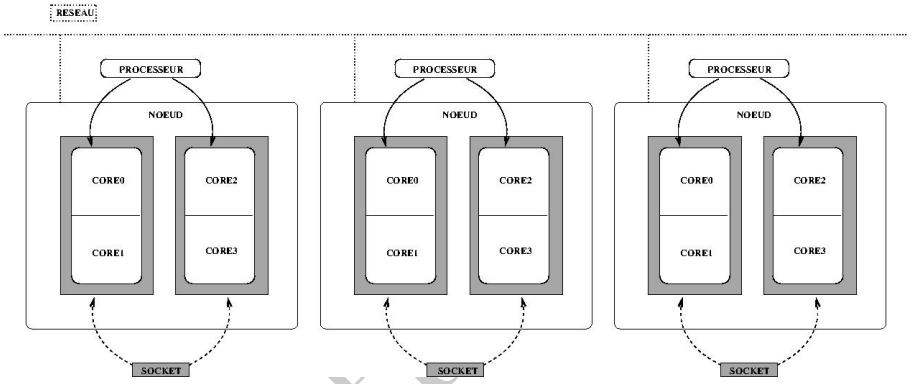


Fig. 2.4 – Cluster de calcul constituée de 3 nœuds bi-sockets (donc bi-processeurs) bi-cœurs.

Enfin, une *grille* de calcul est constituée de plusieurs machines des deux premiers types. C'est une infrastructure virtuelle regroupant des ressources (serveurs, stockage, ...) géographiquement réparties, hétérogènes et autonomes. Une grille de calcul intègre un *middleware* (intergiciel) souvent assez complexe qui permet de voir l'infrastructure comme une seule entité virtuelle. Le principe d'une grille est de mettre à disposition un grand nombre d'unités de calcul, vues comme des éléments individuels, de façon transparente. Du point de vue du développeur, les calculs se résument souvent au lancement multiple d'un même programme.

Bien entendu, il faut prendre en compte cette typologie lors du développement d'un programme destiné à être exploité sur une ou l'autre de ces catégories. Ainsi, les chapitres 3 et 4 emmèneront le programmeur sur une machine multi-cœurs, tandis que le chapitre 5 abordera la problématique des clusters.

2.3 Le calcul parallèle

Les architectures de ce début de siècle mettent à disposition du programmeur un ensemble d'unités de calcul à exploiter. Ceci engendre un changement fondamental dans la conception des codes qui doivent répondre à cette multiplicité d'unités par différents niveaux de parallélisme.

2.3.1 Éléments de vocabulaire

Séquentiel vs. distribué vs. parallèle vs. réparti

Un code R qui est conçu, depuis l'algorithme jusqu'à son implémentation, pour être exécuté sur une unique unité de calcul est dit *séquentiel*. Lorsqu'il s'agit d'utiliser plusieurs unités de calcul, plusieurs situations sont possibles.

La première consiste à exécuter un code R unique et séquentiel pour plusieurs jeux d'entrées (données et/ou paramètres) différents : chaque exécution est indépendante et utilise une unité de calcul. Les résultats de ces calculs peuvent être utilisé *a posteriori* lors d'une étape de post-traitement. On parle alors de *calcul réparti*. Ce type de calcul est finalement peu dépendant de l'architecture puisqu'il revient en définitive à utiliser chacune des unités de calcul avec une exécution séquentielle : que l'on soit sur un nœud multi-cœurs, sur un cluster de calcul, sur une grille, tous les cas sont réalisables.

La deuxième situation consiste à exécuter un code R pour un jeu d'entrées unique (données et/ou paramètres) à l'aide de plusieurs unités de calcul et obtenir un résultat unique. Il s'agit dans ce cas de *calcul parallèle* ou de *parallélisme*. Dans cette situation, le programmeur doit envisager le parallélisme dans chaque étape de conception (algorithme, code, tests) d'autant que deux cas de figure peuvent se présenter : implémenter l'ensemble dans un cadre de parallélisme à *mémoire partagée*, en visant l'exécution sur un seul nœud de calcul ou considérer un calcul *distribué* sur un cluster de calcul.

> Prenons l'exemple de l'analyse de sensibilité et d'incertitude qui consiste à évaluer l'impact de variations des entrées sur la sortie d'un modèle, en réalisant des simulations pour des entrées et paramètres perturbés. Chaque simulation sera réalisée en parallèle par le code séquentiel (en utilisant `Rscript` en série, par exemple) et on tirera profit de toutes les unités de calcul disponibles à chaque instant (potentiellement nombreuses) pour exécuter au fur et à mesure l'ensemble des simulations sans se soucier d'une quelconque dépendance. L'étape finale consistera à rassembler les résultats des simulations, écrits sur le disque dans des fichiers (texte ou binaire de type `.RData`), lors d'une étape de post-traitement. Cette configuration est clairement celle du *calcul réparti*.

Considérons maintenant l'exemple du produit matrice-vecteur : le vecteur résultat est composé du produit de chaque ligne de la matrice par le vecteur ; ces produits peuvent être réalisés par plusieurs unités de calcul en parallèle. Dans ce cas, il y a bien une unité de résultat qui est calculé de manière coopérative, on parle bien de *calcul parallèle*.

Tâche vs. granularité, scalabilité vs. overhead

Nous avons déjà abordé quelques éléments de terminologie associés aux aspects matériels dans les parties 2.2.1 et 2.2.2. Le calcul parallèle possède lui aussi son vocabulaire spécifique qu'il est bon de maîtriser afin de bien identifier les notions manipulées.
Une *tâche* est une unité indivisible issue du partage algorithmique du travail : une tâche représente une quantité de calculs à réaliser (sommer les éléments d'une ligne de matrice par exemple).
Pour prétendre qu'une tâche est indivisible, on choisira un niveau de *granularité* :

— une décomposition en nombreuses tâches de petite taille est appelée *grain-fin* (*fine-grained*) ;

— au contraire, une décomposition en un petit nombre de tâches conséquentes est appelé *gros-grain* (*coarse-grained*).

Le calcul parallèle est très lié aux problématiques de performances. Le moyen le plus simple pour analyser la performance d'une application est de mesurer son temps d'exécution (voir 1.2.1). Le *speedup* ou *accélération* obtenue après une amélioration du code s'exprime très simplement :

$$A = T_{init}/T_{new}$$

où T_{init} est le temps d'exécution de la version initiale, et T_{new} celui de la version améliorée.
Cette accélération peut être en particulier due à la parallélisation de l'application considérée, ce qui nous amène à la notion de *scalabilité*. La scalabilité mesure la capacité d'un programme à « passer à l'échelle », c'est-à-dire d'augmenter ses performances lorsqu'on augmente les ressources (en particulier le nombre d'unités de calcul). On distingue deux types de scalabilité :

— scalabilité forte : on fixe la taille du problème et on augmente le nombre d'unités de calcul. Ce type de scalabilité indique une accélération plus ou moins proportionnelle au nombre d'unités de calcul utilisées ;

— scalabilité faible : on augmente la taille du problème en même temps que le nombre d'unités de calcul. Ce type de scalabilité indique la capacité de maintenir un temps à la solution fixe lors de la résolution d'un problème plus gros sur un nombre d'unités de calcul plus important.

Enfin, la notion d'*overhead* mesure le surcoût (en terme de ressources qui se traduit généralement par du temps d'exécution supplémentaire) engendré par la mécanique du parallélisme. Ce surcoût est lié à des opérations qui ont lieu au niveau du système, de la mémoire, de la couche réseau... L'overhead est bien entendu associée à la scalabilité, car ces deux notions s'opposent.

> Reprenons l'exemple du produit matrice-vecteur, en supposant que l'on a plusieurs matrices à traiter. Chaque produit matrice-vecteur est une tâche indivisible d'un parallélisme *gros-grain*. Par contre, le produit de chaque ligne de matrice par le vecteur est une tâche indivisible d'un parallélisme *grain-fin*.
> Si le nombre de matrices est grand devant le nombre d'unités de calcul, on aura alors une bonne *scalabilité* avec un bon *speedup*, car chaque unité de calcul aura du grain à moudre (*i.e* des matrices à traiter). On comprend cependant vite que le gros-grain peut s'avérer sous-optimal, si on n'a que quelques matrices mais des dizaines d'unités de calcul à disposition qui du coup seront pour certaines inutilisées. Dans ce cas, un parallélisme grain-fin sera approprié car le nombre de lignes de matrice sera potentiellement grand devant le nombre d'unités de calcul et donc, de nouveau, chaque unité de calcul aura du grain à moudre (*i.e* des lignes de matrices à traiter). Cependant, la mécanique derrière ce parallélisme pourra occasionner un *overhead* du fait du grand nombre de tâches à réaliser en parallèle (nombre de matrices × nombre de lignes).

2.3.2 Penser parallèle

Avant d'allumer l'ordinateur et de programmer en R, il peut être utile de prendre un papier et un crayon pour concevoir un algorithme. Concevoir un algorithme séquentiel, c'est identifier une suite d'étapes, de blocs de calculs (on parlera de tâches) à réaliser dans un ordre défini. Concevoir un algorithme parallèle s'avère plus complexe et passe par les étapes suivantes (voir également Grama *et al.* (2003)) :

1. identifier les tâches indivisibles (on parle de *décomposition*) ;
2. lister les dépendances de ces tâches, *i.e* leur ordre relatif ;
3. évaluer (si possible) la taille des tâches ;
4. en conséquence, réaliser une association entre unités de calcul et tâches à réaliser, en respectant les dépendances ;
5. synchroniser régulièrement les unités de calcul ;
6. gérer les lectures/écritures de données communes.

Pour définir les tâches indivisibles, comment arbitrer entre gros-grain et grain-fin ? La réponse est : cela dépend... Gros-grain rime avec facilité d'implémentation mais

mauvaise scalabilité, grain-fin rime avec scalabilité mais induit de l'overhead dû au surcoût de la machinerie parallèle.
Comme mentionné précédemment, la décomposition du problème est la première étape de la parallélisation. « Penser parallèle », c'est envisager l'une des décompositions suivantes (voir Grama *et al.* (2003) pour plus de détails) :

— *décomposition par les données* : partitionnement des données d'entrée à traiter ou de sorties à produire. L'exemple le plus classique consiste à partitionner un matrice en blocs de lignes (et/ou colonnes). Ceci est illustré dans le pseudo-exemple R non parallélisé ci-après :

```
apply(big.matrice, 1, FUN=...) # partitionnement de la matrice
                               # d'entrée, par ligne
lapply(1:1e5, FUN=...)         # partitionnement des 1e5 sorties
                               # à produire
```

Un autre exemple consistera à paralléliser les itérations indépendantes d'une boucle, chaque itération agissant sur une partie des données. On parle ici de *parallélisme embarrassant* (rien de grave en soi, cela précise simplement que le challenge intellectuel associé est mineur).

— *décomposition récursive* : spécifique aux problèmes résolus par une approche *divide and conquer*. Les sous-problèmes sont traités en parallèle puis les résultats sont fusionnés. Cette décomposition revient à la décomposition par les données, avec en plus l'étape de fusion.

— *décomposition fonctionnelle ou par les tâches* : au lieu de partitionner les données et de faire le même enchaînement de calculs sur chaque unité, on partitionne ici les tâches. Les chaînes de traitements (ou pipeline, le résultat d'une tâche est l'entrée de la suivante), très courante en bioinformatique ou en traitement d'image, sont également des candidates à la décomposition fonctionnelle. Cette décomposition sera rapidement abandonnée, du fait de son absence de scalabilité : en effet, le nombre d'unités de calcul disponibles sera rapidement plus grand que le nombre de tâches. Dans le pseudo-exemple R non parallélisé qui suit, cette décomposition reviendrait à paralléliser le calcul des résultats `r1,r2,...r10`... sur au plus 10 unités de calcul !

```
apply(big.matrice, 1, FUN=function(v){
 r1 <- sd(v)       # partition en 10 calculs distincts
 r2 <- mean(v)
 ...
 r10 <- sum(v^2)
 return(list(r1,r2,...r10))
})
```

— *décomposition exploratoire* : adapté à la recherche de solutions dans des grands espaces. L'espace des solutions est partitionné puis exploré en parallèle jusqu'à une synchronisation. La meilleure solution temporaire est retenue

et on partitionne de nouveau la partie de l'espace sélectionnée, et ainsi de suite ; cette stratégie est appliquée pour les Monte Carlo Markov Chain parallèle (Jacob *et al.*, 2011). Plus prosaïquement, cette exploration correspond à la fonction `kmeans` qui tire plusieurs points d'initialisation et les itérations qui suivent peuvent être réalisée en parallèle.

2.4 Limites aux performances

Quel que soit ce que le futur réservera au niveau technologique, il est clair que les processeurs offriront des performances toujours plus importantes grâce à la parallélisation, à travers un nombre d'unités de calcul par processeur toujours plus important. Le développement d'un code parallèle ajoute cependant un niveau de complexité au processus de programmation. Il est donc nécessaire de bien évaluer le gain théorique de performance lié à l'utilisation de plusieurs unités de calcul. La parallélisation présente-elle un avantage pour tous les problèmes ? Comment estimer le succès de la parallélisation d'un programme ?

Plusieurs lois, qui sont plutôt des arguments numériques, permettent ainsi d'évaluer l'efficacité du parallélisme.

2.4.1 Loi d'Amdahl

La première de ces lois fut énoncée par G. Amdahl en 1967. Celui-ci est parti du principe que certaines phases de l'exécution d'un programme sont par nature intrinsèquement séquentielles, et ne tirent donc pas profit de la parallélisation de l'application. Par exemple, les entrée-sorties d'un code de calcul sont, la plupart du temps, séquentielles. La figure 2.5 illustre un cas classique avec une partie calculatoire potentiellement parallélisable et une partie liée aux sorties des résultats, qui ne peut pas être parallélisée. On normalise à 1 le temps de calcul séquentiel composé des parties séquentielle s et parallélisable p :

$$T(1) = s + p = 1$$

Le temps de l'application parallélisée sur m unités de calcul sera donc :

$$T(m) = s + p/m$$

L'accélération s'exprime donc sous la forme de la *loi d'Amdahl* (Amdahl, 1967) :

$$\textit{accélération}(m) = \frac{T(1)}{T(m)} = \frac{1}{s + \frac{1-s}{m}}$$

Cette accélération est valable pour une taille de problème constante et correspond à la mesure de la scalabilité forte (voir 2.3.1). Si la partie séquentielle est inexistante

ou négligeable devant p, l'accélération est égale à m. Si on augmente à l'infini le nombre d'unités de calcul :

$$\lim_{m \to +\infty} accélération(m) = \frac{1}{s}$$

La partie séquentielle du code limite le gain du parallélisme. Ainsi, si seule la moitié du programme est parallèle, le facteur d'accélération ne dépassera pas 2 même sur le plus gros supercalculateur au monde !

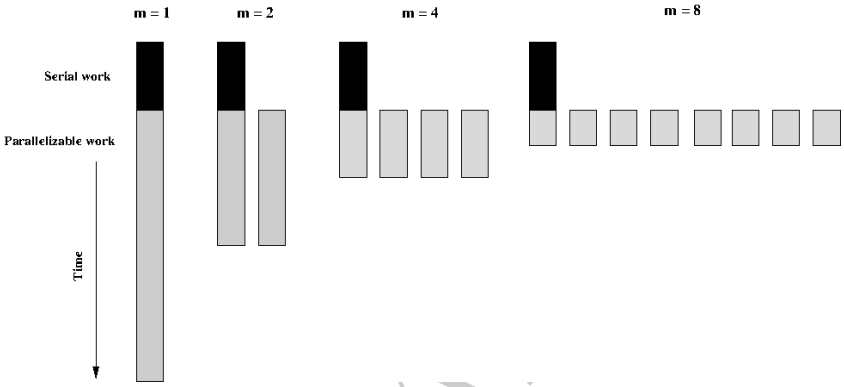


Fig. 2.5 – Partie séquentielle (en noir) et partie parallélisable (en gris) dans un code. Dans la loi d'Amdahl, l'accélération est limitée par la partie séquentielle de l'exécution.

La loi d'Amdahl n'est pas très réaliste, elle intègre implicitement plusieurs hypothèses (accélération parfaite de la partie parallèle, pas d'overhead lié au parallélisme...) qui ne sont pas pertinentes.

2.4.2 Loi de Gustafson

La *loi de Gustafson* énoncée en 1988 (Gustafson, 1988) est plus pragmatique que la loi d'Amdahl : elle prend en compte le fait que, lorsqu'on dispose d'un nombre d'unités de calcul important, on réalise des calculs plus conséquents sur des données plus volumineuses ou avec des modèles plus complexes (scalabilité faible, voir 2.3.1). Dans ce cas, la partie séquentielle devient alors plus petite en relatif par rapport au problème global. Gustafson part donc du principe que la taille de l'application croît proportionnellement au nombre de ressources disponibles (m unités de calcul) alors que le temps de la partie séquentielle reste à peu près constant :

$$accélération(m) = \frac{s + m(1-s)}{s + (1-s)} = s + m(1-s)$$

Ainsi, la partie parallèle du travail $(1-s)$ diminue avec l'augmentation du nombre de processeurs dans le numérateur de l'expression, alors que la partie séquentielle n'est pas modifiée. Cette loi caractérise le parallélisme de données : plus celles-ci sont importantes, plus le parallélisme est efficace. Et il n'y a plus de limite sur le nombre de processeurs m.

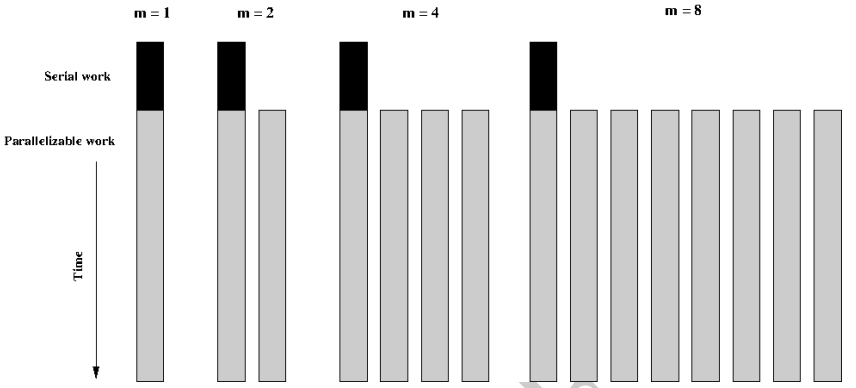


Fig. 2.6 – Dans la loi de Gustafson, si la taille du problème augmente avec le nombre de ressources disponibles alors que la partie séquentielle reste quasiment fixe, l'accélération croît avec le nombre d'unités de calcul supplémentaires.

En conclusion, ces formules théoriques illustrent deux notions :

— paralléliser un programme qui exécute de nombreux calculs en parallèle sur le même ensemble de données est limité par la loi d'Amdahl ;

— le parallélisme de données, consistant à effectuer un même calcul sur un ensemble de données dont la taille augmente est efficace en terme de nombre d'unités de calcul.

2.4.3 Et dans la vraie vie

Amdahl et Gustafson se concentrent sur le comportement théorique en terme de performance dans des cas idéaux. Cependant, dans les situations réelles, de nombreux paramètres limitant interviennent, paramètres qui dépendent à la fois de l'architecture matériel utilisée, du langage de programmation et du codage lui-même :

— du point de vue matériel, certaines ressources sont partagées (par exemple les bus de communication entre les cœurs et la mémoire, le réseau...) et ne peuvent donc être utilisées que de façon séquentielle même si cela reste transparent pour le développeur ;

— du point de vue programmation et algorithme, certains éléments comme les entrées/sorties (lire les données pour réaliser le calcul et écrire les résultats sur le disque) sont intrinsèquement séquentiels, et peuvent être très pénalisants ;

— l'utilisation des outils du parallélisme entraîne un overhead (voir 2.3.1) : gestion de la création des processus ou des threads, des synchronisations, de l'accès concurrent aux données, des communications...

