

AD5L-L1 Scalable architecture

Rapport de projet (Mandelbrot load-balancer)

Professeur : Monsieur Quentin Lurkin

Groupe :

- Jonathan Miel (16013)
- Charles Vandermies (15123)

Table des matières

Introduction.....	3
Fonctionnement général.....	3
Résultat.....	4
Outils utilisés	4
Load-balancing	4
Configuration.....	5
Mise en place.....	6
Vérification du bon fonctionnement.....	6
Un seul serveur.....	6
Six serveurs.....	6
Limitations.....	7
Pistes d'améliorations	8
Optimisation d'Nginx.....	8
Amélioration du Hardware.....	8
Optimisation du calcul de Mandelbrot	8
Conclusion	8

Introduction

L'ensemble de Mandelbrot est un ensemble de nombres complexes très connu dans le monde mathématique. Pour chaque point du repère, un calcul complexe fastidieux doit être effectuée afin de déterminer s'il fait partie de l'ensemble ou pas. Il n'y a pas de raccourci, et chaque point est indépendant l'un de l'autre.

Il s'agit donc d'un cas parfait pour exploiter la parallélisation, puisque celle-ci marche au mieux quand aucune valeur ne dépend d'une autre, évitant ainsi tout temps mort.

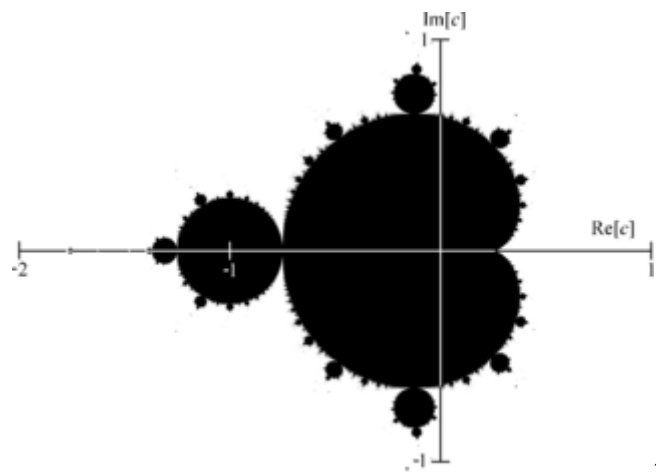
Dans cet exercice, il est demandé de calculer l'ensemble de Mandelbrot, depuis un site web, en envoyant une requête pour chacun de ses points. Pour optimiser ce processus, il faut profiter d'un système de *load-balancing* qui a pour objectif de répartir le flux en plusieurs serveurs, qui pourront dès lors travailler en parallèle.

Fonctionnement général

Depuis le frontend, l'utilisateur peut choisir la résolution de l'ensemble de Mandelbrot, en nombre de pixels. Il peut également choisir combien il y aura d'itérations pour la génération de l'ensemble. Au plus le nombre d'itérations est grand, au plus fin le résultat sera, aux dépens d'une puissance de calcul plus élevée.

Choose image size: Choose number of iterations:

Une fois que l'utilisateur a choisi les paramètres qui lui conviennent, il peut appuyer sur le bouton de validation. A ce moment-là, le programme va construire l'ensemble des requêtes qu'il faudra envoyer via l'API. Pour chaque pixel, les valeurs réelles et imaginaires sont calculées, en son centre. Ce calcul est effectué en considérant un repère complexe dont les valeurs en imaginaires vont de -1 à 1 et réelles de -2 à 1. De cette manière, nous avons la forme typique de l'ensemble de Mandelbrot.



Pour calculer les coordonnées de chaque pixel, il faut utiliser le rapport entre la résolution choisie par l'utilisateur et la taille du repère.

Ensuite, cet ensemble de requêtes est divisé en plusieurs groupes plus petits. Cela permet d'éviter d'être bloqué par le navigateur pour trop d'envois simultanés. Cette problématique sera développée plus tard dans le rapport.

¹ <https://www.lucaswillems.com/fr/articles/3/symetrie-de-lensemble-de-mandelbrot>

Avant d'arriver au *backend*, les requêtes passent par une adresse intermédiaire qui va faire office de *load-balancer*, qui va s'occuper de distribuer les requêtes parmi plusieurs serveurs.

Chaque instance de serveur va calculer si le point reçu fait partie de l'ensemble de Mandelbrot. Si c'est le cas, il renvoie la couleur « noir », sinon, il renvoie une couleur de valeur RGBA, déterminée en fonction du nombre d'itérations qu'il a fallu pour découvrir qu'il n'appartenait pas à l'ensemble.

Une fois toutes les réponses obtenues, le *front-end* les rassemble pour former une image qu'il affiche dès lors à l'écran, en plus du temps qu'il a fallu pour l'afficher.

