Des choix IT éco-responsables pour une meilleure modélisation du risque sismique

Adrien Pothon- *Earthquake risk expert - PhD (Earth Science)*

Reda Jarir- *CEO and co-founder at URLab*

Sommaire

1. Introduction et contexte

Introduction

Contexte:

Modélisation de l'incertitude de l'aléa sismique avec une corrélation spatiale, appelée incertitude de première espèce.

Approche utilisée:

Analyse des résidus intra-événement pour estimer la corrélation spatiale

Modèle utilisé

Modèle Goda & Atkinson - équation fondamentale*:

$$\rho_{\varepsilon}(\Delta, T) = \max\left[\gamma(T).\, e^{\left(-\alpha(T)\Delta^{\beta(T)}\right)} + \gamma(T) - 1, 0\right]$$

- Δ : distance de séparation entre stations
- T : période de vibration
- α, β, γ : paramètres du modèle

Implémentation du modèle:

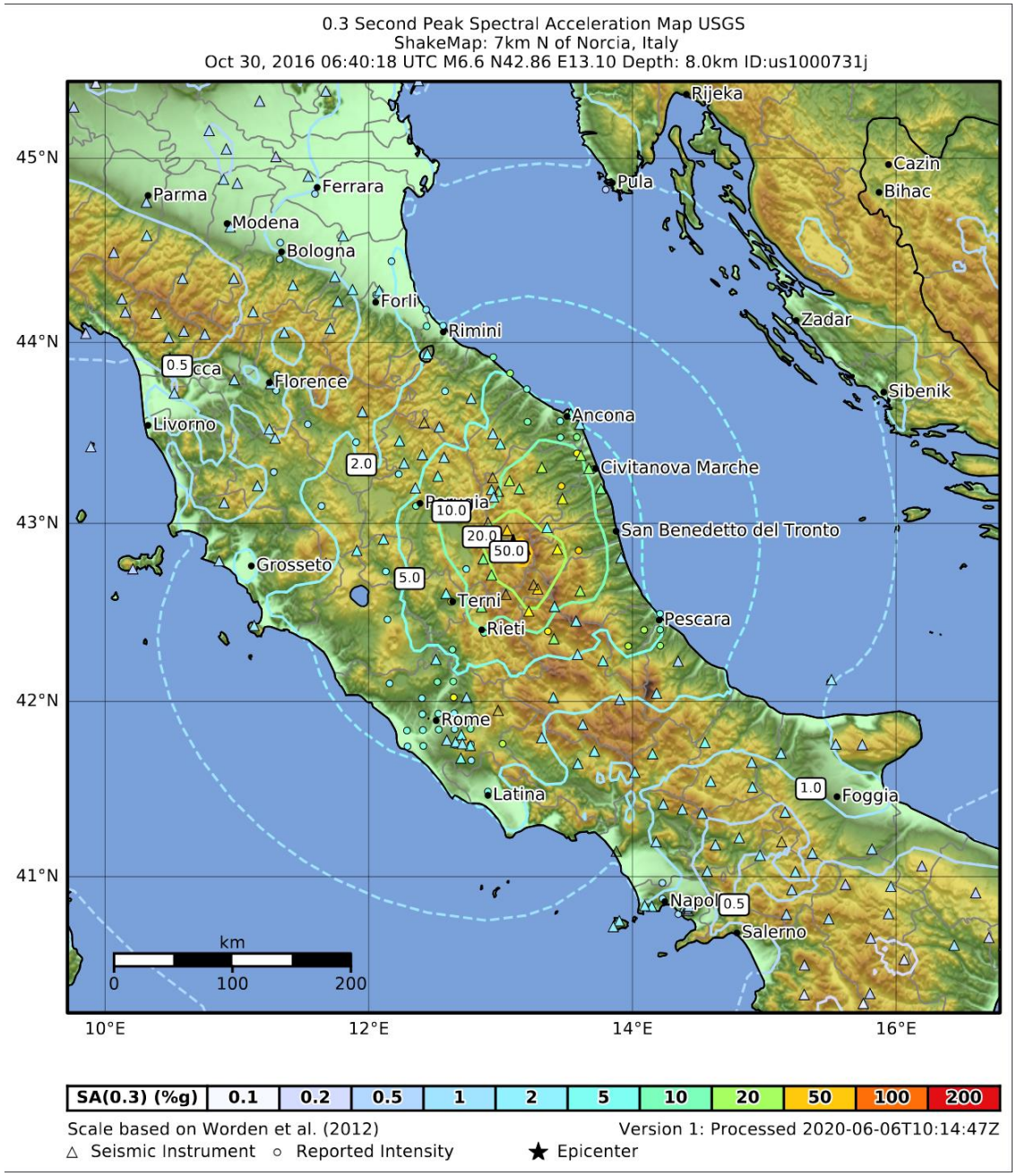
Implémentation sous R et calcul de la distance d'haversine à l'aide du package « geosphere »