— il n'est pas rare d'avoir des problèmes d'équilibrage de charge, certaines unités de calcul ayant des tâches plus importantes à traiter et entraînant des temps d'attente pour les autres unités.

Les éléments théoriques d'Amdahl et de Gustafson sont donc une base importante pour la compréhension du comportement des applications parallèles, mais le réel, le réel et encore le réel pourra réserver des surprises au niveau des performances ! « *The free lunch is over* », décidément.

Chapitre 3

Calcul parallèle avec R sur machine multi-cœurs

Nous nous plaçons ici dans le cadre d'architectures matérielles constituées par une seule machine, composée d'un ou plusieurs processeurs, eux-même intégrant plusieurs cœurs de calcul. Les principales caractéristiques de ces machines (qui vont du simple portable au serveur de calcul) sont rappelées dans le chapitre précédent. Tirer profit de ce type d'architecture avec R est un défi du quotidien : il s'adresse à tous les usages, depuis les développeurs qui souhaitent proposer un package performant jusqu'aux « *data analysts* » qui interagissent avec R et implémentent des chaînes de traitement de données les plus efficaces possibles.

3.1 Principe général

R propose désormais nativement une ensemble de fonctionnalités intuitives pour mettre en œuvre une implémentation parallèle. Supposons que m unités de calcul sont à disposition - sans préciser comment elles peuvent communiquer et sans préciser l'architecture matérielle.
Le schéma classique implémenté dans R correspond aux étapes suivants :

1. un processus R *père* est lancé et servira de chef d'orchestre ;
2. m processus R fils sont démarrés ;
3. le processus père découpe le travail et les données associées (s'il y en a) et le répartit entre les processus fils ;
4. le processus père collecte les résultats du travail réalisé par les fils, puis éventuellement réalise une synthèse globale ;
5. le père arrête les processus R fils et retourne le résultat global.

Ce schéma sera particulièrement efficace lors de la parallélisation des boucles qui, de fait, correspond à une part importante des possibilités offertes par R au niveau

du parallélisme. A noter que l'implémentation de ce schéma ne sera pas à la charge du développeur, néanmoins la mécanique sous-jacente mérite d'être maîtrisée et sera présentée dans la suite.

3.2 Le package `parallel` et son utilisation

Le package **parallel** est inclus dans R depuis sa version `2.14.0` (31 octobre 2011). Il propose un ensemble de fonctionnalités pour introduire du parallélisme dans les codes R : il s'agit également d'attirer le plus grand nombre d'utilisateurs de R et développeurs de packages vers le calcul parallèle compte tenu de l'évolution des architectures matérielles. **parallel** est dérivé des packages CRAN **multicore** (2009-) et **snow** (2003-) : il propose des fonctions de substitutions à la plupart des fonctionnalités de ces deux packages, en intégrant par ailleurs la gestion des générateurs de nombres pseudo-aléatoires parallèles.
De nombreux codes R intègrent les fonctions de la famille `*apply` qui sont des versions avantageuses des boucles `for` (en terme de performance souvent et en terme de lisibilité). Dans cette logique fonctionnelle, la plupart des utilisateurs de **parallel** se focaliseront sur les versions parallèles de ces fonctions : `parApply, parLapply, parSapply, parVapply` et autres (dans la suite, on parlera des `par*apply`). Ces fonctions reprennent les même arguments que les versions séquentielles, mais nécessitent la création d'un pool de processus fils avec la fonction `makeCluster`, le passage de ce pool en premier argument, puis la suppression de ce pool :

```
library(parallel)
cl <- makeCluster(<taille du pool>)
# un ou plusieurs appels à parApply/parLapply/parSapply
parLapply(cl, <même arguments que lapply>)
stopCluster(cl)
```

Les fonctions `par*apply` reprennent le schéma expliqué en 3.1, le processus père découpe les itérations de boucles en paquets (dits *chunks*) puis les dispatche vers les fils, pour finalement retourner le même objet que la version séquentielle de la famille `*apply`. Les sorties standards `stdout` et `stderr` des processus fils sont détruites par défaut, néanmoins il est possible les enregistrer sur le disque avec l'option `outfile` de `makeCluster`.
Ces fonctions parallèles encapsulent différents mécanismes qui varient selon le type de pool de fils que l'on crée : dans les paragraphes suivants, on verra comment spécifier des mécanismes hérités de **snow** (compatible Windows/Mac OSX/Linux) ou de **multicore** (compatible Mac OSX/Linux), avec les conséquences sur les performances générales.

> Supposons que l'on souhaite estimer la qualité de prédiction d'un modèle linéaire, ici un modèle linéaire pour la régression de la largeur d'une pétale sur sa longueur sur le jeu de données `iris` de R. Une technique envisageable

est le « *leave-one-out* » qui consiste à estimer l'erreur de généralisation (erreur faite pour des nouveaux individus qui viennent de la même distribution que les individus utilisés pour apprendre le modèle). Le principe est simple : on estime le modèle avec tous les individus sauf un, on fait une prédiction pour cet individu, et on regarde l'erreur quadratique (*PRESS -Prediction Sum of Squares- statistic*) entre la prédiction et la valeur connue ; on répète l'opération pour chacun des individus et on somme les erreurs obtenues.

```
leave.one.out <- function(i){
 model <- lm(Petal.Width ~ Petal.Length, data=iris[-i,])
 pred.petal.width <- predict(model,
  data.frame(Petal.Length=iris[i,"Petal.Length"]))
 return((pred.petal.width-iris[i,"Petal.Width"])^2)
}
```

Il faut donc appeler la fonction `leave.one.out` pour chacun des 150 individus (fleurs) du jeu de données, de la façon suivante par exemple :

```
microbenchmark(
lapply(1:150, FUN=function(i) leave.one.out(i)),
times=25
)
Unit: milliseconds
min    lq     mean   median  uq    max
329.25 334.27 338.89 336.14 338.71 371.83
```

Chaque appel de la fonction étant indépendant, on peut donc « penser parallèle » et observer un cas de parallélisme embarrassant comme défini en 2.3.2. Dès lors, on tirera avantage à utiliser la version parallèle de `lapply`, à savoir `parLapply`, en reprenant le canevas présenté précédemment :

```
cl <- makeCluster(4)
# + appel à d'autres fonctions auxiliaires
# (voir suite de l'exemple en 3.2.1)

microbenchmark(
 parLapply(cl, 1:150, fun=function(i) leave.one.out(i)),
 times=25
)
min   lq     mean   median uq     max
97.20 102.58 108.14 104.01 106.27 137.65

stopCluster(cl)
```

L'approche « *leave-one-out* » parallèle est environ 3.3 fois plus rapide que la version séquentielle avec l'utilisation de 4 unités de calcul. Cette ac-

célération est très bonne quoique inférieure à 4 (la mécanique interne du parallélisme induit un léger coût, la création/gestion de la liste des résultats étant par exemple séquentielle). Ca vaut le coup !

3.2.1 L'approche `snow`

Il est donc possible d'utiliser **parallel** dans la continuité de **snow**. Dans ce cas, les processus fils sont lancés au moyen d'une commande `system("Rscript")` (ou similaire) et en conséquence les fils sont des processus indépendants semblables au processus père. Ces processus utilisent différents mécanismes de transport pour communiquer : on se focalisera dans cet ouvrage sur les *sockets* qui permettent d'établir une session TCP entre processus, puis de recevoir et d'expédier des données grâce à elle par le réseau (ne pas confondre avec les sockets matérielles évoquées en 2.2.1). Les processus père et fils sont donc reliés par des « tuyaux d'information » par lesquels transitent les données et les résultats de calcul. Pour créer un pool de processus sur ce modèle, on utilisera l'option `type` :

```
cl <- makeCluster(<taille du pool>, type="PSOCK")
```

ou tout simplement :

```
cl <- makeCluster(<taille du pool>)
```

Ce mécanisme basé sur les sockets est multi-plateforme (*i.e.* compatible avec les trois systèmes d'exploitation Linux, Mac OSX et Windows), c'est pourquoi l'approche `snow` peut être préférée lors du développement d'un package pour le CRAN. Elle peut également être utilisée dans un branchement qui teste le type de système :

```
if(Sys.info()[['sysname']]=="Windows"){
 # approche snow
}
```

Les processus père et fils sont indépendants, prêts à communiquer mais cela ne veut pas dire qu'ils partagent des données ou tout autre information présente en mémoire. Comment est-il donc possible qu'un processus fils travaille avec des éléments qui sont dans la mémoire associée au processus père ? Quelles sont les conséquences de ce mécanisme de socket sur le fonctionnement des `par*apply` ? Nous allons voir que le développeur devra veiller à 1/ initialiser correctement les processus fils, 2/ organiser le transferts de données depuis le père vers le fils et 3/ surveiller la consommation mémoire induite par le parallélisme de type **snow**.

L'initialisation des processus fils

L'approche `snow` impose de configurer au préalable chaque processus fils. Les processus fils doivent être initialisés de façon conjointe ; dans le cas contraire, les

processus fils vont émettre des erreurs. Pour cela, le programmeur devra ouvrir ses deux yeux, le premier sur la configuration et le second sur les objets.
Premièrement, il faut charger les packages et données nécessaires grâce aux fonctions `clusterEvalQ` et `clusterCall` : ces fonctions permettent d'évaluer une expression ou de réaliser un appel à une fonction (en particulier les fonctions `library()` et `data()`) sur chaque processus fils en parallèle.
En second lieu, concernant les objets (données et fonctions) nécessaires à la réalisation des calculs par les processus fils, il faut s'assurer que chaque processus fils en a reçu une copie soit de manière implicite (*i.e.* automatique), soit de manière explicite en utilisant la fonction `clusterExport` : cette fonction provoque un transfert de données par les sockets depuis le père vers tous les fils séquentiellement (l'un après l'autre... ce qui peut faire perdre du temps).

Dans l'exemple suivant, nous réalisons un clustering sur les données `Boston` du package **MASS** avec l'algorithme *k-means*. Cet algorithme étant itératif et nécessitant un point de départ, il est recommandé de tester `nstart` points de départ aléatoires et de conserver le meilleur résultat :

```
library(MASS)
res <- kmeans(Boston, 4, nstart=2000)
```

Naturellement, nous pouvons réaliser les *nstart* calculs en parallèle (comme proposé dans McCallum & Weston (2011)), en veillant à fixer la graine du générateur aléatoire à des valeurs différentes sur chacun des fils :

```
library(parallel)
library(MASS)
seeds <- runif(2,1L,.Machine$integer.max)

do.kmeans <- function(i){ # i numéro de la graine à utiliser
 set.seed(seeds[i])
 kmeans(Boston, 4, nstart=1000)
}

cl <- makeCluster(2)
par.res <- parLapply(cl, 1:2, fun=do.kmeans)
Erreur dans checkForRemoteErrors(val) :
2 nodes produced errors; first error: objet 'seeds' introuvable
```

L'exécution se termine par une double erreur (une par processus fils) : l'objet `seeds`, créé par le processus père, n'est pas connu des fils. Il faut donc le transférer explicitement :

```
clusterExport(cl, list("seeds"))
```

On reprend donc notre appel à `parLapply` :

```
par.res <- parLapply(cl, 1:2, fun=do.kmeans)
Erreur dans checkForRemoteErrors(val) :
2 nodes produced errors; first error: objet 'Boston' introuvable
```

L'objet `Boston`, mis à disposition au moment du chargement du package **MASS**, n'est pas non plus connu par les processus fils. Il suffit donc de rajouter l'une des lignes suivantes avant l'appel à `parLapply` pour charger le package **MASS** au niveau des fils :

```
clusterEvalQ(cl, library(MASS))
# OU
clusterCall(cl, function() {library(MASS)})
```

Une autre solution peut consister à transférer l'objet `Boston` explicitement :

```
clusterExport(cl, list("Boston"))
```

La nécessité d'organiser (*i.e.* implémenter explicitement) ou non leur transfert entre le père et les fils repose sur le concept clé d'*environnement* en R (voir Wickham (2014) pour des rappels sur cette notion) :

— tous les objets qui appartiennent à l'environnement global `.GlobalEnv` et qui sont utiles pour les calculs en parallèle doivent être explicitement transférés aux processus fils avec la fonction `clusterExport`. On rappelle que tous les objets sont dans l'environnement global lorsque l'on code en mode interactif dans l'invite de commande de R. Dans l'exemple suivant de l'implémentation naïve du produit matrice-vecteur, le vecteur `V` devra ainsi être explicitement transféré avec `clusterExport` :

```
cl <- makeCluster(2)
M <- matrix(rnorm(1e6), ncol=1e2)
V <- rnorm(1e2)
product = parApply(cl, M, 1, FUN=function(M_i) M_i%*%V)
Erreur dans checkForRemoteErrors(val) :
2 nodes produced errors; first error: objet 'V' introuvable

clusterExport(cl, list("V"))
product <- parApply(cl, M, 1, FUN=function(M_i) M_i%*%V)
stopCluster(cl)
```

On notera que la matrice `M` n'a pas besoin d'être transférée, car elle est traitée par le père au niveau du `parApply` : `M` est découpée et dispatchée vers les fils automatiquement ;

— les éléments en argument des fonctions `par*apply` sont automatiquement transférés. On peut ainsi reprendre l'exemple précédent et ajouter le vecteur `V` en paramètres, de sorte que son transfert vers les fils est automatique :

```
M <- matrix(rnorm(1e6), ncol=1e2)
V <- rnorm(1e2) # pas besoin de clusterExport
product <- parApply(cl, M, 1, FUN=function(M_i, V) M_i%*%V, V)
```

— tout ce qui appartient à un autre environnement sera *sérialisé* (copié dans son ensemble). Ce sera le cas à l'intérieur d'une fonction, dans les classes S4, etc. Pour notre exemple, on peut encapsuler le produit matrice-vecteur dans une fonction, et en conséquence les objets `M` et `V` seront dans l'environnement de la fonction (donc pas dans `.GlobalEnv`) :

```
do.product <- function(cl){
 M <- matrix(rnorm(1e6), ncol=1e2)
 V <- rnorm(1e2) # pas besoin de clusterExport
 product <- parApply(cl, M, 1, FUN=function(M_i) M_i%*%V)
}
```

Le corollaire de ce mécanisme est que tout objet inutile pour le calcul en parallèle sera malgré tout copié et transféré. Le petit test suivant montre bien que le variables `M` et `V` sont copiés vers les fils sans que cela soit nécessaire :

```
serialisation.test <- function(cl){
 M <- matrix(rnorm(1e6), ncol=1e2)
 V <- rnorm(1e2)
 parSapply(cl, 1:2, FUN=function(i){ # ne fait rien
  exists("M") & exists("V") # et pourtant M et V sont copiés
 })
}
serialization.test()
[1] TRUE TRUE
```

Si, dans cet exemple de test, `V` occupait 50% de la mémoire, alors sa duplication vers les 2 fils demanderait $3 \times 50\%$ de la mémoire... crash de la machine en vue, tout ça pour ne rien faire !

Reprenons notre exemple du « *leave-one-out* ». Sans précaution, nous obtenons une erreur :

```
cl <- makeCluster(4)
parLapply(cl, 1:150, fun=function(i) leave.one.out(i))
Erreur dans checkForRemoteErrors(val) :
4 nodes produced errors; first error: impossible de trouver
la fonction "leave.one.out"
```

Il faut en effet initialiser les fils de façon à ce qu'ils connaissent la fonction `leave.one.out` :

```
cl <- makeCluster(4)
clusterExport(cl, list("leave.one.out"))
```

```
parLapply(cl, 1:150, fun=function(i) leave.one.out(i))
stopCluster(cl)
```

Le jeu de données `iris` utilisé fait partie du package **datasets** qui est automatiquement chargé dans R et est automatiquement mis à disposition. Si cela n'avait pas été le cas, il aurait fallu ajouter la phase d'initialisation des fils suivante :

```
clusterEvalQ(cl, library(datasets))
clusterEvalQ(cl, data(iris))
```

Supposons maintenant qu'on ne travaille pas sur `iris` mais sur un autre objet `my.iris`.

```
my.iris <- iris
# ou bien :
# my.iris <- read.csv("iris.txt", header=T)
leave.one.out <- function(i){
 model <- lm(Petal.Width ~ Petal.Length, data=my.iris[-i,])
 pred.petal.width <- predict(model,
  data.frame(Petal.Length=my.iris[i,"Petal.Length"]))
 return((pred.petal.width-my.iris[i,"Petal.Width"])^2)
}
clusterExport(cl, list("leave.one.out"))
parLapply(cl, 1:150, fun=function(i) leave.one.out(i))
Erreur dans checkForRemoteErrors(val) :
4 nodes produced errors; first error:
objet 'my.iris' introuvable
```

Il faudra donc explicitement le transférer vers les fils comme suit :

```
clusterExport(cl, list("leave.one.out","my.iris"))
parLapply(cl, 1:150, fun=function(i) leave.one.out(i))
```

L'inflation mémoire induite par l'approche snow

Dans le schéma suivant, `parLapply` découpe la liste `L` en autant de sous-listes qu'il y a de processus fils et ces sous-listes vont transiter via les sockets (voir Figure 3.1 (a)) :

```
cl <- makeCluster(<taille du pool>, type="PSOCK")
L <- <liste grande taille>
parLapply(cl, L, FUN=...)
stopCluster(cl)
```

Ainsi, l'objet sur lequel s'applique le `par*apply` est transféré par morceaux. Si la quantité de données qui transite est non négligeable par rapport au temps de calcul (voir l'exemple extrême ci-dessous) et *a fortiori* si l'appel aux `par*apply` est fréquent, les performances du parallélisme peuvent s'avérer très mauvaises, comme le montre l'exemple suivant :

```
L <- lapply(1:5e2, matrix, nrow = 1e3, ncol = 1e3 )
object_size(L)
2 GB
system.time( lapply(L, FUN=function(e) {NULL} ) )[3]
0
system.time( parLapply(cl, L, fun=function(e) {NULL} ) )3]
5.259
# >5 secondes pour la création des sous-listes et leur transfert!
```

Par ailleurs, `parApply` encapsule un appel à `parLapply` précédé d'opérations de conversions/duplications (`matrix` vers `list`) qui occasionnent des pertes de performance en temps mais surtout un encombrement de la mémoire qui peut s'avérer critique (voir Figure 3.1 (b)).

(a)

```
%MEM     TIME+ COMMAND
24,5   0:10.85 R
12,4   0:03.31 R
12,4   0:03.32 R
```

(b)

```
%MEM     TIME+ COMMAND
19,2   0:11.66 R
 4,7   0:08.74 R
 4,7   0:08.81 R
```

Fig. 3.1 – Visualisation de la mémoire associée aux processus père et fils avec la commande `top` (voir 2.2.1) dans l'approche **snow**, suivant l'initialisation `makeCluster(2, type="PSOCK")` et après la fin des transferts réalisées avec les sockets, (a) lors de l'appel à `parLapply` et (b) lors de l'appel à `parApply`. (a) Les deux processus fils ont chacun une moitié de l'objet original présent dans la mémoire du processus père, l'empreinte mémoire totale étant ainsi doublée. (b) Comme (a) pour les processus fils mais le processus père utilise le double de mémoire que celle requise pour l'objet original du fait d'une conversion de `matrix` vers `list`.