Résultat



Time to render : 2371.75 seconds

Il a fallu 40 minutes pour rendre cette image. C'est beaucoup mais il ne faut pas oublier que cela constitue $480 \times 320 = 153\,600$ requêtes, avec pour chacune, jusqu'à 100 itérations pour le calcul.

Cela correspond à environ 15 millisecondes de temps de calcul par pixel.

Outils utilisés

Pour réaliser l'exercice, nous avons le choix des technologies. Nous avons décidé de partir avec React pour le Frontend, Axios pour effectuer des requêtes http et NodeJS (plus précisément Express pour la création de serveurs) pour le Backend.

C'est donc une API REST relativement simple puisqu'elle comprend une seule requête GET avec quelques *query params* :

GET / ?real=...&imaginary=...&iterations=...

- real : la valeur réelle du pixel
- imaginary : la valeur imaginaire du pixel
- iterations : le nombre d'itérations pour le calcul de l'appartenance à l'ensemble de Mandelbrot

Notes : la valeur d'itérations est envoyée à chaque requête mais il aurait été préférable de ne l'envoyer qu'une fois, au début, pour alléger davantage les requêtes.

Pour le load-balancer, c'est Nginx que nous avons choisi.

Load-balancing

Le load-balancing est un procédé qui permet de distribuer un trafic réseau parmi plusieurs serveurs, et ce, de la manière la plus équilibrée possible. C'est donc un intermédiaire qui va décider, en fonction de l'état actuel de chaque serveur, où envoyer la requête qu'il vient de recevoir.

Le front-end envoie alors toutes ses requêtes à l'adresse et au port paramétrés dans le fichier de configuration de Nginx. Dans le cadre de l'exercice, il s'agit simplement de l'adresse *localhost* avec le port http 80.

Par défaut, l'algorithme du load-balancing est le Round-Robin. Nginx propose d'autres algorithmes qui peuvent correspondre à différents cas d'utilisations :

- Least-Time
- Least-Connections
- Ip hash
- ...

Dans le cadre de l'exercice, le plus efficace est Round-Robin car il s'agit de l'algorithme recommandé lorsqu'il n'y a pas de grosses différences de puissance entre les différents serveurs². Comme tous les serveurs tournent sur le même ordinateur, ils ont à priori tous les même puissances (en considérant que Node JS distribue équitablement la puissance de calcul entre toutes les instances de serveurs générés)

Pour vérifier cette affirmation, nous avons effectué un test : générer un ensemble de Mandelbrot de 120x80 avec 10 000 itérations. Lorsque Round-Robin est configuré, le résultat est obtenu en **50.71 s** et avec Least-Connections : **60.36 s** (l'algorithme de Least-Time ne fonctionnait pas en configuration pour une raison inconnue). Les algorithmes de Hash sont, eux, sélectionnés lorsqu'une persistance de session est nécessaire, ils n'ont donc pas été testés.

Configuration

Comme c'est le Round-Robin qui s'avère être le plus efficace, il n'y a rien besoin d'ajouter dans la configuration comme c'est celui sélectionné par défaut.

La seule chose à faire c'est de créer le lien entre l'adresse du *load-balancer* et les serveurs *Express* qui vont traiter les requêtes :

```
upstream app_servers {
    server localhost:3004;
    server localhost:3005;
    server localhost:3006;
    server localhost:3007;
    server localhost:3008;
    server localhost:3009;
}

server {
    listen      80;
    server_name localhost;
    location / {
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header Host      $http_host;
        proxy_pass      http://app_servers;
    }
}
```

Dans *server* est spécifié l'adresse du *load-balancer* ainsi que son port. Il est aussi indiqué à quel sous-adresse est associé quel groupe de serveurs. Ici dans *location*, / correspond au proxy http://app_servers dont la liste est précisée plus haut. Ce sont 6 serveurs avec chacun, forcément, un port différent. Le choix du nombre 6 n'est pas anodin, il s'agit du nombre de cœurs disponible dans le processeur de l'ordinateur utilisé pour les tests.

² <https://www.nginx.com/blog/choosing-nginx-plus-load-balancing-techniques/>

Mise en place

Pour lancer le processus, il faut d'abord exécuter nginx, et pour ce faire il suffit de lancer l'exécutable. Ensuite, il faut générer les serveurs avec les ports adéquats.

```
[nodemon] restarting due to changes...
[nodemon] starting 'node server.js'
server is up on port 3004

[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node server.js 3005'
server is up on port 3005

[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node server.js 3006'
server is up on port 3006

[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node server.js 3007'
server is up on port 3007

[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node server.js 3008'
server is up on port 3008

[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node server.js 3009'
server is up on port 3009
```

Vérification du bon fonctionnement

Pour vérifier que chaque serveur est utilisé à son plein potentiel, il suffit de faire la même requête avec tous les serveurs ouverts ou un seul.