Simulation à itérer
10 000 à 50 000 fois

* Goda, K., & Atkinson, G. M. (2010). Intraevent spatial correlation of ground-motion parameters using SK-net data. Bulletin of the Seismological Society of America, 100(6), 3055-3067.

Exemple utilisé pour l'étude

- Séisme du 30 octobre 2016, de magnitude importante (6,5 sur l'échelle de Richter) à proximité de la ville de Norcia (Ombrie, Italie du centre)
- Nombre de points de coordonnées: **134,078 points**



Code initial et défis

```
54 tirages<-as.matrix(tirages)
55 carte=rep(NA,nrow(tirages))
56
57
58 t1 <- Sys.time()
59
60 for(i in 1:nrow(tirages)){
61
62   d1 =distHaversine(tirages[i,c(1,2)],coordonnees,6378.137)
63   a_1j = pmax(5*exp(-0.06*d1^0.283)-4,0)
64
65   carte[i]<-sum(tirages[,3]*a_1j)/sqrt(sum(a_1j^2))*U[i]
66 }
67 t2 <- Sys.time()
68 t2-t1
69
70 rascar = raster(nrow=sum(shakemap[,"Longitude"]==shakemap[1,"Longitude"]), nc
71 values(rascar) <- exp(carte)
72 x11();plot(rascar)
```

Goulots d'Étranglement Actuels

Temps d'exécution : $O(n^2)$

Calculs réalisés

- Distance Haversine : n calculs par itération
- Exponentielles et puissances : n calculs par itération
- Sommes et normalisations : n opérations par itération

Temps total = $O(n) \times O(n) = O(n^2)$
Espace mémoire minimal = $O(n)$

Avec n = 134,078 points : **$134{,}078^2 \approx$ 18 milliards operations**

Temps d'exécution sur puce Apple M2 Max : 2h

Défis

Besoin de réduire le temps d'exécution à quelques minutes avec les considérations suivantes:

- ❑ Garder le code facile à maintenir et à la portée du modélisateur
- ❑ Le coût des simulations doit être raisonnables et limiter l'empreinte carbone

2. Pourquoi plus de CPU/RAM n’est pas une solution

Etude théorique

Améliorations théoriquement limitées

- Augmenter le CPU : gain linéaire maximum
- Augmenter la RAM : évite swapping mais ne résout pas $O(n^2)$
- Améliorer le cache : impact marginal

Croissance quadratique du temps de calcul

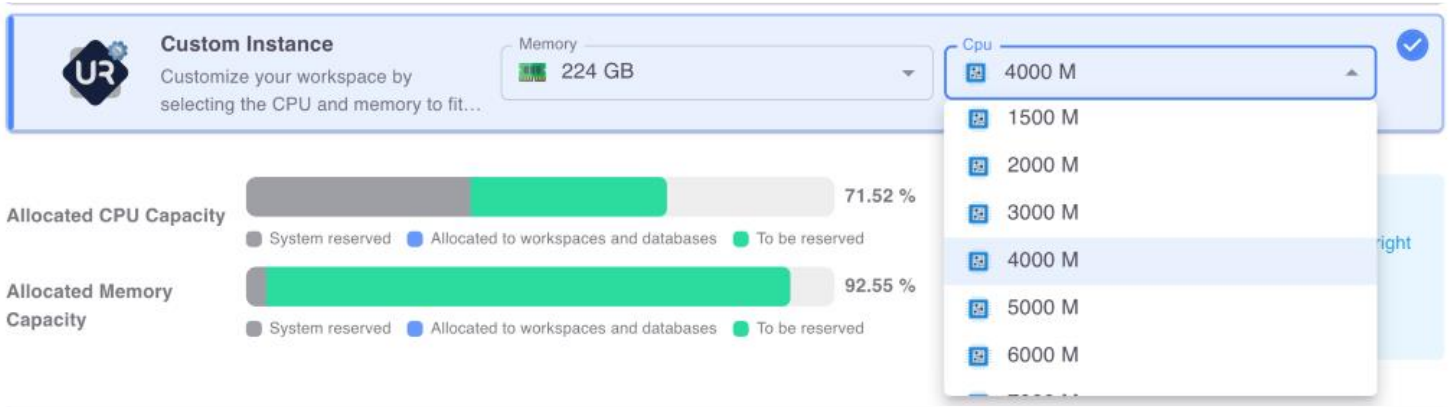
Taille de la carte	134 k points	268 k points
Nb d'opérations	~18 milliards	~72 milliards
Temps de calcul*	2h	8h