Enfin, dans le contexte actuel de la manipulation de données massives, la question des transferts d'objets lors de l'initialisation des fils pourra représenter un frein significatif à l'utilisation des fonctionnalités héritées de **snow**... En effet, les duplications d'objets sur les fils occasionnent un encombrement de la mémoire car, rappelons-le, nous sommes dans le cadre de l'utilisation d'une seule machine. Il est néanmoins possible de faire un peu de gymnastique pour éviter les transferts indésirables ! En effet, le programmeur a la possibilité de basculer des objets ou des fonctions « encombrantes » vers l'environnement global `.GlobalEnv` afin de limiter les transferts automatiques :

```
environment(big.data) <- .GlobalEnv
# big.data devra être transféré explicitement si besoin
```

Cette gymnastique, qui est expliquée plus en détail dans McCallum & Weston (2011), est illustrée dans l'exemple suivant où différentes situations conduisent ou non à la sérialisation d'objets non souhaités :

```
parSleep <- function(){
 M  <- matrix(rnorm(1e3*3e4), ncol=3e4)
 M2 <- matrix(rnorm(1e3*3e4), ncol=3e4)

 doSleep1 <- function(i) {
  Sys.sleep(i)
  mem_used()
 }
 # sérialisation automatique de M et M2
 # car doSleep1 est dans l'environnement de parSleep

 doSleep2 <- function(i) {
  Sys.sleep(i)
  mem_used()
 }
 environment(doSleep2) <- .GlobalEnv
 # empèche la sérialisation de M et M2

 doSleep3 <- function(i, param.M) {
  Sys.sleep(i)
  mem_used()
 }
 environment(doSleep3) <- .GlobalEnv
 # idem, sauf que param.M en paramètre est serialisé

 library(pryr)
 mem0 <- mem_used()

 cl <- makeCluster(2, type="PSOCK")
 clusterEvalQ(cl, {library(pryr); NULL})

 mem1 <- parLapply(cl, list(5,5), doSleep1)
 mem2 <- parLapply(cl, list(5,5), doSleep2)
 mem3 <- parLapply(cl, list(5,5), doSleep3, M)
 # param.M=M donc M sérializé

 stopCluster(cl)
```

```
 list(mem0, mem1,mem2,mem3)
}

parSleep()
[[1]]
502 MB     # mémoire total sur le processus père

# version doSleep1
[[2]]
[[2]][[1]]
501 MB     # M et M2 copiés sur les processus fils
[[2]][[2]]
501 MB

# version doSleep2
[[3]]
[[3]][[1]]
21.1 MB    # M et M2 pas copiés sur les processus fils

[[3]][[2]]
21.1 MB

# version doSleep3
[[4]]
[[4]][[1]]
261 MB     # seul M copié sur les processus fils

[[4]][[2]]
261 MB
```


Cette gymnastique est souvent rebutante...

3.2.2 L'approche `multicore`

Avec **parallel**, nous allons voir qu'il est possible (et préférable, sauf sous Windows) de s'appuyer sur les mécanismes hérités de **multicore**. Dans ce cas, les processus fils sont créés avec l'appel système `fork` qui fait partie des appels système standard d'UNIX (norme POSIX). De fait, cette approche ne fonctionnera pas sous Windows : pas de bug mais pas de parallélisme (par exemple, les `par*apply` se comporteront comme de simples `*apply`, sans parallélisme). Pour créer un pool de processus sur ce modèle, on utilisera l'option `type` :

```
cl <- makeCluster(<taille du pool>, type="FORK")
```

`fork` permet au processus père de créer des processus fils qui sont sa copie conforme, de façon instantanée. Les père et fils partagent donc le même espace mémoire jus-

qu'à ce qu'un fils applique une modification sur un élément et le principe dit de *copie-en-écriture* est appliqué : le système dupliquera l'espace mémoire modifié. Plus besoin de transferts explicites, plus besoin de manipuler les fonctions `clusterExport` et `clusterEvalQ`! En conséquence, cette approche par *forking* sera extrêmement efficace si l'algorithme parallèle suppose la lecture conjointe de grandes structures de données, partagées et non copiées comme avec **snow** (voir Figure 3.2). Elle peut ainsi être utilisée dans un branchement qui vérifie que le système n'est pas Windows :

```
if(Sys.info()[['sysname']] %in% c("Darwin","Linux"){
 # approche multicore
}
```

```
Mem:   8160336k total,  7873028k used,   287308k free,   316108k buffers
Swap:  3905532k total,      160k used,  3905372k free,  1790872k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
31994 vmiele    20   0 1679m 1.6g 1452 R   99 20.0   2:39.66 R
31986 vmiele    20   0 1678m 1.6g 1448 R   99 20.0   2:38.12 R
31985 vmiele    20   0 1678m 1.6g 1452 R   95 20.0   2:38.92 R
31987 vmiele    20   0 1678m 1.6g 1452 R  101 20.0   2:40.18 R
31917 vmiele    20   0 1672m 1.6g 4168 S    0 20.0   0:11.99 R
28989 vmiele    20   0 3141m 361m  29m S    0  4.5   1:29.82 java
12103 vmiele    20   0 1319m 282m  41m S    0  3.5  29:16.51 firefox
23444 vmiele    20   0 1065m 150m  45m S    0  1.9   2:02.49 thunderbird
```

Fig. 3.2 – Visualisation de la mémoire associée aux processus père et fils avec la commande `top` (voir 2.2.1) dans l'approche **multicore** suivant l'initialisation `makeCluster(4, type="FORK")`, lors d'un appel à la fonction `parLapply` . Les processus fils (à `90-100 %CPU`) et le processus père (`0 %CPU`) apparaissent avec chacun 20 % de la mémoire de la machine... or ces 20 % correspondent à la même zone mémoire puisque les processus fils sont des fork du processus père. `top` ne peut montrer cette subtilité... et montre une occupation mémoire totale de 110 % des capacités de la machine... Pas de panique, le principe de copie-en-écriture est appliqué. Ouf!

Remarque
Le système des fork et le principe de copie-en-écriture ne donnent toutefois pas carte blanche au programmeur pour faire n'importe quoi avec la mémoire. Si plusieurs processus fils créent des objets nouveaux en mémoire, leur empreinte mémoire s'additionnera. Si la fonction appelée dans un `lapply` classique utilise 30 % de la mémoire disponible en écriture, on ne pourra utiliser `mclapply` au-delà de trois processus fils, sinon le crash de la machine sera inévitable.

Reprenons l'exemple du k-means parallèle abordé en 3.2.1. Avec l'approche **multicore**, la version parallèle sera d'une simplicité enfantine car les objets `seeds` et

`Boston` sont disponibles pour les fils depuis une zone mémoire accessible en lecture par chacun :

```
cl <- makeCluster(2, type="FORK")
par.res <- parLapply(cl, 1:2, fun=function(i){
 set.seed(seeds[i])
 kmeans(Boston, 4, nstart=1000)
})
```

Les fonctions spécifiques héritées de multicore

Les `par*apply` sont des fonctions « couteaux-suisses » qui encapsulent la gestion des processus fils par divers mécanismes. Il peut être préférable d'utiliser des fonctions plus spécifiques, moins généralistes, mais plus efficaces car optimisées pour un traitement donné. On utilisera donc avantageusement la fonction `mclapply` en remplacement de `parLapply` (qui permet de fonctionner avec les pools de processus de tout type, sur tout système d'exploitation, etc.). Contrairement à `mclapply`, `parLapply` a besoin de réaliser le découpage de la liste en argument et ceci occasionne ainsi un surcoût mémoire. `mclapply` s'utilise comme `lapply` mais il faut ajouter l'option `mc.cores` pour indiquer le nombre de processus fils (qui seront créés par `fork`) ou bien utiliser au préalable la commande `options(cores=...)`. En l'absence de `mc.cores` et `options(cores=...)`, `mclapply` lancera deux processus fils par défaut. Il est également possible de connaître le nombre d'unités de calculs présentes (ce qui ne veut pas dire disponibles...) sur sa machine avec la fonction `detectCores()`. On notera qu'il est donc inutile ici d'utiliser les fonctions de création/suppression de pool `makeCluster/stopCluster`.

```
# options(cores=2) # autre solution
mclapply(1:2, function(i) Sys.getpid(), mc.cores=2)
[[1]]
[1] 28052

[[2]]
[1] 28053
```

Avec `mclapply`, les sorties standards `stdout` et `stderr` seront conjointement utilisées par les processus fils, sans aucune garantie sur la lisibilité finale du fait de conflits potentiels entre fils au moment de l'écriture (on ne peut pas écrire au même endroit en même temps à plusieurs...).
Suivant la même logique, le programmeur expérimenté pourra également remplacer avantageusement `parVapply` par `pvec`, sans garantie de gain conséquent de performance par le parallélisme ; en effet, la plupart des fonctions séquentielles agissant sur les vecteurs sont vectorisées et de fait très performantes. Ceci est mis en lumière par l'exemple suivant :

```
vecteur <- runif(1e7)
options(cores=2)
microbenchmark(
        pvec(vecteur, "^", 1/3, mc.cores=2),
        vecteur^3
        )
Unit: milliseconds
expr             min    lq     mean   median uq      max
pvec(vecteur...) 712.5  714.31 764.07 735.27 751.41  1060.80
vecteur^3        905.57 906.89 927.87 928.80 946.512 949.48
```

On notera qu'il existe aussi une version parallélisée de `Map`, la fonction `mcMap`.

Sur notre exemple de « leave-one-out », l'approche **multicore** simplifie l'utilisation de parLapply : la fonction `leave.one.out` et les objets `iris` ou `my.iris` sont automatiquement disponibles pour les fils, sans aucune recopie en mémoire.

```
if(Sys.info()[['sysname']]=="Windows"){
 cl <- makeCluster(4, type="PSOCK")
 clusterExport(cl, list("leave.one.out"))
} else{ # Linux or Darwin  (cas présent)
 cl <- makeCluster(4, type="FORK")
}
microbenchmark(
 parLapply(cl, 1:150, fun=function(i) leave.one.out(i)),
 times=25
)
Unit: milliseconds
min   lq    mean  median uq    max
94.27 95.72 97.44 96.64  97.93 106.29

stopCluster(cl)
```

Pour plus de simplicité et souvent d'efficacité (en particulier en mémoire), il sera avantageux d'utiliser `mclapply` comme suit :

```
microbenchmark(
 mclapply(1:150, FUN=function(i) leave.one.out(i), mc.cores=4),
 times=25
)
Unit: milliseconds
min    lq     mean   median  uq    max
116.33 117.56 120.14 119.73 120.92 127.17
```

On remarquera que cette dernière version est ici légèrement moins

rapide car la création/destruction du pool de processus fils est intégrée dans `mclapply`. Dès lors, pour une comparaison « *fairplay* », il aurait fallu prendre en compte les temps associés aux opérations `makeCluster/stopCluster`.

Le lancement de tâches en parallèle

L'héritage de **multicore** rend disponible des fonctions permettant de faire du parallélisme de tâches : des calculs sont envoyés au fil de l'eau dans une liste de tâches réalisées par des processus fils. Ainsi, la fonction `mcparallel` crée un processus fils, sans contrôler le nombre d'autres processus fils, et demande la réalisation d'une tâche par ce fils en toile de fond (*i.e.* R rend la main après l'appel à `mcparallel`). Ceci peut donc s'avérer utile si l'on souhaite *recouvrir* un calcul par un autre, c'est-à-dire continuer à travailler avec le processus père pendant qu'un autre calcul, externalisé vers un fils, se réalise. Il est ensuite nécessaire de récupérer un ou tous les calculs externalisés en utilisant la fonction `mccollect`. Cette fonction agit comme une *barrière de synchronisation*. `mccollect` a deux options importantes, `wait` et `timeout` respectivement : si `wait=TRUE`, alors `mccollect` attend la fin de toutes les tâches de la liste donnée en argument ; si `wait=FALSE`, alors `mccollect` attend au maximum `timeout` secondes et donne le résultat des tâches finies à cet instant (sinon retourne `NULL`) comme dans l'exemple suivant :

```
dormir <- function(i) {Sys.sleep(i); i}

p1 <- mcparallel(dormir(5))
p2 <- mcparallel(dormir(10))
mccollect(list(p1,p2), wait=TRUE)
$‘3099‘
[1] 5
$‘3100‘
[1] 10

p1 <- mcparallel(dormir(5))
p2 <- mcparallel(dormir(10))
mccollect(list(p1,p2), wait=FALSE, timeout=6)
[[1]]
[1] 5
```


3.2.3 `foreach + doParallel`

Le package **foreach** propose une implémentation des boucles `for` qui a deux atouts majeurs : 1/ itérer directement sur les éléments d'un ensemble (vecteur, liste, etc.) sans passer par une variable de compteur de boucle et 2/ coupler (ou combiner) le

retour des résultats avec l'exécution d'une fonction tiers (`rbind, c(), max,...`). Ci-après, on itère directement sur `v` (pas besoin de `i` et de `v[i]`) et on combine avec l'addition :

```
v <- c(15,43,63)
foreach(i = v, .combine = "+") %do% { sqrt(i) }
# équivalent de sum(sqrt(v))
```

Mais plus encore, l'intérêt de **foreach** provient de la possibilité de paralléliser le traitement des itérations. Bien sûr, dans ce cas, les itérations doivent être indépendantes. Pour réaliser cette parallélisation, il suffit de réaliser un petit changement syntaxique : remplacer `%do%` par `%dopar%` disponible dans le package **doParallel**. Il faut cependant choisir un mécanisme de parallélisation proposé dans **parallel** (approches `snow` ou `multicore`, comme précédemment) et le mettre en relation avec **foreach** grâce à la fonction `registerDoParallel`. Le programmeur retiendra que **doParallel** est une sur-couche de (ou une interface vers) **parallel** et que, dès lors, les principes fondamentaux (sockets, fork, ...) expliqués en 3.2 seront donc valable dans le cadre de l'utilisation du couple **foreach+doParallel**. Pour l'approche `snow`, on gèrera le pool de processus fils comme suit :

```
cl <- makeCluster(2)
registerDoParallel(cl)
# un ou plusieurs appels à foreach
foreach...%dopar% {...}
stopCluster(cl)
```

La procédure est comparable pour l'approche `multicore` :

```
cl <- makeCluster(2, type="FORK")
registerDoParallel(cl)
# un ou plusieurs appels à foreach
foreach...%dopar% {...}
stopCluster(cl)
```

mais il existe un raccourci avec l'option `cores` :

```
registerDoParallel(cores=2)
foreach...%dopar% {...}
```

La version `foreach` de l'exemple « *leave-one-out* » s'avère intéressante car il est possible de combiner l'exécution des itérations de boucles avec la fonction `sum` qui nous produira directement l'erreur de généralisation :

```
microbenchmark(
 foreach(i=1:150 .combine=sum) %do% {
  leave.one.out(i)
```

```
 },
times=25
)

Unit: milliseconds
min    lq     mean   median  uq    max
430.60 435.22 439.61 437.25 438.45 478.71
```

La version parallèle nécessite la mise en relation entre **foreach** et un pool de processes (ici, suivant l'approche `multicore`) et l'utilisation de l'élément de syntaxe `%dopar%` en lieu et place de `%do%` :

```
library(foreach)
library(doParallel)
registerDoParallel(cores=4)

microbenchmark(
 foreach(i=1:150 .combine=sum) %dopar% {
  leave.one.out(i)
 },
times=25
)

Unit: milliseconds
min      lq     mean median  uq  max
194.29 200.33 210.01 206.41 213.00 283.20
```

La boucle `foreach` parallèle est environ 2.5 fois plus rapide que la version séquentielle avec de 4 unités de calcul. Ces performances sont sensiblement plus faibles qu'avec l'utilisation précédente des `par*apply`, mais ceci provient du fait que la mécanique derrière la mise en œuvre de `foreach+dopar` devient ici non négligeable devant la petitesse du temps de calcul des itérations `leave.one.out(i)`. Sur des exemples concrets de plus grande envergure, les performances seront au rendez-vous.

3.2.4 Génération de nombres pseudo-aléatoires

Dans de nombreux cas, les calculs à paralléliser font intervenir le générateur de nombres pseudo-aléatoires, au travers de nombreuses fonctions comme `sample`, `rnorm`, `runif`, ... C'est bien sûr le cas des implémentations du boostrap, des méthodes Monte-Carlo, des méthodes itératives avec point de départ (k-means par exemple, méthodes EM), etc. Techniquement parlant, le générateur réalise le calcul des éléments d'une suite mathématique qui ont des propriétés proche du hasard.

Cette suite doit cependant être initialisée à partir d'un *graine* (ou *seed*) : il s'agit de l'objet `.Random.seed` (un vecteur d'entier, pour les curieux) que l'on peut initialiser avec la commande `set.seed` comme suit :

```
set.seed(63)
rnorm(1)
1.324112
set.seed(63) # reproductibilité
rnorm(1)
1.324112
rnorm(1)
-1.873185
```

Dans le cas parallèle, chaque processus, père ou fils, doit proposer des séquences de nombres indépendantes... il faut donc garantir que les graines ne sont pas les même sur chaque processus ! Dans l'approche **multicore**, le fork conduit à ce que les processus utilisent le même `.Random.seed` qui est en mémoire partagée. Dans l'approche **snow**, il est possible que `.Random.seed` fasse partie d'un environnement qui a été copié à l'identique. Dans les deux cas, si le code qui s'exécute en parallèle réalise une initialisation de la graine (fréquemment à partir de l'heure courante), la graine sera également identique par chaque processus. Le développeur devra donc veiller à adopter une stratégie alternative.
Une première solution pourrait consister à réaliser tous les tirages aléatoires au niveau du processus père... contraignant. Alternativement, dans l'approche **snow**, on pourrait initialiser chaque `.Random.seed` de chaque processus avec une valeur différente donnée à `set.seed` : c'est le cas dans l'exemple de k-means parallèle en 3.2.1. Mais le package **parallel** propose une solution bien plus robuste : le générateur dit « L'Ecuyer (1999) » est disponible via la commande `RNGkind("L'Ecuyer-CMRG")` et il présente les bonnes propriétés pour la génération en parallèle. Pour s'en servir, la procédure sera la suivante en fonction des approches :

1. ajouter `clusterSetRNGStream(cl)` (cette fonction contient un appel à `RNGkind("L'Ecuyer-CMRG")`) après tout appel à `makeCluster` ;
2. laisser l'option `mc.set.seed=TRUE` (valeur par défaut) lors de l'appel à `mclapply` ou `mcparallel`.

Prenons l'exemple ci-dessous qui permet d'estimer la variabilité des coefficients d'un modèle linéaire par la méthode bootstrap, en parallèle avec **parLapply** :

```
n <- 1000
nb.simu <- 1000
x <- rnorm(n,0,1)
y <- rnorm(n,1+2*x,2)
data <- data.frame(x,y)
cl <- makeCluster(4, type="FORK")
clusterSetRNGStream(cl)
```

```
bootstrap.coef <- parLapply(cl, 1:nb.simu, fun=function(i) {
 bootstrap.resample <- data[sample(n,n,replace=T),]
 lm(y~x,bootstrap.resample)$coef
})
stopCluster(cl)
```

Les fils ont tiré des échantillons bootstrap en parallèle qui ont les bonnes propriétés aléatoires.

3.3 L'équilibrage de charge

La clé de la performance d'un code parallèle R est la répartition équilibrée des tâches à réaliser entre les processus fils, de façon à minimiser les temps d'attente (un processus attend le résultat d'un autre sans réaliser d'opérations de calcul). Imaginons que l'on se met à plusieurs pour repeindre une pièce biscornue : si, au moment du partage des tâches, l'un de nous a hérité d'un mur sensiblement plus grand, celui-ci travaillera alors plus longtemps que les autres qui auront fini bien avant et attendrons (la solidarité entre unités de calcul, ça n'existe pas... en R ; des implémentations de ce concept existent dans d'autres langages et on parlera de *vol de tâches*). On cherchera donc à optimiser l'*équilibrage de charge* (ou *load balancing*) dans le but de :

— minimiser le temps total du calcul réalisé en parallèle ;

— ne pas laisser inactives des unités de calcul (elles ne réalisent pas d'opérations) qui sont dédiées au calcul en cours et donc pas disponibles pour d'autres calculs.