Cette vérification était primordiale car au début de nos recherches, nous avons essayé de faire nous même un load-balancer, assez basique, qui distribuait les requêtes à chaque serveur, chacun son tour. Avec quelques *logs*, cela semblait marcher, chaque serveur semblait occupé, mais en réalité, ils ne travaillaient pas en parallèle, mais chacun à leur tour. Le load-balancer attendait que le serveur précédent ait fini son travail avant de passer la requête suivante au prochain serveur.

Pour vérifier que le load-balancer effectue bien ce qu'on lui demande, nous allons comparer le temps d'exécution pour l'affichage d'un ensemble de Mandelbrot d'une résolution de 60x40 avec 10 000 itérations. Le nombre d'itérations est grand car c'est ce paramètre qui est optimisé par l'utilisation d'un *load-balancer*.

Un seul serveur

Pour ce premier test, nous n'allons ouvrir qu'un seul serveur :

```
server is up on port 3004
[nodemon] restarting due to changes...
[nodemon] starting 'node server.js'
server is up on port 3004

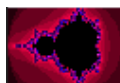
[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node server.js 3005'
server is up on port 3005
^C
C:\Users\jojom\Documents\nodejs\LoadBalancer>

[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node server.js 3006'
server is up on port 3006
^C
C:\Users\jojom\Documents\nodejs\LoadBalancer>

[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node server.js 3007'
server is up on port 3007
^C
C:\Users\jojom\Documents\nodejs\LoadBalancer>

[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node server.js 3008'
server is up on port 3008
^C
C:\Users\jojom\Documents\nodejs\LoadBalancer>

[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node server.js 3009'
server is up on port 3009
^C
C:\Users\jojom\Documents\nodejs\LoadBalancer>
```

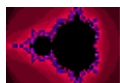


Choose image size: Choose number of iterations:

Time to render : 28.252 seconds

Le temps de rendu est d'environ 30 secondes.

Six serveurs



Choose image size: Choose number of iterations:

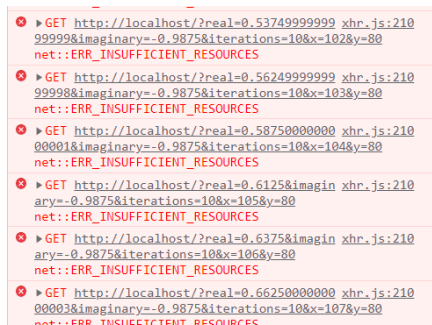
Time to render : 7.286 seconds

Le temps de rendu est d'environ 8 secondes

On voit qu'il y a donc une nette améliorations du temps de calcul.

Limitations

Dans la partie *fonctionnement général*, il est indiqué que les requêtes sont fragmentées en plusieurs groupes. Initialement, toutes ces requêtes étaient envoyées d'un coup. Cela ne posait pas de problème tant que la résolution de l'ensemble de Mandelbrot était faible, 60x40 par exemple. Mais dès qu'une résolution plus élevée était sélectionnée, il y avait des erreurs à l'issues des requêtes HTTP :



```
➤ GET http://localhost/?real=0.53749999999 xhr.js:210
99999&imaginary=-0.9875&iterations=10&x=102&y=80
net::ERR_INSUFFICIENT_RESOURCES

➤ GET http://localhost/?real=0.56249999999 xhr.js:210
99998&imaginary=-0.9875&iterations=10&x=103&y=80
net::ERR_INSUFFICIENT_RESOURCES

➤ GET http://localhost/?real=0.58750000000 xhr.js:210
00001&imaginary=-0.9875&iterations=10&x=104&y=80
net::ERR_INSUFFICIENT_RESOURCES

➤ GET http://localhost/?real=0.6125&imagin xhr.js:210
ary=-0.9875&iterations=10&x=105&y=80
net::ERR_INSUFFICIENT_RESOURCES

➤ GET http://localhost/?real=0.6375&imagin xhr.js:210
ary=-0.9875&iterations=10&x=106&y=80
net::ERR_INSUFFICIENT_RESOURCES

➤ GET http://localhost/?real=0.66250000000 xhr.js:210
00003&imaginary=-0.9875&iterations=10&x=107&y=80
net::ERR_INSUFFICIENT_RESOURCES
```

Il était difficile de savoir d'où venait le problème. Au départ, nous pensions qu'il s'agissait de Nginx qui n'acceptait pas autant de requêtes simultanées. Malgré différents essais de paramétrages, le problème persistait. De plus, il s'est avéré que Nginx pouvait facilement gérer entre 50 000 et 80 000 requêtes par secondes³, mêmes sans être particulièrement optimisé. Comme ce problème apparaissait même pour une image d'une résolution de 120x80 (9600 requêtes <<< 50 000 – 80 000), cette possibilité a été mise de côté.