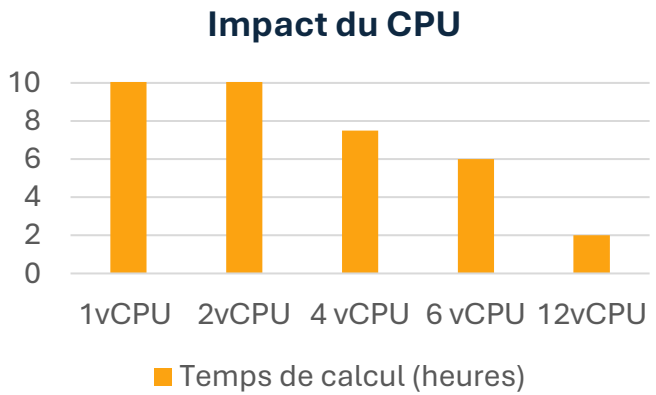
Nécessité de réaliser des milliers de scenarii stochastiques sur des cartes de dimensions différentes

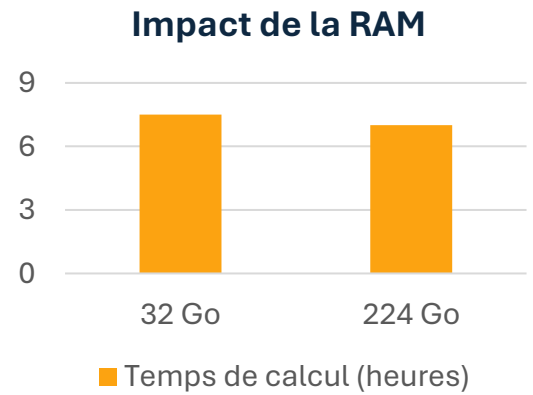
La réduction du temps de calcul en se basant uniquement sur l'amélioration hardware reste insuffisante pour rendre réalisable la génération de milliers de scénarios stochastiques

*Test réalisé sur puce Apple M2 Max – 12 cœurs, 32 Go RAM

Expériences réalisées sur le CPU/RAM







Confirmation de la nécessité d'intervenir au niveau du software pour résoudre le problème

3. La vectorisation et ses limites

Vectorisation sous R

Approche

- Élimination de la boucle for
- Utilisation d'opérations matricielles

Changements Techniques

- Utilisation de apply() pour les calculs de distance
- Opérations matricielles directes avec colSums()
- Optimisation des allocations mémoire

Résultat

Échec pour cause de limitation physique :

RAM nécessaire >224 GO

- Mémoire minimale : 287.6 GB
- Copies temporaires : ~3-4x la taille initiale
  - ➢ Mémoire totale nécessaire : ~860-1150 GB

```r
coordonnees <- coordinates(raster_U)


t1 <- Sys.time()

# Calcul des distances de Haversine pour toutes les coordonnées à la fois
d1_matrix <- apply(tirages[, c(1, 2)], 1, function(x) distHaversine(x, coordonnees, r = 6378.137))

x<-as.data.frame(d1_matrix)

# Calculer le facteur aj pour toutes les distances
aj_matrix<-pmax(5 * exp(-0.06 * d1_matrix^0.283) - 4, 0)

# Calcul de la carte de manière vectorisée
carte_vect <- colSums(tirages[, 3]* aj_matrix) / sqrt(colSums(aj_matrix^2)) * values(raster_U)
carte_vect<-unlist(carte_vect)



rascar_vect =  raster(nrows = nrow(raster_SM), ncols = ncol(raster_SM), xmn = xmin(raster_SM), xmx = xmax(raster
values(rascar_vect) <- exp(carte_vect)
plot(rascar_vect)

rasfin = raster(nrows = nrow(raster_SM), ncols = ncol(raster_SM), xmn = xmin(raster_SM), xmx = xmax(raster_SM),
values(rasfin) <- exp(carte_vect) * values(raster_SM)/100
plot(rasfin)
```