Dans ce but, le programmeur doit prévoir la répartition des tâches entre les processus fils, on parle alors d'*ordonnancement* de tâches.

Supposons par exemple que l'on réalise un calcul sur chaque ligne d'une matrice triangulaire inférieure : si un processus fils s'occupe de la première moitié des lignes (les plus courtes) et l'autre de la seconde moitié (les plus longues), il y aura un évident déséquilibre et le premier processus sera inutilement en attente de la fin des opérations réalisées par le second.

Dans R, cette notion d'équilibrage se rencontre le plus souvent au niveau de la répartition des itérations des boucles dans les versions parallèles des `*apply` (même principe pour `foreach+doParallel`). Dans la suite, on privilégiera le terme *tâche* même s'il s'agit presque exclusivement d'itérations de boucles. Par défaut, les tâches sont réparties de façon *statique*, on parlera alors d'*ordonnancement statique* : chaque processus fils reçoit une liste fixe de tâches/itérations à réaliser. Dans `mclapply`, l'ordonnancement est basé sur la méthode *round-robin* qui consiste à attribuer les itérations de boucles une à une aux processus fils comme lors d'une distribution de cartes à jouer ; *a contrario*, l'ordonnancement dans les `par*apply`

s'appuie sur un découpage en blocs ou *chunks* d'itérations consécutives de même taille (ou presque). Supposons une liste de tâches {A,B,C,D,E,F} à traiter avec deux unités de calcul, `mclapply` les découpera en deux paquets {A,C,E} et {B,D,F} tandis que `par*apply` fera {A,B,C} et {D,E,F}.

Dans l'exemple illustratif suivant qui consiste à faire « dormir » les processus fils, on vérifie que les tâches (à savoir la liste des `temps` d'endormissements) sont alloués suivant la méthode *round-robin* avec les paquets {5,5} et {30,10} :

```
dormir <- function(i) {
        Sys.sleep(i)
        return(paste("le fils",Sys.getpid(),"a dormi",i,"secondes"))
}
temps <- list(5,30,5,10)
mclapply(temps, dormir, mc.cores=2) #  1 par 1
[[1]]
[1] "le fils 3296 a dormi 5 secondes"
[[2]]
[1] "le fils 3297 a dormi 30 secondes"
[[3]]
[1] "le fils 3296 a dormi 5 secondes"
[[4]]
[1] "le fils 3297 a dormi 10 secondes"
```

ou par un découpage en chunks d'itérations consécutives avec les paquets {5,30} et {5,10} :

```
cl <- makeCluster(2)
res <- parLapply(cl, temps, dormir) # par chunks
[[1]]
[1] "le fils 3299 a dormi 5 secondes"
[[2]]
[1] "le fils 3299 a dormi 30 secondes"
[[3]]
[1] "le fils 3308 a dormi 5 secondes"
[[4]]
[1] "le fils 3308 a dormi 10 secondes"
stopCluster(cl)
```

Dans cet exemple, les deux processus fils ont travaillé au total 10 et 40 secondes dans le premier cas avec `mclapply`, 35 et 15 secondes dans le second cas avec `parLapply` : ceci montre clairement un problème d'équilibrage de charge puisque dans les deux cas un processus fils sera en attente pendant 20 secondes... Par ailleurs, on constate que les deux stratégies d'ordonnancement statique (de `mclapply` et `parLapply`) ont conduit à un temps total d'exécution différent, 40 et 35 secondes

respectivement.

Il est donc recommander d'optimiser l'équilibrage de charge et deux cas de figures sont alors à considérer : on connaît ou on ne connaît pas la durée des tâches...

3.3.1 Durée des tâches connue et ordonnancement statique

Dans ce premier cas, les durées des tâches parallèles sont connues. Ces durées peuvent être mesurées par l'expérience (voir 1.2.1) ou estimées par extrapolation de mesures en utilisant la complexité (voir Annexe A.2). A partir de ces durées, on peut alors élaborer un algorithme qui propose un ordonnancement statique efficace. Il existe une importante littérature en informatique fondamentale sur les algorithmes d'ordonnancement (Brucker, 2001) qui permettent de profiter finement de la connaissance des durées pour optimiser le temps total d'exécution parallèle : nous nous bornerons ici à mentionner l'algorithme *Longest processing time first* (LPT, Graham (1969)) qui consiste à utiliser la méthode *round-robin* après avoir classé les tâches par ordre décroissant de durée.

Prenons par exemple le problème du MIS (*Maximum Independant Set*) qui consiste à trouver dans un graphe la plus grande liste de sommets qui ne sont pas connectés entre eux. Les algorithmes actuels ont une complexité exponentielle avec le nombre de sommets d'un graphe. On gagnera à appliquer l'algorithme LPT en traitant d'abord les grands graphes. On génère dans un premier temps une liste de graphes de taille variable :

```
library(igraph)
mis <- maximal.independent.vertex.sets
make.graph <- function(nb.vertices){
        erdos.renyi.game(nb.vertices, 3/nb.vertices)
}
graph.list <- lapply(c(20,20,25,40,30,40), make.graph)
```

La version `mclapply` nécessite de donner l'ordre des graphes par taille décroissante dans `order.LPT`, de façon à avoir un gain substantiel de temps de calcul global grâce au LPT :

```
system.time( mclapply(graph.list, mis, mc.cores=2) )[3]
9.233

order.LPT <- order(unlist(lapply(graph.list, vcount)), decreasing=T)
system.time( mclapply(graph.list[order.LPT], mis, mc.cores=2) )[3]
5.894
```

Pour la version `parLapply`, il faut manuellement créer des *chunks* qui prennent alternativement les graphes du plus grand au plus petit pour obtenir un meilleur ordonnancement avec l'algorithme LPT :

```
cl <- makeCluster(2, type='PSOCK')
system.time( parLapply(cl, graph.list, mis) )[3]
10.040

order.LPT <- order(unlist(lapply(graph.list, vcount)), decreasing=T)
order.LPT.bychunk <- c(order.LPT[c(1,3,5)], order.LPT[c(2,4,6)])
// le plus grand dans le 1er chunk, le suivant dans le 2e
// puis le suivant dans le 1er, etc...
system.time( parLapply(cl, graph.list[order.LPT.bychunk], mis) )[3]
6.096

stopCluster(cl)
```

La question de l'ordonnancement est ici prépondérante et conduit, quand le programmeur se questionne, à des gains de performance non négligeables.

3.3.2 Durée des tâches inconnue et ordonnancement dynamique

Dans le second cas, on ne connaît pas les durées des tâches. Cela peut être le cas avec des algorithmes itératifs qui s'arrêtent quand une condition est satisfaite au bout d'un plus ou moins grand nombre d'itérations selon les données, ou bien les algorithmes stochastiques qui convergent avec une vitesse aléatoire. Dans ce cas, il est envisageable de réaliser un *ordonnancement dynamique*. Le principe est simple : le père distribue les tâches une à une (ou par blocs de tâches) en fonction des disponibilités au niveau des processus fils et le calcul parallèle s'arrête quand la liste de tâches restantes est vide.
L'ordonnancement dynamique est disponible dans `mclapply` en désactivant l'option `mc.preschedule` (mettre à `FALSE`), les processus fils étant créés par `fork` et détruits pour chaque tâche à réaliser. Ceci peut occasionner un *overhead* si le temps de calcul des tâches est petit, le coût de création/suppression des processus fils devenant alors non négligeable.

Pour notre exemple illustratif des processus fils que l'on fait dormir, on remarque que le temps total passe à 30 secondes avec l'ordonnancement dynamique, ce qui est meilleur que la version statique :

```
temps <- list(5,30,5,10)
system.time(
  res <- mclapply(temps, dormir, mc.cores=2, mc.preschedule = FALSE)
 )[3]
30.040

res
[[1]]
```

```
[1] "le fils 12757 a dormi 5 secondes"

[[2]]
[1] "le fils 12758 a dormi 30 secondes"

[[3]]
[1] "le fils 12759 a dormi 5 secondes"

[[4]]
[1] "le fils 12760 a dormi 10 secondes"
```

Par ailleurs, l'ordonnancement dynamique est en théorie disponible dans les versions `par*applyLB` (`LB` pour « *load balancing* ») des `par*apply`... en théorie seulement car, jusqu'à `R3.2` au moins, l'implémentation de celles-ci met en œuvre un ordonnancement statique ! Il reste au programmeur une solution acrobatique : utiliser la fonction `clusterApplyLB`, la version dynamique de `clusterApply`, fonction similaire à `parLapply` comme son nom ne l'indique pas (elle prend un vecteur ou une liste en argument, pas une matrice). `clusterApply` envoie les tâches une à une suivant une approche *round-robin* contrairement à `parLapply` qui fonctionne avec des *chunks* de tâches (et donc limite les communications). Mais attention, à chaque envoi de tâche est associé l'envoi des données associées, même si elles étaient déjà utilisées pour les tâches précédentes : ceci peut très vite conduire à une fort *overhead* de communication. Cette fonction n'est donc pas privilégiée en mode statique (c'est pour cette raison qu'elle n'a pas été présentée avant) mais sa version dynamique `clusterApplyLB` peut s'avérer utile dans certains cas. Ici, le pool de processus fils est créé une fois pour toute avant la distribution des tâches.

```
temps <- list(30,5,5,10)
temps <- list(30,5,5,10)
cl <- makeCluster(2)
system.time( res <- clusterApplyLB(cl, temps, dormir) )[3]
30.004

stopCluster(cl)
res
[[1]]
[1] "le fils 14600 a dormi 30 secondes"

[[2]]
[1] "le fils 14609 a dormi 5 secondes"

[[3]]
[1] "le fils 14609 a dormi 5 secondes"

[[4]]
```

```
[1] "le fils 14609 a dormi 10 secondes"
```

On notera que McCallum & Weston (2011) propose une implémentation de `parLapply` qui réalise un ordonnancement dynamique.

Supposons maintenant que l'on souhaite réaliser l'approche « *leave-one-out* » sur plusieurs jeux de données en parallèle. Il est naturel d'envisager un parallélisme gros-grain (voir 2.3.1 pour la définition de cette terminologie) qui consiste à exécuter sur chaque jeu de données la version séquentielle de la fonction de calcul de l'erreur PRESS suivante :

```
compute.PRESS <- function(dataset){
 Reduce("+", lapply(1:nrow(dataset),
                    FUN=function(i) leave.one.out(i, dataset)))
}
```

Pour illustrer l'approche, on simule des jeux de données en nombre et de tailles différentes (indiquées dans le vecteur `sizes`)

```
simuData <- function(sizes){
 model <- lm(Petal.Width ~ Petal.Length, data=iris)
 lapply(sizes, FUN=function(n){ # n nombre de points
  a <- min(iris[,"Petal.Length"])
  b <- max(iris[,"Petal.Length"])
  iris2 <- data.frame(Petal.Width=rep(NA,n),
                      Petal.Length=runif(n,a,b))
  iris2[,"Petal.Width"] <-
   predict(model,data.frame(Petal.Length=iris2[,"Petal.Length"])
   +rnorm(n,0,0.1))
  iris2
 })
}
```

La parallélisation est réalisée avec `mclapply` (même idée avec `parLapply`) où les itérations correspondent aux différents jeux de données. Alors, ordonnancement statique ou dynamique (avec `mc.preschedule=F`) ? Prenons la situation idéale où la taille des données (200 points) est la même et le nombre de jeux de données (8) est un multiple du nombre d'unités de calcul demandées (4) :

```
multiple.iris <- simuData(rep(200,8))
microbenchmark(
    mclapply(multiple.iris,
             FUN=function(dataset) compute.PRESS(dataset),
             mc.cores=4),
    mclapply(multiple.iris,
```

```
            FUN=function(dataset) compute.PRESS(dataset),
            mc.cores=4, mc.preschedule=F),
    times=10
)
Unit: milliseconds
min     lq      mean    median    uq    max
 974.35  977.01  979.22  978.53  981.78  986.18
1010.32 1012.27 1015.37 1013.92 1016.18 1028.72
```

Dans ce cas, on voit que les approches statiques et dynamiques sont équivalentes, avec un léger *overhead* dans le cas dynamique lié à la création/destruction des fils pour chaque tâche (*i.e.* chaque jeu de données).

Prenons maintenant le cas de jeu de données de tailles hétérogènes, par hasard mal réparties si on les prend suivant un round-robin, et dont le nombre (10) n'est pas multiple du nombre d'unités de calcul (4) :

```
multiple.iris <-
   simuData(c(1000,200,1000,200,1000,200,250,100,100,100))
# même microbenchmark
Unit: seconds
min  lq   mean median uq max
6.13 6.13 6.15 6.14 6.16 6.18
3.53 3.53 3.60 3.54 3.55 4.16
```

Dans ce cas mal équilibré, la version dynamique est très sensiblement plus performante que l'ordonnancement statique naïf. Dans le cas statique, certaines unités de calcul seront inactives pendant des intervalles de temps non négligeables.

3.3.3 Granularité et équilibrage de charge

Si l'on revient au but de l'équilibrage de charge comme énoncé en début de 3.3, on comprend qu'il ne faut pas laisser des unités de calcul inactives. Il faut donc avoir assez de travail à distribuer. Prenons le cas extrême de trois tâches à traiter avec deux unités de calcul : une unité de calcul sera inactive pendant que la troisième tâche est traitée. Comment remédier à ce problème ? Modifier la granularité (voir 2.3.1), *i.e.* modifier le contour des tâches à paralléliser. Voici un exemple concret en pseudo-code où l'on estime des modèles de plusieurs tailles (pour réaliser une sélection de modèle), mais la procédure d'estimation contient une boucle interne (par exemple, on réalise une approche stochastique avec 200 points de départ) :

```
model.estimation <- function(size){
 lapply(1:200, FUN=...{ # boucle 1
```

```
  // etc...
 })
 // etc...
}
models <- lapply(1:10, model.estimation) # boucle 2
```

Quelle `lapply` faudra-t-il paralléliser ? Grain-fin (boucle 1) ou gros-grain (boucle 2) ? Il n'y a pas ici de réponse toute faite... Les deux ? Surement pas, sous peine d'aller à la catastrophe car chaque processus fils deviendrait père d'autres processus fils, et les nombres d'unités de calcul demandés avec `mc.cores` se multiplieraient. Le programmeur doit juger en fonction de chaque cas concret. La granularité fine est souvent plus efficace mais en général 1/ plus intrusive que la parallélisation gros-grain (*i.e.* elle demande plus d'efforts de programmation et de modification de code) et 2/ peut occasionner un *overhead* si le coût de la mécanique derrière la parallélisme est non négligeable devant le temps de calcul. Dès lors, c'est le rapport coût/bénéfice pour le programmeur qui devra être étudié avant de choisir une implémentation.

Reprenons notre exemple sur plusieurs jeux de données. Il est possible d'envisager un parallélisme grain-fin qui reprend l'implémentation du « leave-one-out » parallèle vu en 3.2 :

```
parcompute.PRESS <- function(dataset){
    Reduce("+", mclapply(1:nrow(dataset),
                FUN=function(i) leave.one.out(i, dataset),
                mc.cores=4))
}
```

On peut donc évaluer les performances des deux approches 1/ grain-fin qui consiste à boucler séquentiellement sur les jeux de données en utilisant le « *leave-one-out* » parallèle et 2/ gros-grain (abordée dans la section précédente) qui revient à paralléliser la boucle sur les jeux de données en utilisant le « *leave-one-out* » séquentiel.

Dans la situation idéale précédemment décrite, le gros-grain est légèrement plus performant car il limite l'*overhead* de la mécanique parallèle (oui, il n'y a qu'un seul `mclapply` appelé contre 8 pour le grain-fin) :

```
multiple.iris <- simuData(rep(200,8))
microbenchmark(
    lapply(multiple.iris,   # grain-fin
           FUN=function(dataset) parcompute.PRESS(dataset)),
    mclapply(multiple.iris, # gros-grain
             FUN=function(dataset) compute.PRESS(dataset),
```

```
            mc.cores=4, mc.preschedule=F),
    times=10)
Unit: milliseconds
min     lq      mean    median  uq      max
1233.55 1240.96 1246.12 1245.11 1250.25 1265.74
980.57  985.86  1012.59 1008.19 1017.99 1095.18
```

Dans le cas où la question de l'ordonnancement n'est pas triviale au niveau gros-grain, alors l'approche grain-fin sera plus efficace. C'est le cas sur notre exemple de données de tailles hétérogènes :

```
multiple.iris <-
   simuData(c(1000,200,1000,200,1000,200,250,100,100,100))
# même microbenchmark
Unit: seconds
min  lq   mean median uq  max
3.32 3.33 3.37 3.35 3.37 3.56
3.55 3.55 3.56 3.55 3.56 3.62
```

Remarque
Supposons de plus que l'on ait plus d'unités de calcul disponibles que de jeux de données dans `multiple.iris`, alors il est évident qu'il faudra choisir l'approche grain-fin

Nous mentionnons également le cas des boucles imbriquées `foreach` (*nested loops*, boucle `foreach` qui en contient une autre, qui peut en contenir une autre également...). La même question sur la granularité peut se poser : quelle boucle paralléliser avec `%dopar%` ? Le package apporte une solution avec l'opérateur `%:%` qui, de fait, revient à aplatir l'imbrication des boucles en transformant les boucles multiples en boucle unique et c'est cette boucle qui subira la parallélisation. L'exemple de la procédure d'estimation se décline avec le pseudo-code suivant :

```
foreach(q=1:10) %:%
 foreach(i=1:200) %dopar % {
  // etc...
 }
```

La question de la granularité restera un challenge intellectuel important pour le programmeur : « *The free lunch is over* », de nouveau...

Chapitre 4

Calcul parallèle dans C++ interfacé avec R sur machine multi-cœurs

Comme dans le chapitre précédent, nous nous plaçons de nouveau dans le cadre d'architectures matérielles constituées par une seule machine, composée d'un ou plusieurs processeurs, eux-même intégrant plusieurs cœurs de calcul.

4.1 Principe général

Dans un code R qui fait appel à un langage compilé au travers d'une interface (on se bornera ici à C++ et à l'utilisation de Rcpp), le parallélisme peut se trouver au niveau du code R et au niveau des parties implémentées en langage compilé, ou bien aux deux niveaux. Avant de présenter comment on introduit du parallélisme en *mémoire partagée* au niveau de C++, il est souhaitable de parler du schéma à adopter pour bien choisir le grain du parallélisme (voir 2.3.1) et maximiser le gain théorique de performance (voir 2.4). Prenons des exemples significatifs :

— le code est constitué de plusieurs blocs qui se suivent, les uns en pur R et les autres en C++. Pour minimiser la partie séquentielle, il faudra implémenter le parallélisme à la fois au niveau de R et au niveau de C++ ;

— R est juste utilisé comme interface utilisateur encapsulant du code compilé qui réalise la (quasi-)totalité des calculs (c'est le cas de la fonction `hclust` par exemple, ou de nombreuses fonctionnalités disponibles dans BioConductor `https://www.bioconductor.org/`). C'est donc la partie en C++ qui devra être parallélisée ;

— R et C++ sont emboités, il faudra alors s'interroger sur le grain à adopter : parallélisme gros grain en R ou grain fin en C++ ? Mais surtout pas les deux,

sous peine de subir l'effet multiplicatif des parallélisme et de faire couler la machine de calcul (disons 4 unités de calcul sollicités par R mais 4 également par C++, cela fait $4 \times 4 = 16$ au total!!). Prenons le pseudo-code suivant :

```
# code R
for (q in 1:Q){
 fonction_Rcpp(q, params)
}

// code C++
fonction_Rcpp(params){
 for (int i=0; i<N; i++){
  ... // plein de calcul
 }
}
```

Si Q est grand, à quoi bon se compliquer la vie et introduire du parallélisme dans C++ ? On se bornera à paralléliser les itérations de la première boucle enR. Ce sera un cas de parallélisme embarrassant (voir 2.3.1). Si Q est petit, on préférera introduire du parallélisme dans C++, suivant les même arguments que ceux abordés en 3.3.3.