L'origine du problème vient plutôt du navigateur web qui n'autorise pas autant de requêtes simultanées. C'est une mesure de sécurité pour éviter les attaques DoS (*Denial of Service*) par exemple⁴.

Ici nous envoyons des milliers de requêtes simultanément depuis **un seul** ordinateur pour calculer l'ensemble de Mandelbrot. Le cas d'utilisation classique d'un *load-balancer* est la gestion d'un flux élevé de connexion vers un site web, depuis de **nombreux** ordinateurs. Dès lors, il n'y a pas ce problème de limite de requête par navigateur à prendre en compte en tant normal. Mais dans le cadre de l'exercice, si.

Pour contourner le problème, nous avons donc augmenté le nombre de groupes de requêtes et ajouté un délai. Cela permet de ne jamais dépasser la limite de requêtes concurrentes.

Après plusieurs essais, nous avons décidé d'envoyer les requêtes par groupe de 150. Expérimentalement, c'était un bon équilibre entre rapidité et résolution du problème. Un délai de 50 millisecondes est appliqué entre chaque groupe de requête, valeur trouvée aussi de manière empirique.

Le nombre de groupe est donc calculé en fonction de la résolution choisie. Par exemple, pour une image de 480x320 (153 600 requêtes), il y a 1024 groupes de 150 requêtes qui sont envoyées.

Cela a résolu le problème, aux dépens d'une perte de rapidité, en effet, durant le délai, aucun calcul n'est effectué.

³ <https://github.com/denji/nginx-tuning>

⁴ <https://www.linkedin.com/pulse/why-does-your-browser-limit-number-concurrent-ishwar-rimal>

Pistes d'améliorations

Optimisation de Nginx

Nginx est un outil très complet où il est possible d'optimiser de nombreux paramètres. Des recherches étendues auraient pu permettre d'accélérer la façon dont le load-balancer traite les données. Selon un article Github, il est possible de monter jusqu'à 500-600 000 requêtes par secondes, ce qui aurait permis de se passer de la division des requêtes et des nombreux délais.

Amélioration du Hardware

Comme expliqué plus tôt, 6 serveurs ont été générés, compte tenu du nombre de cœurs physiques du processeur. Avec plus de cœurs, plus de processeurs, plus d'ordinateurs, il serait évidemment possible d'augmenter le nombre de serveurs et donc la puissance de calcul en parallèle. On parle d'*horizontal scaling*.

L'utilisation de processeurs plus puissants est aussi une éventualité, on parle alors de *vertical scaling*, mais on arrive vite à un rapport d'augmentation de prix/performance qui diminue drastiquement. La meilleure solution, la moins chère, est donc de se tourner vers l'*horizontal scaling* en augmentant le nombre de processeurs.

Optimisation du calcul de Mandelbrot

La façon la plus évidente pour accélérer le rendu de l'ensemble de Mandelbrot est simplement d'exploiter sa symétrie ; il suffit de calculer la moitié de l'ensemble et de dupliquer les points, puis d'effectuer une symétrie horizontale.

Conclusion

Ce travail a pris plus de temps qu'espéré. Au départ nous avons mal compris l'énoncé et étions parti sur l'envoi d'une seule requête et optimisation côté *backend* en exploitant les capacités de parallélisation de NodeJS. Mais bien entendu ça n'était pas l'objectif de l'exercice.

L'installation d'Nginx n'était pas évidente mais il a fonctionné sans trop de difficulté. Le vrai problème a été cette limitation imposée par le navigateur web. Problème qui n'aurait pas été rencontré si nous avions dû designer un *load-balancer* pour gérer le flux de requêtes venant de plusieurs ordinateurs au lieu d'un seul.

Ce qui fut contre-intuitif est le fait que la génération de l'ensemble de Mandelbrot est plus rapide en effectuant une seule requête et une parallélisation *backend*. C'est notamment ce qui a causé une certaine incompréhension vis-à-vis de la façon d'aborder l'énoncé.

Néanmoins, ce fut un travail intéressant qui a mis en lumière l'utilisation d'un service qui est habituellement utilisé sur les sites web qui génèrent des trafics élevés. C'est un outil supplémentaire qui est le bienvenu dans les compétences d'un ingénieur en informatique, qui se doit de pouvoir répondre à des demandes de *scaling* d'un site internet, conséquence d'un gain de popularité par exemple.