- ❑ La vectorisation sous R exploite exclusivement la RAM. La taille requise n'est pas réalisable sachant que la RAM d'un serveur standard varie ente 256Go-512Go.
- ❑ Besoin d'explorer la vectorisation sur d'autre outils où le disque peut être utiliser pour compenser la limite de la RAM.

DuckDB : Promesses vs Réalité

Promesses:

« DuckDB runs analytical queries at blazing speed thanks to its columnar engine, which supports parallel execution and **can process larger-than-memory workloads** ».

Source: duckdb.org

Objectif:

Contourner la limitation liée à la RAM pour l'opération de vectorisation en exploitant le disque (opération I/O gérées automatiquement par DuckDB)

Commande utilisée

con <- dbConnect(duckdb::duckdb(), dbdir = "./db.duckdb", config = list(memory_limit = " 200GB"))

```
# 7. Calcul des distances de Haversine dans DuckDB
query_distances <- "
SELECT t1.id_coord AS id_tirage, t2.id_coord AS id_coord,
       t1.Longitude AS Longitude1, t1.Latitude AS Latitude1, t1.simu,
       t2.Longitude AS Longitude2, t2.Latitude AS Latitude2,
       6378.137 * 2 * ASIN(SQRT(
           POWER(SIN(RADIANS((t2.Latitude - t1.Latitude) / 2)), 2) +
           COS(RADIANS(t1.Latitude)) * COS(RADIANS(t2.Latitude)) *
           POWER(SIN(RADIANS((t2.Longitude - t1.Longitude) / 2)), 2)
       )) AS distance_km
FROM tirages t1 CROSS JOIN coordonnees t2
"

# Exécuter la requête et créer une table
dbExecute(con, paste("CREATE TABLE distances AS", query_distances))

# 8. Calcul du facteur aj et des sommes nécessaires dans DuckDB (inchangé)
dbExecute(con, "
CREATE TABLE calculs AS
SELECT id_coord,
       SUM(simu * GREATEST(5 * EXP(-0.06 * POWER(distance_km, 0.283)) - 4, 0)) AS numerateur,
       SQRT(SUM(POWER(GREATEST(5 * EXP(-0.06 * POWER(distance_km, 0.283)) - 4, 0), 2))) AS denominateur
FROM distances
GROUP BY id_coord
")
```

I/O Intensif

Temps d'accès :

RAM : ~100 ns vs SSD : ~100 µs vs HDD : ~10 ms

Résultats des tests

- ✓ Performances décevantes sur disque
- ✓ Complexité accrue du code

4. Intégration d’un langage compilé

Langage interprété Vs compilé

Limitations des langages interprétés

- ✓ Exécution ligne par ligne ralentissant le traitement
- ✓ Typage dynamique nécessitant des vérifications constantes
- ✓ Surcharge mémoire due à la gestion automatique des ressources

Avantages des langages compilés

- ✓ Traduction directe en code machine
- ✓ Optimisations au niveau du compilateur
- ✓ Performances jusqu'à 100x supérieures pour les calculs intensifs

Pourquoi Julia ?	Bénéfices de l'intégration R-Julia
✓ Performance proche du C/C++ ✓ Syntaxe moderne similaire à R/Python ✓ Optimisé pour le calcul scientifique	✓ Accélération des parties calculatoires intensives ✓ Conservation de R pour l'analyse de données et la visualisation