Cette fois-ci, l'introduction du parallélisme au niveau des parties implémentées en langage compilé va conduire à l'utilisation de plusieurs threads et non pas plusieurs processus indépendants comme dans le chapitre précédent.

Le terme *thread* a déjà été défini en 2.2.1 : il s'agit de processus légers (on parle aussi de fil d'exécution) initiés par un processus (dit *lourd*), qui partagent une grande partie des ressources de ce processus maître, notamment la mémoire.
Les *threads* utilisant un espace mémoire commun, il est important de s'assurer de l'indépendance des calculs réalisés en parallèle. Si deux *threads* doivent accéder à la même donnée en mémoire, on parle alors de *concurrence*. L'accès concurrent aux données est une des difficultés de la programmation en mémoire partagée, et conduit assez régulièrement à des bugs souvent difficile à détecter... il faut donc être particulièrement vigilant.
Du point de vue de la programmation, le *thread* correspond à un ensemble d'instructions indépendantes qui peuvent se dérouler en parallèle. Il existe plusieurs façons de programmer les *threads* que nous allons détailler dans la suite :

— l'utilisation des directives openMP (option la plus courante) ;

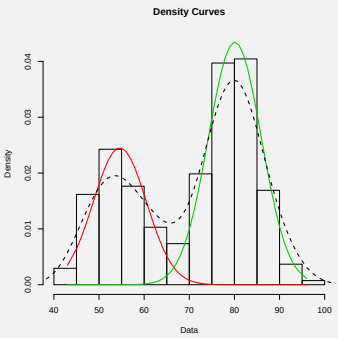
— le support du *multithreading* de la norme C++11 ;

— la bibliothèque C++ Intel Threading Building Blocks ;

— (il est aussi possible d'utiliser d'autres bibliothèques comme les implémentations de la norme Posix (les *Pthreads*), mais nous n'en parlerons pas ici).

Supposons que l'on s'intéresse au modèle de mélange gaussien qui consiste à supposer que des données proviennent d'une source contenant plusieurs sous-échantillons gaussiens.

Ci-contre un exemple du package **mixtools** montrant l'histogramme des données, les densités de deux gaussiennes (trait plein) et la densité du mélange (pointillé). L'objectif du programmeur sera d'implémenter un algorithme de classification « non supervisé » qui vise à calculer la probabilité que chaque donnée appartienne à chaque sous-échantillon.



Un choix classique consiste à implémenter l'algorithme itératif *EM* (pour *Expectation-Maximization*) qui, sans rentrer dans les détails, demande un grand nombre d'évaluation de la fonction de densité gaussienne (pour chaque itération×donnée×gaussienne). Il faut donc trouver une approche efficace pour implémenter ces calculs de densité, possiblement en parallèle...

Dans ce qui suit, nous allons détailler les différentes alternatives, en réalisant des tests sur une architecture `Linux` (certains détails syntaxiques peuvent être différents sur les autres types de système). Supposons que `x` représente les données. On utilisera dans un premier temps la fonction vectorisée `dnorm` de R :

```
x <- runif(1e6)
mu <- 1
sigma <- 0.2
microbenchmark(
 dnorm(x,mu,sigma),
 times=10
)
Unit: milliseconds
min   lq    mean  median  uq  max
87.97 88.27 88.82 88.93 89.33 89.43
```

Vu la taille de `x`, on peut espérer gagner en performance en tirant profit du parallélisme. Comment s'y prendre ?

Un première idée pourrait consister à paralléliser les itérations sur tous les éléments de `x` avec `mclapply` :

```
microbenchmark(
 mclapply(x, FUN=function(xi) dnorm(xi,mu,sigma), mc.cores=4),
 times=10
)
Unit: seconds
min  lq   mean median uq max
1.01 1.11 1.19 1.21 1.24 1.37
# attention, temps en secondes, non plus en millisecondes
```

Ceci donne des performances catastrophiques car on perd ici la force de la vectorisation de `dnorm` (appelée $1e6$ fois dans ce cas !).

Une solution pourrait consister à découper `x` en autant de sous-vecteur que de processus fils et à dispatcher les calculs sur chaque sous-vecteur avec `mclapply` (ou un `par*apply`) :

```
microbenchmark(
 unlist(mclapply(1:4, FUN=function(i){
  l <- length(x)/4
  a <- (i-1)*l+1
  b <- i*l
  dnorm(x[a:b],mu,sigma)
 }, mc.cores=4)),
 times=10
)
Unit: milliseconds
min   lq    mean  median  uq  max
62.44 65.69 72.11 68.21 69.76 116.01
```

Les performances sont médiocres ($\times 1.3$ avec 4 unités de calcul) car la création des sous-vecteurs en mémoire prend du temps...

Reste alors l'idée de tirer profit de `C++`. On peut ainsi proposer une implémentation `Rcpp` de `dnorm` (dans ce qui suit, nous utilisons `Rcpp` et le package **inline** pour déclarer des fonctions R interfacée avec `C++`, voir en Annexe A.5 et Eddelbuettel (2013) pour plus de détails) :

```
cppFunction('NumericVector dnormC(NumericVector x,
                                  double mu,
                                  double sigma){
 size_t n = x.size();
```

```
 double sqrt2pi = sqrt(2*M_PI);
 Rcpp::NumericVector density(n);
 for (size_t i=0; i<n; i++)
  density[i] = 1/(sigma*sqrt2pi)
       *exp(-0.5*(pow((x[i]-mu)/sigma,2)));
  return density;
 }
',includes=c("#include<cmath>"))

microbenchmark(
 dnormC(x,mu,sigma),
 times=10
)
Unit: milliseconds
min   lq    mean  median  uq  max
33.51 33.53 33.82 33.75 33.98 34.40
```

La version `C++ dnormC` est déjà plus rapide que `dnorm` ! Il paraîtrait naturel de reprendre l'approche précédente avec un `mclapply` qui utilise `dnormC` pour introduire du parallélisme, mais de nouveau les performances ne sont pas au rendez-vous (coût de la création des sous-vecteurs), pire la version parallèle est plus lente que la séquentielle :

```
microbenchmark(
 unlist(mclapply(1:4, FUN=function(i){
  l <- length(x)/4
  a <- (i-1)*l+1
  b <- i*l
  dnormC(x[a:b],mu,sigma)
 }, mc.cores=4)),
 times=10
)
Unit: milliseconds
min   lq    mean  median uq   max
52.59 53.62 67.16 55.03 58.68 171.40
```

Que faire alors ? Introduire du parallélisme en *mémoire partagée* au niveau de `C++`.

4.2 openMP

4.2.1 Les bases d'openMP

`OpenMP` est un standard de programmation faisant suite à de nombreux « dialectes » mis au point par les constructeurs (de machines), permettant d'implémenter le parallélisme en mémoire partagée. Les langages supportés sont `Fortran` et `C/C++`. Depuis la fin des années 90, `OpenMP` est incontournable et supporté par la grande majorité des compilateurs (voir OpenMP-ARB (2016)). Son principe est simple : mettre à disposition des développeurs un ensemble de directives (*i.e.* des mots clés) pour aider le compilateur à paralléliser tout seul un programme. Saupoudrer notre code `C++` de directives `OpenMP`, voilà le programme ! Ce jeu de directives de compilation est standard mais il est vu comme des commentaires par les compilateurs qui ne supportent pas `openMP` : un programme contenant de l'`openMP` est aussi un programme séquentiel valide, il est donc *portable* (accepté par de nombreux environnements machine). La syntaxe sera la suivante :

```
#pragma omp directive-name [clause[ [,] clause]...]
```

Pour que cette syntaxe se transforme en directive pour le compilateur, il faudra ajouter une option de compilation (`-fopenmp` pour `GCC et Clang`).

L'entité centrale d'un programme avec `openMP` n'est plus le processus mais le *thread*. Le *thread master* (identifiant = 0) est immédiatement actif et dans des *régions parallèles* une équipe de *m threads* exécute les instructions en parallèle (voir Figure 4.1).

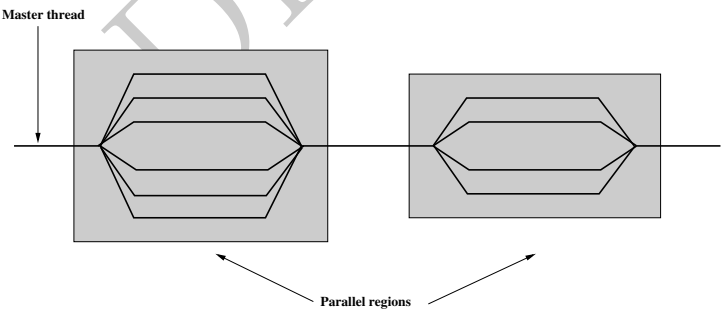


Fig. 4.1 – Illustration des régions parallèles avec `openMP`.

Dans un code `R` qui appelle du `C++` parallélisé avec `openMP`, il n'y aura donc que le processus père (pas de fils) qui mettra en place les différents *threads* : il n'y aura donc qu'un processus `R` qui tourne, mais son activité pourra monter jusqu'à $m \times 100\%$ (voir Figure 4.2).

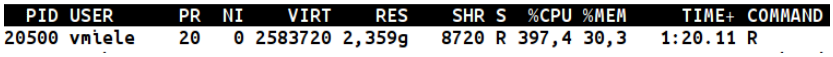


Fig. 4.2 – Visualisation du pourcentage d'activité du processus R avec la commande `top` (voir 2.2.1) lors de l'utilisation d'`openMP` avec quatre threads. Son activité pourra monter jusqu'à $4 \times 100\%$.

Choix du nombre de threads . Pour sélectionner le nombre de *threads*, on renseigne la variable d'environnement suivante (ici sous Linux) :

```
export OMP_NUM_THREADS=10
```

ou bien on utilise des fonctions *run-time* (disponible dans le header `omp.h`) dont l'utilisation n'est pas possible par le code séquentiel. Il faut donc les utiliser dans un bloc entouré des directives `#ifdef _OPENMP` comme suit :

```
#ifdef _OPENMP
#include <omp.h>
#endif
...
...
#ifdef _OPENMP
omp_set_num_threads(10)
#endif;
```

Principales directives . Le programmeur demandera l'ouverture de régions parallèles avec la directive `parallel` :

```
#pragma omp parallel
{
do_something_in_parallel(...);
}
```

Compte tenu de l'architecture en mémoire partagée, le travail en parallèle s'effectue sur des variables communes (dites *partagées*, par défaut) ou privées (si on le demande) à chaque *thread*. En `C++`, les variables déclarées en dehors des régions parallèles sont communes à tous les *threads* (`shared` par défaut), mais les variables déclarées à l'intérieur sont privées (`private`) pour chaque *thread* :

```
int A[10], B[10];
#pragma omp parallel
{
int bound = omp_get_thread_num()*5; // bound est private
for (int i=bound; i<bound+5; i++)
 A[i] = B[i]*B[i];                  // A et B partagés en lecture
```

```
                                        // et en écriture
}
```

Par ailleurs, on peut organiser le partage ou la privatisation des variables : les clauses `shared, private, firstprivate` permettent respectivement de partager, de privatiser ou de copier en privatisant vers des variables privées, comme suit :

```
int i = 100;
#pragma omp parallel firstprivate(i)
{
i = i + omp_get_thread_num(); // i = 100 ou 101 ou 102...
                              // selon les threads
}
// ici i vaut toujours 100
```

Dans beaucoup de codes de calcul en `C++`, les boucles sont omniprésentes. Si et seulement si les itérations sont indépendantes, elles sont candidates pour la parallélisation. Les itérations sont distribuées entre les *threads* (ci-dessous, chaque *thread* a un jeu de valeurs `i`) et le *thread master* reprend la main quand tous les threads ont fini leur itérations :

```
#pragma omp parallel
{
#pragma omp for
for (int i = 1; i <10; i++)
   A[i] = B[i]*A[i];
// A[i] = B[i]*A[i-1]; impossible ! pas indépendance des itérations
}
```

On notera qu'il est possible de combiner les deux directives sous la forme `#pragma omp parallel for`.

Le programmeur ne peut maîtriser la façon dont une variable `shared` est utilisée en écriture. L'exemple suivant donnera un résultat différent à chaque exécution :

```
int sum = 0
#pragma omp parallel
{
#pragma omp for
for (int i = 0; i <1000000; i++)
   sum += i;  // CATASTROPHE
}
```

Et `firstprivate(sum)` ne résout pas le problème car `sum` vaudra 0 après la fin de la boucle... Les *sections critiques* résolvent le problème : un *thread* après l'autre

exécute une partie du code (ordre inconnu). Mais attention : les *threads* risquent d'attendre leur tour... et les performances peuvent s'effondrer. L'exemple précédent est donc modifié par l'ajout d'une section critique qui permettra de sommer dans `sum` les variables privées `tsum` sans problème d'accès concurrent :

```
int sum = 0, tsum = 0;
#pragma omp parallel firstprivate(tsum)
{
#pragma omp for
for (int i = ; i <10; i++){
        tsum += i;
}
#pragma omp critical
{
        sum += tsum
}
}
```

Il est également possible d'assembler les contributions de chaque *thread* au moyen d'un opérateur (somme, produit, etc.) : on parle alors d'opération de `reduction`. Une version optimale de notre exemple contiendra une réduction de type somme, de sorte que la variable `sum` sera la somme des `sum` privées de chaque *thread* :

```
int sum = 0;
#pragma omp parallel reduction(+:sum)
{
#pragma omp for
for (int i = ; i <10; i++)
        sum += i;
}
}
```


L'ordonnancement des itérations de boucle est configurable (le principe sous-jacent est le même que pour R, voir 3.3). Le plus simple reviendra à diviser les itérations de boucles en *chunks* contigus de taille donnée. On parlera de nouveau ici d'ordonnancement statique et la syntaxe sera la suivante :

```
#pragma omp parallel for schedule(static)
```

Il est également possible d'opter pour un ordonnancement dynamique, où les *chunks* sont assignés dynamiquement à chaque *thread* disponible :

```
#pragma omp parallel for schedule(dynamic)
```

Il faudra par contre bien choisir la taille des *chunks* : petite (1 en particulier), elle conduira à un risque d'*overhead* (accès concurrent à la liste des itérations à

réaliser) ; grande, elle pourra créer un problème d'équilibrage de charge.

Le développeur R souhaitant approfondir l'utilisation d'openMP pourra lire l'excellent ouvrage Hager & Wellein (2011).

4.2.2 R et openMP

A l'heure de tirer profit d'openMP dansR, il est important de mettre en garde le développeur sur le fait que la configuration du compilateur, d'openMP et de R dépendent de la machine cible et de son système d'exploitation : une consultation éclairée d'internet (R-help, stackoverflow, etc.) doit permettre à chacun de trouver une solution adaptée à son cas. On notera toutefois que les systèmes Linux sont plus faciles à gérer sur cette question.
La solution la plus aisée pour gérer à la fois l'interfaçage entre R et C++ et l'utilisation d'openMP est de développer un package R. Pour cela, les différentes étapes sont expliquées dans le document de référence RCoreTeam (2016) qui contient la section « OpenMP support ». Tout est en effet prévu pour pouvoir préciser l'option de compilation nécessaire au support d'openMP. En particulier, sous Linux, il faut modifier le fichier src/Makevars en ajoutant :

```
PKG_CXXFLAGS=$(SHLIB_OPENMP_CFLAGS)
PKG_LIBS=$(SHLIB_OPENMP_CFLAGS)
```

ou, dans le cas de l'utilisation conjointe de Rcpp :

```
PKG_CXXFLAGS=-I. `${R_HOME}/bin/Rscript -e "Rcpp:::CxxFlags()"`
        $(SHLIB_OPENMP_CFLAGS)
PKG_LIBS=`${R_HOME}/bin/Rscript -e "Rcpp:::LdFlags()"`
        $(SHLIB_OPENMP_CFLAGS)
```

Pour notre exemple du mélange gaussien, nous ne construisons pas un package mais nous utilisons le package **inline** pour déclarer une fonction R similaire à dnormC mais parsemée de directives openMP. Pour que celles-ci soient considérées par le compilateur (et pas vue comme des commentaires) et que la variable d'environnement _OPENMP soit définie, il faut ajouter la bonne option de compilation comme suit :

```
Sys.setenv("PKG_CXXFLAGS"="-fopenmp")
Sys.setenv("PKG_LIBS"="-fopenmp")
```

On propose alors la version openMP suivante :

```
cppFunction('NumericVector dnormC_omp(NumericVector x,
                                      double mu,
                                      double sigma){
 size_t n = x.size();
```

```
 double sqrt2pi = sqrt(2*M_PI);
 Rcpp::NumericVector density(n);
#ifdef _OPENMP
 omp_set_num_threads(4);
#endif
#pragma omp parallel for
 for (size_t i=0; i<n; i++)
  density[i] = 1/(sigma*sqrt2pi)
       *exp(-0.5*(pow((x[i]-mu)/sigma,2)));
  return density;
}
',includes=c("#include<cmath>","#include <omp.h>"))
```

On notera que le *header* `omp.h` a été ajouté dans la liste des `includes` de façon à ce que la fonction `omp_set_num_threads` soit connue.

Cette version avec `openMP` développe de très bonnes performances, avec une accélération $\times 3$ avec 4 unités de calcul par rapport à `dnormC` et $\times 8$ par rapport à `dnorm` :

```
microbenchmark(
 dnormC_omp(x,mu,sigma),
 times=10
)
Unit: milliseconds
min   lq    mean  median  uq  max
11.37 11.39 11.73 11.66 12.03 12.23
```

Il est par ailleurs possible de modifier la fonction en ajoutant un paramètre permettant de choisir le nombre de *threads*, comme suit :

```
cppFunction('NumericVector dnormC_omp(int nbthreads,...
 ...
 omp_set_num_threads(nbthreads);
 ...
```

Remarque
Avec le compilateur `clang`, il peut s'avérer nécessaire d'utiliser `-fopenmp -lgomp`.

4.3 Les threads de C++11

4.3.1 Les bases des threads de C++11

En 2011, le comité du Standard C++ a approuvé une nouvelle norme appelée C++11 (faisant suite à C++98). Celle-ci introduit dans C++ un ensemble de nouveautés au niveau du langage lui-même et surtout de ses librairies. Les compilateurs ont progressivement supportés cette norme et il est désormais possible d'utiliser les fonctionnalités C++11 dans la plupart des codes, en particulier dans le code C++ appelé par R. Le document de référence RCoreTeam (2016) évoque d'ailleurs cette alternative dans le chapitre « Using C++11 code ».
Une des grandes nouveautés de la librairie standard C++11 est l'introduction du support natif du *multithreading* (*i.e.* le parallélisme à base de *threads*, comme précédemment). Créer des *threads*, passer des fonctions et des arguments, récupérer des résultats, se synchroniser entre *threads* : tout est désormais possible avec la librairie `<thread>`. Néanmoins, la maîtrise de cette librairie et son utilisation sans risque demande une connaissance importante à la fois du C++ et des mécanismes de *threads*. Si la parallélisation avec openMP reste assez facile d'accès, les *threads* de C++11 sont à réserver à un public averti. Dans cette ouvrage, nous nous bornerons à présenter les interfaces de plus haut niveau `std::async()` et `std::future<>` (les interfaces plus bas niveau `std::thread, std::promise, std::mutex,...` pourront être découvertes lors d'une lecture appropriée de Josuttis (2012)) :

— `std::async()` permet de créer un objet qui contient une fonctionnalité à réaliser (une fonction à appeler, le plus souvent) en toile de fond par un *thread* ;

— `std::future<>` permet d'organiser la récupération des résultats obtenus par les *threads*.

La logique voudra que l'on associe ces deux interfaces de la façon suivante :

```
std::future< type-of-result > result(
                async(any-fonction-name, some-parameters)
                );
```

avec

— `type-of-result` qui est le type de retour de la fonction `any-fonction-name`. Les afficionados de C++11 pourront remplacer `std::future< type-of-result >` par `auto`, mot clé qui laisse le pré-compilateur deviner le type de la variable considérée.

— `some-parameters` qui représente les variables en paramètre de la fonction. Toutes ces variables seront passées par copie (pas d'alternative possible). Il est toutefois possible de contourner le passage par copie en passant un pointeur ou un objet référence (`std::ref`). Ceci sera obligatoire si l'on souhaite que `any-fonction-name` modifie une variable existante : en effet, les *threads* ne « voient » pas les variables de l'extérieur de la fonction (pas de variable

`shared` au sens d'`openMP` par exemple) et il faudra les passer par adresse (pointeur) ou par référence.

Cependant, `async()` ne garantit pas que le calcul sera réalisé par un *thread*, ni qu'il est fini. On utilisera donc la fonction `get` pour se synchroniser sur l'arrivée des résultats :

```
result.get();
```

A noter enfin que, dans de nombreuses situations classiques, le développeur ajoutera un premier paramètre à `async` pour demander le départ immédiat du calcul par le *thread* : `std::async(std::launch:async,...)`.

4.3.2 R et les threads de C++11

La principale difficulté réside dans la nécessité de demander au compilateur de supporter le standard `C++11`. La solution la plus aisée pour gérer l'interfaçage entre R et `C++11` est de nouveau de développer un packageR. Pour cela, il suffira d'ajouter dans le fichier `src/Makevars(.win)` :

```
CXX_STD=CXX11
```

et modifier le fichier `DESCRIPTION` en ajoutant `SystemRequirements: C++11`.

Nous reprenons notre exemple du mélange gaussien et de nouveau nous utilisons le package **inline** pour créer une fonction qui encapsule du `C++11`. Il faut ajouter l'option de compilation adéquate :

```
Sys.setenv("PKG_CXXFLAGS"="-std=c++11")
```

L'implémentation de la parallélisation est plus délicate car il faut explicitement diviser les itérations de boucles en 4 paquets équivalents (les *chunks*) et les distribuer aux 4 *threads*. Pour cela on crée une *lambda* fonction (nouveauté du `C++11`, c'est une fonction sans nom que l'on déclare avec la syntaxe `[]`) ; cette fonction gérera les itérations entre `beg` et `end`. Il suffira donc de créer des taches consistant à lancer cette fonction sur les `beg/end` de chaque *chunk*, et de passer ces taches à `async` pour qu'elles soient réalisées en parallèle :

```
cppFunction('NumericVector dnormC_cpp11(NumericVector x,
                                        double mu,
                                        double sigma){
 using namespace std;
 size_t n = x.size();
 NumericVector density(n);
 vector<future<void>> tasklist; // les taches à faire
 for (size_t t=0; t<4; t++)
```

```
  tasklist.push_back( async(launch::async,
        [](size_t beg, size_t end,
           NumericVector& x,
           NumericVector& density,
           double mu, double sigma)
           // entête de la
           // lambda function
    {
      double sqrt2pi = sqrt(2*M_PI);
          for (size_t i=beg; i<end; i++)
            // réalise le calcul sur le chunk
            density[i] = 1/(sigma*sqrt2pi)
                 *exp(-0.5*(pow((x[i]-mu)/sigma,2)));
        },
    t*n/4, (t+1)*n/4, ref(x), ref(density), mu, sigma)
    // paramètres donnés à la lambda fonction
    // dont les bornes des chunks [beg,end]
    );
 for (size_t t=0; t<4; t++)
  tasklist[t].get();
 return density;
}
',includes=c("#include<cmath>","#include <future>",
"#include <vector>"))
```

A noter que :

- on a bien passé les variables d'entrée `x` et de sortie `density` sous la forme d'objet référence. Ainsi, `x` ne sera pas copiée et `density` pourra être modifiée dans la *lambda* fonction ;
- on a indiqué à **Rcpp** qu'il faut inclure le *header* `future` qui contient les fonctionnalités `std::async()` et `std::future<>`.

Cette version avec les *threads* de `C++11` est très performante, avec une accélération $\times 3.4$ avec 4 unités de calcul par rapport à `dnormC` :

```
microbenchmark(
 dnormC_cpp11(x,mu,sigma),
 times=10
)
Unit: milliseconds
min  lq   mean  median  uq max
9.81 9.88 10.10 9.93 10.11 10.77
```

4.4 RcppParallel

Le package **RcppParallel** propose une approche alternative basée sur la librairie `Intel Threading Building Blocks` de calcul parallèle en `C++` (TBB, `https://www.threadingbuildingblocks.org`). **RcppParallel** permet en effet de 1/ « brancher » TBB automatiquement sur R sans avoir besoin d'installer autre chose que le package **RcppParallel** et 2/ d'utiliser une interface conviviale à un sous-ensemble de TBB (les fonctions `parallelFor` et `parallelReduce`) utilisable par des programmeurs habitués à **Rcpp**.
Moyennant l'implémentation de quelques lignes de `C++`, **RcppParallel** permet de demander la parallélisation des boucles `for` (au contraire des *threads* **C++11**, la parallélisation et le découpage en *chunks* ne sera pas géré par le développeur, mais seulement gardé à l'esprit).
La librairie TBB propose une orientation 100 % programmation orientée objet et cela s'en ressent lorsqu'il s'agit de prendre en main **RcppParallel**... Tout sera basé sur les concepts clés d'*héritage* et de *classe fonction* ou *foncteur*. Il faut ainsi créer une structure (ou classe, même si le mot clé n'est pas utilisé), qui est appelée foncteur car elle ne propose qu'un constructeur et un opérateur `()` : au lieu d'appeler une fonction, on appelle l'opérateur`()` sur une instance du foncteur. Ce foncteur doit dériver de la classe `Worker` pour pouvoir être utilisé : il doit correspondre à l'opération à réaliser sur un *chunk* d'itérations de boucle, les itérations entre `beg` et `end`.

Reprenons l'exemple implementé avec `openMP` et `C++11`, et examinons la version **RcppParallel**. Il s'agira ici de développer un package R qui contient le code `C++` suivant dans le répertoire `src` :

```
#include <math.h>
#include <Rcpp.h>
#include <RcppParallel.h>
using namespace Rcpp;
using namespace RcppParallel;

struct Dnorm : public Worker // foncteur
{
 const RVector<double> x; // source
 RVector<double> density;   // destination
 double mu;
 double sigma;
 Dnorm(const NumericVector input,
    NumericVector output, double mu, double sigma)
    : x(input), density(output), mu(mu), sigma(sigma) {}
    // constructeur du foncteur
```

```
 // operation à réaliser sur le chunk [beg,end]
 void operator()(std::size_t beg, std::size_t end) {
  double sqrt2pi = sqrt(2*M_PI);
  for (size_t i=beg; i<end; i++)
   density[i] = 1/(sigma*sqrt2pi)
     *exp(-0.5*(pow((x[i]-mu)/sigma,2)));
 }
};
```

On crée le foncteur `Dnorm` qui dérive de la classe `Worker`. On constate que l'opérateur `()` agit sur le chunk de données entre `beg` et `end` : il sera utilisé indépendamment par chaque *threads* sur des *chunks* indépendantes. La suite du code fait apparaître la boucle parallélisée avec le `parallelFor` :

```
// [[Rcpp::export]]
NumericVector dnormC_rcpppar(NumericVector x,
                             double mu,
                             double sigma) {
 NumericVector density(x.length());
 Dnorm foncteur(x,density,mu,sigma);
 parallelFor(0, x.length(), foncteur);
 return(density) ;
}

RcppParallel::setThreadOptions(numThreads=4)
microbenchmark(
 dnormC_rcpppar(x,mu,sigma),
 times=10
)
Unit: milliseconds
min   lq    mean  median  uq  max
11.33 12.13 12.09 12.17 12.20 12.27
```

Avec 4 *threads*, on retrouve des performances équivalents aux approches `openMP` et `C++11`.

Cette alternative présente l'avantage de ne pas nécessiter d'étape de configuration (de variables d'environnement ou d'options de compilation) au contraire des approches basées sur `openMP` et `C++11`. La seule étape requise est en effet l'installation du package avec `install.packages("RcppParallel")`. Ce côté « R-*friendly* » est toutefois à nuancer, compte tenu du fait qu'une certaine maîtrise de la programmation objet en `C++` est nécessaire pour se servir de **RcppParallel**...

Chapitre 5

Calcul et données distribués avec R sur un cluster

Nous nous sommes jusqu'à présent limités à la mise en œuvre du parallélisme sur une machine multi-cœurs. Passer à l'utilisation d'un cluster de calcul, et en tirer profit grâce à un niveau de parallélisme adéquat (calcul ou données), se révèle bien plus technique... Comme précisé dans la section 2.2.3, un cluster est matériellement caractérisé par un ensemble de machines. On peut cependant distinguer deux types de cluster répondant plus particulièrement à deux usages très complémentaires :

- les clusters dédiés au *calcul intensif* et au *HPC* (*High Performance Computing*) ;
- les clusters dédiés au *calcul sur données distribuées* et au *big data* (données trop volumineuses pour un traitement classique).

Comment peut-on tirer bénéfice de ces architectures avec R ?

5.1 Cluster orienté HPC

Les principales caractéristiques des clusters dédiés au HPC sont :

- des machines (les nœuds) autonomes le plus homogène possible ;
- celles-ci sont reliées entre elles par un réseau très performant ;
- chaque nœud dispose de sa mémoire propre autonome (RAM et caches) ;
- un système de fichiers partagé est accessible par chacun des nœuds du cluster ;
- un disque local est souvent disponible sur chaque nœud, à privilégier pour les entrées-sorties intensives (fichiers temporaires,...) pour des raisons de performance.

Ces calculateurs sont principalement conçus et utilisés pour le calcul intensif, c'est-à-dire l'exécution de codes de calcul sur des problèmes particulièrement larges ou complexes, à forte dépendance entre tâches parallèles et de fait nécessitant des transferts de données importants (des *communications*).

Remarque
Les clusters de calcul peuvent évidemment être utilisés pour d'autres types d'applications, en particulier de façon très simple pour les applications de parallélisme embarrassant. Il faut cependant noter que dans ce cas les technologies réseaux avancées (très coûteuses, bien souvent l'*InfiniBand*) ne seront pas utiles puisque qu'il n'y aura que peu de communications. Le programmeur pourra donc se faire reprocher d'« utiliser un vélo de compétition pour aller chercher le pain »...

Chaque machine du cluster étant autonome, le développement d'un programme sur une architecture de calcul intensif de type cluster nécessite donc pour le programmeur d'expliciter les échanges de données entre les différents éléments du calcul. Ces communications se font dans la très grande majorité des cas via le modèle de passage de message (*Message Passing*). Le standard de fait actuel est défini par la norme *MPI (Message Passing Interface)*. Nous partirons donc du principe qu'un « cluster `MPI` » est à disposition du programmeur R avec l'ensemble des outils configurés au préalable (nous ne présenterons pas ici les détails des implémentations de `MPI`, etc...).

Remarque
Dans ce qui suit, nous nous placerons sur un cluster de type `Linux` (écrasante majorité des clusters disponibles).

5.1.1 Les bases de `MPI`

Aucun ouvrage sur le calcul parallèle se saurait éluder l'approche par passage de message : celle-ci, au travers de la norme `MPI`, implémentée dans de nombreuses bibliothèques libres ou constructeurs, permet le développement de codes capables de tirer profit de la puissance de calcul d'un cluster en permettant la coordination de processus s'exécutant sur des machines différentes. Disponible dans des interfaces `C/C++, Fortran, Python`, elle l'est donc aussi pour R au travers des packages **Rmpi** et **pbdMPI**.

Le principe fondamental de `MPI` est le suivant :

1. Un unique programme est écrit dans un langage supportant `MPI` ;
2. Le programmeur lance conjointement l'exécution de ce programme dans plusieurs processus qui sont liés et qui tournent possiblement sur plusieurs nœuds du cluster ;
3. Ce programme comporte des branchements (des `if`) qui précise le travail à effectuer en fonction du numéro du processus (le *rang*) ;

4. Les processus s'échangent des données par passage de message.

Toutes les variables du programme sont privées et sont localisées dans la mémoire locale de chaque processus sur chacun des nœuds. Si nécessaire, les données sont échangées entre deux ou plusieurs processus via un appel à des fonctions spécifiques à `MPI` par l'intermédiaire d'un *message*. Un message est constitué :

1. des données,
2. de l'identifiant du processus émetteur,
3. de l'identifiant du processus récepteur,
4. d'un numéro de message appelé *tag*,
5. et éventuellement du type de données et de sa taille.

L'*environnement MPI* gère les envois et réceptions de message. Une application `MPI` est donc un ensemble de processus autonomes exécutant chacun le même code et communiquant via des appels à des fonctions propres à la bibliothèque `MPI`. A noter qu'il est nécessaire d'initialiser l'environnement en début de programme et de le désactiver à la fin via des appels à des fonctions spécifiques.

Enfin, la brique élémentaire sur laquelle reposent les communications de `MPI` est la notion de *communicateur*. Un communicateur est un ensemble de processus qui peuvent communiquer entre eux. Deux processus ne pourront s'échanger des données que si ils sont dans le même communicateur. Il existe un communicateur initial par défaut qui englobe tous les processus et qui conviendra à la majorité des développeurs R. Enfin, on peut facilement accéder au nombre de processus gérés par un communicateur ainsi qu'au rang de chaque processus avec des fonctions *ad-hoc*.

Le programmeur R qui souhaite utiliser `MPI` doit bien avoir en tête que le même code sera exécuté par plusieurs processus... Il faut donc donner des instructions à chaque processus. Imaginons de nouveau que l'on se met à plusieurs pour repeindre une pièce qui a de nombreux murs : chacun des travailleurs (les processus) recevra une feuille de route (le code). Cette feuille de route, écrite en pseudo-code ci-dessous, respecte l'esprit de `MPI` :

```
initialisation_du_groupe_de_travailleurs()

mon_rang <- quel_est_mon_rang()

si mon_rang est 0 alors :
 je suis le coordinateur
 je fais des étapes de pré-traitement (achat matériel)
 j'envoie des instructions aux travailleurs (qui peint quoi)
 je peins mon mur
sinon :
```

```
 j'attends de recevoir les instructions du coordinateur
 je peins mon mur

tous_les_travailleurs_se_synchronizent()

si mon_rang est 0 alors :
 je fais des étapes de post-traitement (rangement du matériel)

finalisation_du_groupe_de_travailleur()
```

On aura noté que le rang 0 sert de coordinateur et sera parfois appelé processus `master`.

5.1.2 R et MPI

A l'heure de l'écriture de cet ouvrage, il semble que le package **pbdMPI** présente de meilleures garanties concernant l'activité de développement, les performances et l'aisance de prise en main que son semblable **Rmpi**. Nous choisissons donc d'expliciter des cas d'utilisations réels de R et `MPI` au travers de **pbdMPI**.

Le squelette de base d'un code R s'interfaçant avec `MPI` grâce à **pbdMPI** se présente de la façon suivante :

```
# chargement du package pdbMPI
library(pbdMPI)

# Initialisation de l'environnement
init()

# Récupération des informations sur le nombre de processus et le
# rang du processus courant pour le communicateur par défaut
.comm.size <- comm.size()
.comm.rank <- comm.rank()

# ... reste du code, branchements ...

# Désactivation de l'environnement
finalize()
```

Pour utiliser ce code, il est nécessaire de l'exécuter en mode *batch* (en « toile de fond ») depuis un terminal. En effet, les codes contenant des appels aux fonctions de **pbdMPI** ne peuvent qu'être lancés en mode batch avec `Rscript` et pas en mode interactif (dans la console R) : ceci est une des différences avec **Rmpi** qui accepte la

mode interactif. Pour cela, le programmeur préfixera la commande `Rscript` par la commande `mpirun -np m` où `m` est le nombre de processus `MPI` à mettre en œuvre. Supposons par exemple que le code R est dans un fichier `mon_code.R`, alors son utilisation se fera ainsi :

```
# MPI avec 4 processus
mpirun -np 4 Rscript mon_code.R
```

et 4 processus R seront lancés et liés par `MPI`.

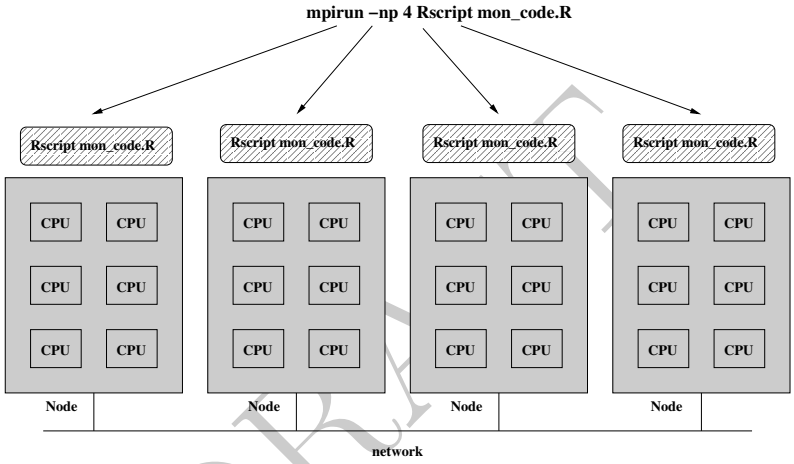


Fig. 5.1 – Fonctionnement d'un cluster `MPI`. La commande `mpirun -np 4` va lancer 4 processus qui exécuteront le code `mon_code.R` en parallèle sur 4 noeuds différents du cluster. Les nœuds sont reliés par un réseau très performant.

Il est désormais nécessaire d'en savoir plus sur la syntaxe associée aux communications `MPI`, clé de voûte de tout programme basé sur `MPI`.

Communications `MPI`

Les communications dites *point à point* ont lieu entre deux processus à l'intérieur d'un communicateur leur permettant d'échanger une donnée. Les deux processus sont dans ce cas nommés respectivement *émetteur* et *récepteur*, et sont identifiés par leur rang. Les fonctions permettant de réaliser des communications point à point sont : `send()` (envoi du message), `recv()` (réception du message) et `sendrecv()` (envoi et réception simultanés).

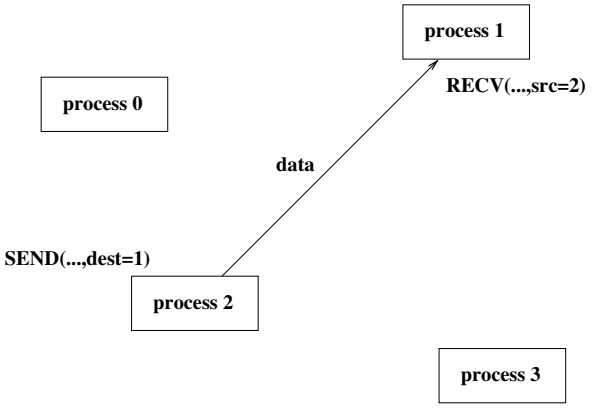


Fig. 5.2 – Communications point à point.