Test du langage Julia –Séquentielle

```julia
# Initialiser Julia
julia_setup()

# Définir la fonction de calcul complète en Julia
julia_command("
function compute_carte(tirages, coordonnees, uncertainty_values, radius=6378.137)
    # Définition de la fonction Haversine
    function haversine(lat1, lon1, lat2, lon2)
        dlat = (lat2 - lat1) * π / 180
        dlon = (lon2 - lon1) * π / 180
        a = sin(dlat / 2)^2 + cos(lat1 * π / 180) * cos(lat2 * π / 180) * sin(dlon / 2)^2
        c = 2 * atan(sqrt(a) / sqrt(1 - a))
        return radius * c
    end

    # Initialiser le vecteur pour les résultats
    n = size(tirages, 1)
    carte = fill(NaN, n)

    # Boucle pour calculer les valeurs de 'carte'
    for i in 1:n
        d1 = [haversine(tirages[i, 2], tirages[i, 1], coordonnees[j, 2], coordonnees[j, 1]) for j in 1:n]
        aj = max.(5 .* exp.(-0.06 .* (d1 .^ 0.283)) .- 4, 0)
        carte[i] = sum(tirages[:, 3] .* aj) / sqrt(sum(aj .^ 2)) * uncertainty_values[i]
    end

    return carte
end
")
```

1. Intégration R-Julia

- Utilisation simple via le package « JuliaCall »
- Conservation de l'environnement R existant
- Transition fluide entre R et Julia

2. Optimisations Naturelles

- Compilation Just-In-Time (JIT)
- Types statiques
- Vectorisation native

Résultat du test

Temps d'exécution: 14 mins*

*Test réalisé sur puce Apple M2 Max – 12 cœurs, 32 Go RAM

Test du langage Julia –Parallèle

```
57
58  # Detect the number of CPU cores on the Mac
59  num_cores <- detectCores()
60  print(paste("Number of CPU cores detected:", num_cores))
61
62  # Set the JULIA_NUM_THREADS environment variable
63  Sys.setenv(JULIA_NUM_THREADS = num_cores)
64
65  # Initialize Julia with multi-threading
66  julia_setup()
67
68  # Define the multi-threaded Julia function
69  julia_command("using Base.Threads")
```

```
85
86      # Parallel loop to compute 'carte' values
87      @threads for i in 1:size(tirages, 1)
88          # Calculate distances using the Haversine formula in parallel
89          d1 = [haversine(tirages[i, 2], tirages[i, 1], coordonnees[j, 2], coordon
90          aj = max.(5 .* exp.(-0.06 .* (d1 .^ 0.283)) .- 4, 0)
91
92          # Compute the 'carte' value for this iteration
93          carte[i] = sum(tirages[:, 3] .* aj) / sqrt(sum(aj .^ 2)) * uncertainty_v
94      end
```

Configuration simple
Identifier les 12 cœurs disponibles

Parallélisation simple
@threads for i in 1:size(tirages, 1)

Résultat du test

Temps d'exécution: 2 mins*

- Parallélisation simple et efficace
- Langage conçu pour les calculs scientifiques et facile à lire
- Performances exceptionnelles

*Test réalisé sur Macbook M2 Max – 12 cœurs, 32 Go RAM

5. Conclusion

Performance comparative

Critère	R Original	R Vectorisé	DuckDB	Julia Seq.	Julia Para.
Simplicité code	★★★★★	★★★	★★	★★★★	★★★★
Maintenance	★★★★★	★★★	★★	★★★★	★★★★
Performance	★★	★★★**	★★★**	★★★★	★★★★★
Scalabilité	★★★	★**	★★**	★★★★	★★★★★
Utilisation RAM	★★★★★	★**	★★**	★★★★	★★★★
Temps d'exéc	2h14min*	-	-	14 min*	2 min*

* Test réalisé sur puce Apple M2 Max – 12 cœurs, 32 Go RAM

** Limité par la taille des données

Impact à grande échelle

Ressources annuelles pour 10 000 simulations

Les tests ont été réalisés sur une puce Apple M2 Max. Pour une machine virtuelle équivalente en puissance sur le cloud :

Série Dsv5 : Configuration D32s v5

Spécifications :

- 32 vCPU
- 128 Go de RAM
- Processeur Intel Xeon Platinum 8370C
- Tarif horaire : 1,54 USD

Méthode	Code initial	Julia parallèle	Economie
Temps	~ 833 jours	~ 14 jours	- 819 jours
Coût	30 788 $	517 $	- 30 270 $

6. Q&R