Remarque
Il est important d'être vigilant sur la programmation des communications point à point afin de ne pas être confronté à une situation de blocage ou *deadlock*. Une situation typique de blocage se produit lorsqu'au moins deux processus `MPI` veulent échanger des messages, mais tous les 2 veulent envoyer leur message respectif et ne sont pas prêts à recevoir. L'exécution d'un programme faisant appel à la fonction `send()` suivie de `recv()` peut également rester bloquée sur `send()` si au moins deux processus s'échangent des messages de tailles importantes. Cette situation est liée au mode d'envoi des messages. En général, les petits messages sont recopiés dans un espace mémoire temporaire avant d'être expédiés (*bufferisés*), alors que les gros messages sont envoyés en mode *synchrone*. Ce mode synchrone implique que pour que le message parte (c'est-à-dire pour que l'appel `send()` puisse se terminer), il faut que la réception de ce message ait été postée (c'est-à-dire que l'appel à `recv()` ait été effectué). Un scénario classique de blocage est le suivant par exemple :

Process 0	Process 1
`recv(...,rank.source =1)`	`recv(...,rank.source=0)`
`send(...,rank.dest=1)`	`send(...,rank.dest=0)`

Cette situation est verrouillée quelque soit la taille des messages envoyés. Les deux processus sont bloqués au `recv()` et les `send()` ne s'exécutent pas. Une solution pourra consister à passer en mode *asynchrone* (pas plus de détails ici).

Considérons une exemple classique de chaîne de traitement en bioinformatique qui conduit à du parallélisme embarrassant. Il s'agit de traiter en parallèle des fichiers de séquences d'ADN. Supposons en effet que nous

avons un répertoire qui contient plusieurs jeux de séquences d'ADN (composées des lettres `A/G/C/T`) et que l'on souhaite les comparer sur la base des fréquences des *k-mers* (*i.e.* les mots de longueur k ; par exemple `CATA` pour $k = 4$). Il faudra donc appliquer la fonction suivante sur chaque fichier pour obtenir ces fréquences (les non-bioinformaticiens ne s'attarderont pas sur les détails d'implémentation basée sur le package **ShortRead**) :

```
get.kmer.freq <- function(path, file){
 kmer.size <- 2
 library(ShortRead)
 rfq <- readFastq(path, file)
 freqs <- colSums(
             oligonucleotideFrequency(sread(rfq), kmer.size)
          )
}
```

Pour traiter les fichiers de séquences en parallèle avec `MPI`, nous allons créer un fichier `fasta2kmers.R` qui contient la définition de la fonction `get.kmer.freq` ainsi que l'ensemble des lignes de codes que nous allons détaillées par la suite.

Il est tout d'abord nécessaire de préciser le répertoire qui contient les fichiers, comme par exemple celui associé au package **ShortRead**) :

```
path.to.fastq <- paste(
                   system.file('extdata',package='ShortRead'),
                   "E-MTAB-1147",
                   sep="/")
```

Puis il faut initialiser le communicateur et chaque processus récupère son rang :

```
library(pbdMPI)
init()
.comm.size <- comm.size()
.comm.rank <- comm.rank()
```

Ensuite, le processus de rang 0 va découper la liste des fichiers en *chunk*s (ou paquets) de taille équivalente, puis transmettre chaque *chunk* à chaque processus :

```
if(.comm.rank == 0){
 ## liste des fichiers de séquences à traiter
 files <- list.files(path.to.fastq, pattern="*fastq*")
 nb.files <- length(files)
```

```
 ## découpage cette liste en .comm.size chunks
 ## grâce à la fonction get.jid de pbdMPI
 jid <- get.jid(nb.files, all = TRUE)
 chunks.of.files <- lapply(1:.comm.size,
                           FUN=function(i) files[jid[[i]]])

 ## envoi des chunks depuis le rang 0 vers les autres rang
 ## grâce à la fonction send de pbdMPI
 for (rank in 1:(.comm.size-1)){
  send(chunks.of.files[[rank+1]], rank.dest=rank, tag=rank)
 }

 ## le premier chunk est pour le rang 0
 files <- chunks.of.files[[1]]
}
```

Chaque processus de rang > 0 va émettre un avis de réception de son *chunk* et le récupérer dans la variable `files` :

```
## réception du chunk grâce à la fonction recv de pbdMPI
if(.comm.rank != 0){
 files <- c()
 files <- recv(files, rank.source=0, tag=.comm.rank)
}
```

Tous les processus (y compris celui de rang 0) vont travailler sur leur *chunk* de fichiers, c'est à dire exécuter la fonction `get.kmer.freq` sur chaque fichier du *chunk* :

```
## travail indépendant de chaque rank sur chaque chunk
res <- sapply(files, get.kmer.freq, path=path.to.fastq)
```

Pour finir, les processus de rang > 0 doivent envoyer leur résultat `res` au processus de rang 0 tandis que celui-ci émet des avis de réception de ces résultats et les fusionne après réception :

```
## récupération centralisée des résultats
if(.comm.rank == 0){
 total.res <- list(res)
 for (rank in 1:(.comm.size-1)){
  ## réception du res de chaque processus de rang >0
  res <- recv(res, rank.source=rank, tag=rank)
  total.res[[length(total.res)+1]] <- res
 }
```

```
 print(total.res)
} else{
 ## envoi de res vers le processus 0
 send(res, rank.dest=0, tag=.comm.rank)
}
```

Le code se terminera par l'instruction :

```
finalize()
```

Pour exécuter ce code, il sera nécessaire de passer par la console `Linux` et de lancer le mécanique de `MPI` comme suit :

```
mpirun -np 2 Rscript fastq2kmers.R
```

Nous parlerons performances un peu plus tard...

Communications collectives

Les communications dites *collectives* permettent de faire en une seule opération une série de communications point à point impactant tous les processus du communicateur concerné. Pour chacun des processus, l'appel se termine lorsque la participation de celui-ci à l'opération collective est achevée. La synchronisation globale de tous les processus est donc implicite après une communication collective.
Il y a trois types de fonctions permettant la gestion des communications collectives :

- celles qui ne font que transférer des données :
  - diffusion globale de données (*broadcast*) : `bcast()` ;
  - diffusion sélective de données : `scatter()` ;
  - collecte de données réparties : `gather()` ;
  - collecte par tous les processus de données réparties : `allgather()` ;
  - collecte et diffusion sélective, par tous les processus, de données réparties : `alltoall()`.
- celles qui, en plus de la gestion des communications, effectuent des opérations sur les données transférées :
  - opérations de réduction : `reduce()`. Une réduction est une opération (somme, produit, maximum, minimum, etc.) appliquée à un ensemble d'éléments pour en obtenir une seule valeur (voir aussi 4.2.1) ;
  - opérations de réduction. avec diffusion du résultat (équivalent à un `reduce()` suivi d'un `bcast()`) : `allreduce()`.

— celle qui assure les synchronisations globales : barrier() .

On notera également que **pbdMPI** s'appuie sur le package **rlecuyer** pour la gestion des générateurs de nombres pseudo-aléatoires en parallèle (de la même façon qu'au chapitre 3). La communication collective `comm.set.seed()` permet ainsi d'initialiser les générateurs en parallèle.

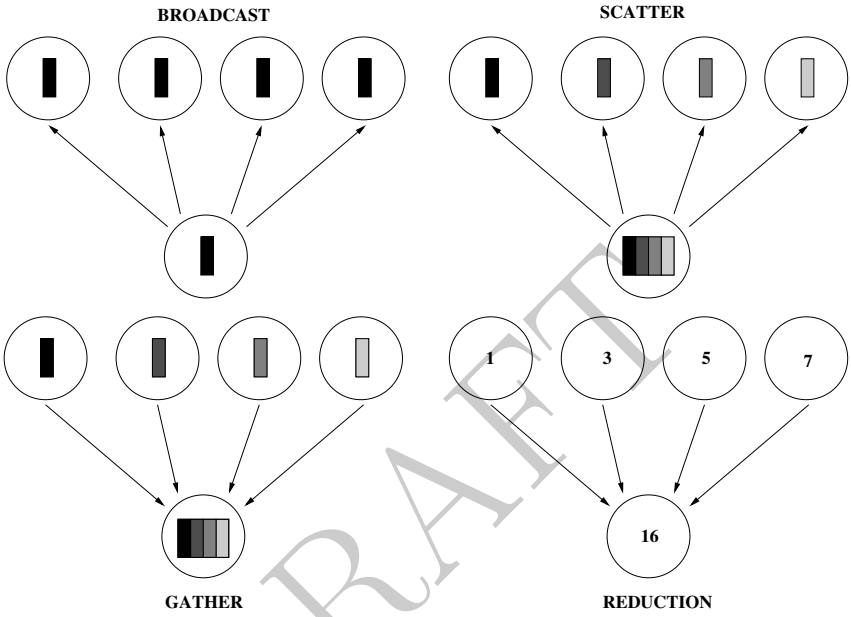


Fig. 5.3 – Communications collectives.

Remarque

Cette petite introduction à `MPI` a permis de passer en revue les principaux éléments de cette norme. Il faut cependant noter que `MPI` est bien plus complet, et permet entres autres de faire des communications beaucoup plus complexes, synchrones ou asynchrones, persistantes ou non, avec recouvrement de calcul. La norme intègre également la gestion d'entrées/sorties collectives. Un document de référence pour l'apprentissage de `MPI` est incontestablement le cours proposé par l'IDRIS (Institut du développement et des ressources en informatique scientifique) disponible en ligne (IDRIS-CNRS, 2016). Le programmeur `R` trouvera également de remarquables explications dans l'ouvrage Hager & Wellein (2011).

Reprenons notre exemple de bioinformatique. Il est possible de largement simplifier l'implémentation en remplaçant l'envoi des *chunk*s avec `send/recv` par une communication collective `scatter` :

```
## scatter des chunks de fichiers
files <- scatter(chunks.of.files, rank.source=0)
```

De même, on remplacera avantageusement l'utilisation des `send/recv` par une communication collective `gather` pour la récupération des résultats au niveau du processus 0 :

```
## récupération centralisée des résultats
## après synchronization
barrier()
res <- gather(res)
```

Dans les deux cas, `scatter` et `gather` doivent être exécutés par chaque processus et ils ne doivent donc pas apparaître dans un branchement de type `if(.comm.rank==0)` ou `if(.comm.rank!=0)`.

Cette configuration de parallélisme embarrassant implémenté avec `MPI` s'avère très efficace. A titre indicatif, voici les performances de ce code utilisé pour analyser 80 millions de séquences réparties en 10 fichiers de taille variable (de 300 à 800 Mo) sur le cluster du CC LBBE/PRABI :

nombre de noeuds du cluster (un processus `MPI` par noeud)	1	2	5
temps (secondes)	566	329	126

Dans ce test de performance, il n'y a qu'un processus `MPI` par noeud : il est bien-sûr possible de lancer plusieurs processus `MPI` par noeud lorsque ceux-ci intègrent plusieurs cœurs de calcul.

Remarque

On notera également l'existence du package **pbdMAT** qui permet d'utiliser les techniques précédentes sous forme de « boîte noire » pour manipuler des matrices distribuées (au sens de `MPI`) avec en particulier une collection de fonction pour l'algèbre linéaire.

Dans le même esprit « boîte noire », des versions parallèles des `*apply` sont disponibles dans `pbdMPI` et utilisables sans que le programmeur n'ait à se soucier de la mécanique sous-jacente (qui, on s'en doute, reprend les techniques implémentées dans les fonctions de base de `pbdMPI`). Nous proposons ici un exemple illustratif :

```
### A lancer avec: mpirun -np 2 Rscript nomfichier.R
library(pbdMPI)
init()
X <- matrix(...)
Y <- pbdApply(X, 1, sum)
finalize()
```

Derrière cet appel à `pbdApply` se cachent plusieurs étapes : la matrice `X` sera d'abord distribuée depuis le processus `master` grâce à un `scatter`, puis tous les processus réaliseront le calcul (à base de `sum`) sur leurs sous-parties de matrice, et enfin les résultats seront envoyés avec un `gather` vers le `master` qui les fusionnera.

5.2 Cluster orienté *big data*

Les clusters dédiés aux données sont intéressants si la quantité de données à traiter est réellement importante et dépasse la capacité de la mémoire vive et des disques durs actuels. Dans ce qui suit, nous nous bornerons à évoquer les principes des clusters orienté données sans rentrer dans les détails : en effet, ces technologies sont en perpétuelle évolution et le lecteur trouvera de nombreuses ressources documentaires 1/ à jour et 2/ exhaustives sur l'internet ou dans différents ouvrages (McCallum & Weston (2011); Lim & Tjhi (2015); Matloff (2015)).

Hadoop

Les clusters dédiés au calcul intensif, très onéreux du fait du coût des technologies réseau, ne sont pas forcément nécessaires pour traiter de données massives. Deux ingénieurs de Google, Jeffrey Dean et Sanjay Ghemahat ont ainsi proposé une approche originale : plutôt que de déplacer les données (et donc d'avoir besoin d'un réseau performant, comme pour `MPI`), pourquoi ne pas déplacer le code ? ! Et, dès lors, pourquoi ne pas s'appuyer sur des machines potentiellement d'entrée de gamme, sans spécificité particulière, non nécessairement homogènes et connectées par un réseau classique ? ! Leur article (Dean & Ghemawat, 2008) décrit les principes de ce nouveau paradigme : c'est sur cette approche que repose les clusters *Hadoop* (*High-availability distributed object-oriented platform*).

`Hadoop` [`http://hadoop.apache.org`) propose un système de stockage distribué via son système de fichier `HDFS` (Hadoop Distributed File System). C'est la clé de voûte de `Hadoop` : avec `HDFS`, les données sont directement stockées de manière distribuée sous forme de blocs répartis sur les différents noeuds. Il propose également un système d'analyse des données appelé *MapReduce* qui interagit avec le système de fichiers `HDFS` pour réaliser des traitements sur des gros volumes de données. Popularisé depuis une décénie, MapReduce est un algorithme de décomposition itérative (voir 2.3.2). Son principe est simple : il s'agit de décomposer une tâche en sous-tâches qui sont des tâches identiques portant sur des sous-ensembles des données initiales. Les sous-tâches (et leurs sous-données) sont ensuite déployés vers différents noeuds, puis les résultats sont récupérés. La phase de décomposition en sous-tâches est appelée *map*, la phase de gestion des résultats est appelée *reduce*. La décomposition étant récursives, des noeuds peuvent aussi déléguer vers d'autres noeuds. Chaque sous-tâche sera préférentiellement attribuée à un noeud qui stocke déjà les sous-données correspondantes. Ainsi, chaque noeud apporte sa puissance de calcul et ses possibilités de stockage.

Les clusters Hadoop sont conçus pour passer à l'échelle : il est facile de rajouter des nœuds si le volume des données est trop important. Ceux-ci sont aussi tolérants aux pannes : les données ne sont pas perdues en cas de défaillance d'un nœud car chaque bloc est répliqué sur plusieurs noeuds (trois par défaut).

Hadoop propose nativement une implémentation Java du MapReduce. Cette implémentation suppose que le programmeur a écrit les fonctions map() et reduce() dans le langage de son choix (et pourquoi pas R !) : ce sera la première alternative pour méler R et Hadoop. Le système se charge d'appliquer sur chaque bloc de données distribuées la fonction map(), de façon totalement transparente pour l'utilisateur, et gère également la machinerie derrière les appels successifs à la fonction reduce(). Une deuxième alternative résidera dans l'utilisation de **RHadoop** qui est une collection de cinq packages pour la manipulation et l'analyse de données avec Hadoop. Les fonctions de ces packages sont de haut niveau et permettent une abstraction des éléments techniques autour d'Hadoop. Bien-sûr, il est possible de réaliser une approche MapReduce avec le package **rmr2** et sa fonction mapreduce(). Par ailleurs, le package **rhdfs** permet d'intéragir avec le système de fichiers HDFS (lecture, écriture, modification) via R. Enfin, le package **plyrmr** de RHadoop implémente dans un cadre MapReduce un ensemble de fonctionnalités de manipulation de données, suivant la même syntaxe et logique que les packages **plyr** et **reshape2**, mais sur des données massives stockées sur un cluster Hadoop.

Spark

Spark (http://spark.apache.org/) est une alternative « in-memory » (c'est à dire qui exploite la mémoire plutôt que le disque) plus rapide que le traditionnel MapReduce de Hadoop (lui-même basé sur des écritures continues sur le système de fichiers). Développé à l'origine dans le laboratoire AMPLab de l'université de Berkeley, c'est désormais un projet phare de la fondation Apache et la version 1.0.0 de Spark est sortie en mai 2014.
La couche logicielle Spark est dédiée à la mise en œuvre aisée des tâches d'analyse de données de manière distribuée. Spark s'exécute sur des infrastructures Hadoop et utilise HDFS, mais est compatible avec d'autres systèmes de cluster orienté données. La clé de la performance de Spark provient de l'utilisation de la mémoire : Spark permet de partager les données en mémoire et les résultats intermédiaires restent en mémoire, ce qui présente un avantage considérable lorsque il s'agit de travailler à plusieurs reprises sur la même donnée. Spark est interfacé avec Java, Python et depuis 2015 avec... R dans le package **SparkR**. **SparkR** permet de développer des chaînes de traitement de données à base de sélection, de filtres ou d'agrégations dans un esprit proche de celui de **dplyr**.

Annexe A

Notions complémentaires

A.1 Problématique du calcul flottant

Peut-on faire confiance aux résultats de calcul d'un ordinateur ? Cette question peut paraître délicate dans un ouvrage dédié à l'utilisation de machines performantes pour réaliser des calculs. Elle n'est cependant pas anodine. En effet, les mathématiques manipulent des nombres réels, contrairement aux ordinateurs qui ne peuvent traiter que des nombres représentés sur un nombre fini de *bits* (élément de base de la représentation de l'information pour un ordinateur). L'implémentation de méthodes mathématiques entraîne nécessairement le choix d'un arrondi dans la plupart des opérations effectuées.

Si ces imprécisions ne sont pas contrôlées, elles peuvent provoquer des situations potentiellement critiques dans certaines circonstances. Ainsi ce missile Patriot tiré en 1991 de Dahran en Arabie Saoudite a manqué l'interception d'un missile irakien qui a provoqué la mort de 28 soldats et blessé plusieurs dizaines d'autres personnes. L'erreur était due à une imprécision dans le calcul de la date du Patriot : une multiplication par 1/10 pour obtenir le temps en seconde. Or 1/10 n'est pas représentable exactement en machine, l'erreur s'est accumulée pour arriver à un décalage suffisant dans le calcul de l'heure pour rater le missile irakien.

Heureusement, le calcul sur ordinateur ne conduit pas toujours à de telles catastrophes ! Cependant, les problèmes liés aux erreurs d'arrondi, et à leur accumulation sont encore plus importants sur des machines parallèles.

L'arithmétique des nombres réels sur ordinateur est dite *arithmétique à virgule flottante*. Nous ne détaillerons pas ici cette représentation mais nous allons brièvement citer ses principales caractéristiques. Pour approfondir le sujet, nous recommandons l'excellent ouvrage Muller *et al.* (2010). Pendant longtemps, ces représentations flottantes étaient implémentées différemment selon la machine sur laquelle on calculait : le résultat d'une simple opération arithmétique pouvait alors avoir un résultat différent d'un ordinateur à l'autre.

La norme IEEE 754, introduite en 1985 et révisée en 2008, a permis d'harmoniser l'arithmétique flottante en spécifiant la représentation des nombres, le comportement des opérations élémentaires, les modes d'arrondis, les valeurs spéciales (Nan, infty, ...). Elle permet notamment de favoriser le déterminisme des programmes d'un ordinateur à l'autre.
Ce n'est cependant pas toujours le cas en particulier pour des programmes parallèles. En effet, ces applications ne sont pas déterministes car l'ordre d'exécution des calculs ne l'est pas : il dépend de nombreux paramètres (charge de la machine, accès aux données, ordonnancement...). Plusieurs propriétés de l'arithmétique (associativité, distributivité, ...) ne sont plus valides en arithmétique flottante. Le fait de modifier l'ordre des calculs conduit donc à des résultats potentiellement différents (de l'ordre de la précision de la machine). Les conséquences sont à prendre en compte, en particulier quand on implémente des tests (voir 1.1.3).
L'exemple trivial suivant permet d'illustrer une des problématiques associées au calcul flottant, celle de l'arrondi :

```
> if (0.1 + 0.2 == 0.3) print('Super, non ?')
> if (! 0.1 + 0.2 == 0.3) print('Bizarre, non ?')
[1] "Bizarre, non ?"
```

L'explication est simple : 0.1 n'est pas représentable exactement en machine, l'addition entraîne donc une erreur d'arrondi.

« Floating point numbers are like piles of sand ; every time you move them around, you lose a little sand and pick up a little dirt. » (Kernighan & Plauger, 1978)

A.2 La complexité

La *complexité* d'un problème caractérise les ressources nécessaires pour les résoudre. Les ressources essentielles sont le temps (d'éxecution) et l'espace (mémoire). Jusqu'aux années 70, seule la mesure expérimentale de la complexité d'un algorithme était (parfois) effectuée et dépendait des machines... une mesure théorique devint souhaitable ! La *complexité algorithmique* est un ordre de grandeur théorique du temps de calcul (*complexité en temps*) et/ou de l'espace mémoire (*complexité en mémoire*) utilisé en fonction d'une mesure des données. Ces deux complexités sont importantes car, fréquemment, si un algorithme permet de gagner en temps, il occupe davantage de mémoire...
Il existe trois types de complexité :

— la complexité *au pire*, *i.e.* la complexité « maximum » dans le cas le plus défavorable ;

— la complexité *en moyenne*, *i.e.* la moyenne des complexités obtenues selon les cas ;

— la complexité *au mieux*, *i.e.* la complexité « minimum », dans le cas le plus favorable. En pratique, cette complexité n'est pas très utile.

Premons l'exemple de la recherche d'un élement dans une liste non triée de taille n. A pire, il faudra n opérations, en moyenne $(n+1)/2$ opérations et au mieux 1 opération.

Pour comparer des algorithmes sans considérer l'implémentation, on peut comparer les complexités au regard de la taille attendue des données. Pour cela, on s'intéresse à la complexité au pire et à sa forme générale qui indique la façon dont elle évolue en fonction d'un paramètre (ou plusieurs). On utilisera alors la *notation de Landau* $O(\ldots)$ qui veut dire « de l'ordre de… ».

Définition A.1
Notation de Landau : la fonction positive g est dite en $O(f)$ s'il existe des constantes $c > 0$ et $x_0 >> 0$ tq $g(x) < c * f(x)$ pour tout $x > x_0$. On note $g = O(f)$ et on dit que g est dominée asymptotiquement par f.

Propriété A.1
Si $g = O(f)$ et $f = O(h)$ alors $g = O(h)$.
Si $g = O(f)$ et k un nombre, alors $k * g = O(f)$.
Si $f1 = O(g1)$ et $f2 = O(g2)$ alors $f1 + f2 = O(g2 + g2)$.
Si $f1 = O(g1)$ et $f2 = O(g2)$ alors $f1 * f2 = O(g2 * g2)$.

Exemple A.1
Si n est la taille des données, la complexité en temps est :

— $O(1)$ *constante* pour l'extraction d'un élément dans un vecteur,

— $O(log(n))$ (*logarithmique*) pour la recherche dichotomique dans un vecteur triée (voir après),

— $O(n)$ (*linéaire*) pour le parcours d'une liste,

— $O(n\ log(n))$ (*quasi-linéaire*) pour le tri d'un vecteur,

— $O(n^2)$ (*quadratique*) pour le calcul du maximum d'une matrice carré,

— $O(n^3)$ (*cubique*) pour la multiplication matricielle,

— $O(2^n)$ (*exponantielle*) pour le problème du sac à dos par force brute (énumération).

On détaille ici le calcul de la complexité en temps de la recherche dichotomique de x dans un vecteur trié : on compare x à la valeur du milieu y du vecteur ; si $x > y$, on s'intéresse au segment qui commence en y, sinon au segment qui se termine en y. On répète l'opération de comparaison. Et ainsi de suite. Les segments de tableau à traiter se succèdent et sont de taille n, puis $n/2$, puis $n/4$, ..., jusqu'à ce que $n/2^t = 1$ au pire. Le nombre de comparaisons est donc $t = log(n)/log(2)$. La complexité en temps est donc $O(log(n))$.

Remarque
La notation de Landau peut utiliser plusieurs paramètres décrivant la taille du problème (par exemple, $O(m * n + q)$).

Le nombre de processeurs disponibles est une donnée à prendre en compte dans la complexité. Deux alternatives pour exprimer la complexité sont alors envisageables :

— $O(f(n))$ avec $g(n)$ processeurs, qui exprime la complexité obtenue avec un nombre idéal de processeurs ;

— $O(f(n, m))$ avec m le nombre de processeurs, qui est plus pragmatique.

La complexité nous aide à choisir le bon algorithme et les bonnes structures de données. C'est un indicateur, mais il faut faire attention aux constantes devant le $O(\ldots)$! Par ailleurs, il faut impérativement veiller aux coûts réels de l'implémentation engendrés par les accès mémoire, la bande passante, les architectures des processeurs, etc.

A.3 Typologie des langages

On parle de *niveau* d'un langage en fonction de la nécessité imposée au programmeur de connaître le fonctionnement d'un ordinateur. Le langage de très bas niveau est le langage machine binaire. Plus le niveau est bas, plus les performances sont importantes. `Fortan` et `C/C++` sont considérés de niveau intermédiaire (permet la gestion fine de la mémoire par exemple, et donc de la performance). `Python` ou `Perl` sont des langages de haut niveau, de même que `Matlab/Scilab` ou `R` (parfois appelés *environnements de programmation scientifique*).

Un langage est dit *compilé* quand le *code* ou programme *source* sous forme de texte est tout d'abord lu et traité par un autre programme appelé *compilateur* qui le convertit en langage machine. Le compilateur signale les erreurs syntaxiques présentes dans le code source. Cette phase de compilation peut parfois être très longue. La compilation (et *l'édition des liens* qui assemble différents blocs de langage machine) produit un exécutable autonome qui ne fonctionne que sur le type de machine (OS, 32/64 bits) où la compilation s'est déroulée. `C/C++ et Fortran` sont des langages compilés (avec un choix de compilateurs gratuits ou non). Un programme en langage *interprété* nécessite pour fonctionner un interprète(eur) qui est un autre programme qui va vérifier la syntaxe et traduire directement, au fur et à mesure de son exécution, le programme source en langage machine (un peu comme un interprète durant une interview). Un programme interprété sera plus lent qu'un programme compilé du fait de la traduction dynamique. Quand une ligne du programme doit être exécutée un grand nombre de fois, l'interpréteur la traduit autant de fois qu'elle est exécutée. Néanmoins la correction des erreurs sera plus simple car l'interprète signale à l'exécution où se trouve l'erreur. Et le code source venant d'être écrit peut être directement testé. On notera enfin qu'un

programme dans un langage interprété est parfois appelé *script*. `Python`, `Perl` et R sont des langages interprétés.

Les langages de *typage* fort dit *statique* imposent la déclaration précise de toutes les variables (type, signe, taille) et les éventuelles conversions doivent être explicites. `C/C++ et Fortran` sont de typage statique. Les langages non typés ou de typage *dynamique* sont très souples avec les variables : pas de déclaration et possibilité de changement de type à la volée. La grande flexibilité que permet le typage dynamique se paye en général par une surconsommation de mémoire correspondant à l'encodage du type dans la valeur. `Python`, `Perl` et R sont de typage *dynamique*.

Un langage est dit *procédural* s'il repose sur des enchaînements de procédures/fonctions sur des données globales. `C, Fortran` ou R sont des langages intrinsèquement procéduraux. Un langage est dit *orienté objet* s'il est basé sur la *programmation orientée objet* qui, entre autres principes, requiert de regrouper les données et les fonctionnalités associés en entités logicielles autonomes (les classes). Les principes de la programmation orientée objet sont :

— identifier et grouper les « choses qui vont bien ensemble » en modules ;
— les modules sont testés indépendemment pour assurer la robustesse du code (tests unitaires) ;
— séparer les modules les uns des autres (*modularité*) ;
— réduire au strict minimum la visibilité inter-modules (*encapsulation*) ;
— si nécessaire : structuration hiérarchique des modules (*héritage*) ;
— programme principal : chef d'orchestre entre modules.

`C++` et `Python` sont nativement des langages orientés objet ; à noter toutefois que les classes S4 de R permettent de programmer en R en respectant les principes de la programmation orientée objet.

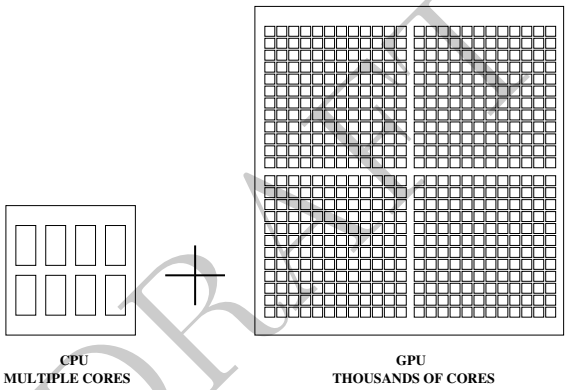
A.4 Les accélérateurs

Nous avons détaillé dans la partie 2.2.3 les différents types de machine utilisés actuellement pour exécuter des codes de calcul.
En particulier, ces dernières années ont vu émerger des architectures hybrides intégrant des cartes accélératrices : carte de type graphique (*GP-GPU*, *General-Purpose computation on Graphic Processing Unit*) puis carte *many-cœurs* (comme les Intel Xeon Phi).

A.4.1 GP-GPU, General-Purpose computation on Graphic Processing Unit

Initialement, les GPU étaient dédiées aux tâches de rendu graphique. En raison du marché très porteur des jeux vidéo, ces cartes sont devenues de plus en plus

performantes, intégrant un nombre d'unités de calcul de plus en plus important. Outre cette capacité de calcul potentielle considérable, la consommation énergétique de ces cartes est plus faible à performance égale que celle des CPU. C'est aussi un matériel peu coûteux qui s'intègre dans la plupart des stations de travail. Les premiers GPU programmables avec un langage de programmation de haut niveau sont apparus en 2007. Depuis lors, l'intérêt qu'ils suscitent n'a fait que croître dans la communauté du calcul.
En parallèle, les outils disponibles pour exploiter ces architectures très particulières se sont beaucoup développés, simplifiant quelque peu la tâche du développeur. Cependant, pour tirer le meilleur parti de ces architectures, il est important de comprendre leurs caractéristiques très différentes des processeurs classiques.
En particulier, les GPU intègrent un très grand nombre d'unités de calcul, et sont donc particulièrement adaptés aux traitements de tâches massivement parallèles et aux algorithmes présentant un fort parallélisme de données.



Leur programmation efficace n'est pas aisée car elle implique une réflexion sur le placement des données en mémoire, les séquences d'accès à celles-ci ainsi que de régler différents paramètres architecturaux propres aux GPU. Le principe est le suivant : le CPU contrôle l'exécution du programme et va en accélérer quelques parties en déportant certains calculs sur le GPU. Cela nécessite notamment de transférer les données du processeur à la carte graphique et inversement, une opération particulièrement coûteuse en temps.
Il existe plusieurs modalités pour programmer des cartes graphiques :

— utiliser des langages dédiés comme `CUDA` ou `OpenCL`. `CUDA` est proposé par NVidia et ne fonctionne donc que sur les matériels de cette marque. C'est une API qui a atteint un niveau de maturité particulièrement intéressant, avec une forte communauté de développeurs, et donc beaucoup de bibliothèques disponibles. `OpenCL` est un standard plus généraliste puisqu'il permet d'adresser tout processeur à plusieurs cœurs. C'est l'approche la plus compliquée mais aussi celle qui permet d'approcher les performances les plus intéressantes ;

— utiliser `openACC` (*Open Accelerators*), un standard très récent, basé comme `openMP` sur des directives de compilation. Cette technologie permet de programmer les accélérateurs de façon non intrusive et portable. Encore très jeune, c'est très probablement une approche prometteuse.

Cet essor des cartes graphiques dans le monde du calcul, caractérisé en particulier par leur omniprésence dans les supercalculateurs les plus puissants au monde (voir le classement du *top 500*), a également un impact dans le monde R avec le développement de plusieurs packages permettant d'exploiter les GPU comme **gputools, gmatrix** et **Rth**.

A.4.2 Carte many-cœurs

Aujourd'hui d'autres architectures massivement parallèles, comme la carte Intel Xeon Phi, viennent concurrencer les GPU. Ces cartes intègrent un nombre de cœurs important (60 pour la première version). Ces spécificités sont très liées au fait que le Xeon Phi est une architecture `x86` native et donc ne nécessite pas de langage de programmation différent : un code en `C`, `C++` ou `Fortran` (suivant les normes de ces langages) recompilé pour cette architecture cible tournera sans trop de souci sur cette carte. L'usage de R sur le Xeon Phi est encore très marginal et se limite à l'interfaçage avec la bibliothèque MKL.

Plus généralement, le développement et l'utilisation d'architectures massivement parallèles va aller en s'amplifiant. Les supercalculateurs du top 500 sont dans un monde où les performances des processeurs séquentiels n'évoluent plus, seule la multiplication du nombre de cœurs permet de gagner en puissance de calcul. La problématique du parallélisme est donc aujourd'hui incontournable.

A.5 Exemples d'utilisation de Rcpp avec la fonction `cppfunction` du package inline

Apprendre par l'exemple, c'est ce que propose cette section qui montre des exemples simples de mise en pratique de l'interfaçage entre R et `C++`. Il est bien sûr recommandé de consulter la documentation associée à **Rcpp** (Eddelbuettel (2013) par exemple) pour aller plus loin. Voici différentes configuration :

— des scalaires en entrée/sortie de fonction :

```
cppFunction('int sixtythree(){
 return(63);
}
')
sixtythree()
[1] 63
```

```
cppFunction('double add(double x, double y){
 return(x*y);
}
')
add(1.5, 6.3)
[1] 9.45
```

— des vecteurs (`NumericVector, IntegerVector, LogicalVector`) en entrée de fonction :

```
cppFunction('
double crossprodC(NumericVector x, NumericVector y){
 double crp = 0.;
 int n = x.size();
 for(int i=0; i<n; i++) crp += x[i]*y[i];
 return(crp);
}
')
crossprodC(c(1.5,6.3), c(0.3,4.3))
[1] 27.54
```

— une matrice (`NumericMatrix, IntegerMatrix`) en sortie de fonction :

```
cppFunction('
NumericMatrix subMatrix(LogicalVector x, NumericMatrix mat){
 int nr = mat.nrow(), nc = mat.ncol();
 if (x.size() != nr)  warning("Inconsistent dimensions");
 int nr2 = 0;
 for(int i=0; i<nr; i++) if(x[i]) nr2++;
 NumericMatrix mat2(nr2,nc);
 int i2 = 0;
 for(int i=0; i<nr; i++)
  if(x[i]){
   for(int j=0; j<nc; j++)
    mat2(i2,j) = mat(i,j);
   i2++;
  }
 return(mat2);
}
')
subMatrix(c(TRUE,FALSE,TRUE), rbind(c(1.,2),c(3.,4.),c(5.,6.)))
     [,1] [,2]
[1,]    1    2
[2,]    5    6
```

— une fonction de modification « *in-place* » (*i.e.* sans copie) d'un vecteur :

```
cppFunction('void squareInPlace(IntegerVector x) {
 int n = x.size();
 for(int i = 0; i < n; ++i) x[i] += x[i]*x[i];
}
')
x = c(1.5, 6.3)
squareInPlace(x)
x
[1]  3.75 45.99
```

Bibliographie

Amdahl G.M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. Dans *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pp. 483–485. ACM, New York, NY, USA.

Beck K. & Andres C. (2004). *Extreme Programming Explained : Embrace Change (2Nd Edition)*. Addison-Wesley Professional.

Beck K., Beedle M., van Bennekum A., Cockburn A., Cunningham W., Fowler M., Grenning J., Highsmith J., Hunt A., Jeffries R., Kern J., Marick B., Martin R.C., Mellor S., Schwaber K., Sutherland J. & Thomas D. (2001). Manifesto for agile software development.

Bioconductor (2016). Unit testing guidelines.

Brucker P. (2001). *Scheduling Algorithms*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 3rd ed.

Bååth R. (2012). The state of naming conventions in r. *The R journal*, **4/2**.

Dean J. & Ghemawat S. (2008). Mapreduce : Simplified data processing on large clusters. *Commun. ACM*, **51**(1), 107–113.

Eddelbuettel D. (2013). *Seamless R and C++ Integration with Rcpp*. Springer.

Graham R.L. (1969). Bounds on multiprocessing timing anomalies. *SIAM JOURNAL ON APPLIED MATHEMATICS*, **17**(2), 416–429.

Grama A., Gupta A., Karypis G. & Kumar V. (2003). *Introduction to Parallel Computing*. Chapman and Hall/CRC.

Gustafson J.L. (1988). Reevaluating amdahl's law. *Commun. ACM*, **31**(5), 532–533.

Hager G. & Wellein G. (2011). *Introduction to High Performance Computing for Scientists and Engineers*. Chapman and Hall/CRC.

IDRIS-CNRS (2016). Cours de mpi.

Jacob P., Robert C.P. & Smith M.H. (2011). Using parallel computation to improve independent metropolis–hastings based estimation. *Journal of Computational and Graphical Statistics*, **20**(3), 616–635.

Josuttis N.M. (2012). *The C++ Standard Library - A Tutorial and Reference, 2nd Edition*. Addison Wesley.

Kernighan B. & Plauger P. (1978). *The Elements of Programming Style*. McGraw-Hill.

Knuth D.E. (1974). Computer programming as an art. *Commun. ACM*, **17**(12), 667–673.

Lim A. & Tjhi W. (2015). *R High Performance Programming*. PACKT.

Matloff N. (2015). *Parallel Computing for Data Science : With Examples in R, C++ and CUDA*. Chapman and Hall/CRC.

McCallum Q.E. & Weston S. (2011). *Parallel R*. O'Reilly Media, Inc.

Muller J.M., Brisebarre N., de Dinechin F., Jeannerod C.P., Lefèvre V., Melquiond G., Revol N., Stehlé D. & Torres S. (2010). *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston. ACM G.1.0 ; G.1.2 ; G.4 ; B.2.0 ; B.2.4 ; F.2.1., ISBN 978-0-8176-4704-9.

OpenMP-ARB (2016). Openmp compilers.

RCoreTeam (2016). Writing r extensions.

Sutter H. & Alexandrescu A. (2004). *C++ Coding Standards : 101 Rules, Guidelines, and Best Practices*. Addison–Wesley.

Visser M., McMahon S.M., Merow C., Dixon P.M., Record S. & Jongejans E. (2015). Speeding up ecological and evolutionary computations in r ; essentials of high performance computing for biologists. *PLoS computational biology*, **11**(3), e1004140–e1004140.

Wickham H. (2011). testthat : Get started with testing. *The R journal*, **3/1**.

Wickham H. (2014). *Advanced R*. Chapman and Hall/CRC, 1 ed.

Index